Emilien Bresson

Thomas Durif

M1-STIC : Projet de Multimédia

Générateur de pages web

Sommaire

Rédigé par Thomas Durif

I. Introduction

a. Rappel du sujet et cahier des charges

Le but de ce projet est de développer un mini générateur de pages web.

Voici une liste des contraintes demandées pour la réalisation du projet :

- La page créée doit répondre aux contraintes du XHTML.
- L'utilisateur doit avoir accès à des listes préétablies pour pouvoir ajouter des éléments à la page ou des éléments de style.
- Supprimer tout JavaScript et style en ligne.
- La page générée doit être utilisable.
- Le langage pour le script server est le PERL.

b. Remarque quant à la répartition des tâches

La répartition des tâches n'a pas été un problème puisque j'ai réalisé ce projet seul. J'ajoute à cela que j'ai aidé bon nombre d'autres groupes à la réalisation de leur projet, que ce soit pour effectuer des choix techniques ou pour résoudre des problèmes rencontrés.

II. Analyse conceptuelle de l'application

a. Aperçu du client/server

On distingue deux parties distinctes pour notre application. La partie cliente, partie sur laquelle l'utilisateur navigue et lui qui permet d'effectuer les différentes actions proposées par l'application. L'autre partie concerne le coté server, qui, à chaque action de l'utilisateur, recevra des commandes pour effectuer quelque chose. Pour communiquer, sans que l'utilisateur s'en aperçoive, l'application va envoyer des informations au server au grâce à l'objet XHR (XMLHttpRequest).

Au départ développé pas Microsoft, il s'agissait d'un objet ActiveX, utilisé pour la première fois dans Internet Explorer 5 en 1998 (sous le nom MSXML). Depuis, le domaine ayant très rapidement évolué, tous les navigateurs embarquent un moteur JavaScript depuis quelques années, permettant d'utiliser nativement un objet HXR. Son utilisation est très simple : on appelle la page de script server avec ou sans paramètres, le server effectue quelque chose, puis le server retourne une information qui peut être traitée en mode texte ou en tant que XML.

Le principal avantage de cet objet réside dans le fait que la partie cliente et la partie server évoluent de manière asynchrone. C'est-à-dire que l'application coté client peu faire une demande qui peut prendre du temps et c'est du coté server que les traitements lourds et longs seront effectués sans que cela gène l'utilisateur. Cette manière de développer une application correspond à ce que l'on appelle l'AJAX (Asynchronus JavaScript and XML), concept fer de lance du Web 2.0. Il s'agit de permettre une expérience utilisateur complexe dans la conception et dans les taches effectuées (par rapport aux applications Web traditionnelles en allant au delà de ce qui est possible de faire simplement) mais légère dans l'utilisation, s'appuyant notamment sur l'expérience qu'ont les utilisateurs sur les applications logicielles.

b. Coté client

On a donc une application plus évoluée qu'une simple page web. D'ailleurs le nom " page " web fait penser à quelque chose de statique, qui une fois affichée ne permet pas de faire grand-chose à part être lue/regardée. A contrario, une application web répondant au concept de l'AJAX est dite **dynamique** ou **riche** (le terme de client riche est parfaitement adapté à la situation). Notre application proposera donc un contenu dynamique à l'utilisateur. Pour cela, on utilise une forte dose de JavaScript. L'expérience utilisateur sera basée sur un ensemble de " boites " qui apparaîtront au dessus de l'application.

La page physique de l'application, répondant aux normes du XHTML, ne représente qu'une très petite partie de l'application. Le plus gros du code réside dans la partie JavaScript. Le sujet ayant demandé de faire abstraction de tout JavaScript en ligne, le plus gros du code est concentré dans le JavaScript.

c. Coté server

Le server est écrit en PERL. Il doit permettre de recevoir des informations en provenance du client, effectuer les traitements en conséquence, et retourner ou non des informations au client. Dans notre cas, il stockera les informations sur les éléments que l'utilisateur utilise pour la création de sa page. La seule chose qui est renvoyée est " ok ", retour qui permet de savoir si tout c'est bien passé. Si autre chose que ok est retourné au client, alors le server à rencontré une erreur. Normalement, un message plus complexe et surtout plus orienté vers l'erreur sera le bienvenue, mais le nombre de source d'erreur dans le code server réalisé n'est pas très élevé, et donc le débuggage rapide.

III. Programmation de l'application

a. Coté client

1. Page de l'application

Il s'agit d'une page en XHTML tout ce qu'il y a de plus basique. Il s'agit du script minimum nécessaire à l'affichage de l'application. Le gros de la page est construit/amélioré par du JavaScript. Le style est complètement tiré d'une feuille de style annexe.

2. JavaScript

Hormis le script de base et l'utilisation de Mootools (abordé au point suivant), le script réalisé comporte un point important : l'utilisation de pseudo-classes, encore appelée pour le JavaScript " prototype de classe ". JavaScript est un langage orienté objets à prototype, c'est-à-dire que les bases du langage et ses principales interfaces sont fournies par des objets qui ne sont pas des instances de classes, mais qui sont chacun équipés de constructeurs permettant de générer leurs propriétés, et notamment une propriété de prototypage qui permet d'en générer des objets héritiers personnalisés. Ces pseudo-classes sont utilisées de deux manières bien différentes :

Premièrement, pour les boites : Comme dit précédemment, l'expérience utilisateur est basée sur des boites qui apparaissent par-dessus l'application et dans ces boites, l'utilisateur rempli des champs et fait des choix dans des zones de sélection. Pour plus de commodité à l'utilisation, toutes les boites sont développées à l'aide de ces pseudo-classes. Si l'on observe la spécification ci-dessous, l'utilisation des classes est plus parlante.

```
function PropertyBox(_sender_id){
this.sender_id = _sender_id;
/**/
   this.CreateElements();
}
PropertyBox.prototype = {
   CreateElements: function () {
   /**/
   },
   Append: function () {
   /**/
   },
   Destroy: function () {
   /**/
   }
}
```

Ainsi, en ajoutant le code fonctionnel à la place des /**/, pour utiliser la boite, on fait comme il suit :

```
// Pour instancier la boite
var ma_boite = new PropertyBox("node-0");

// Pour afficher la boite
ma_boite.Append();

// Pour cacher la boite
ma_boite.Destroy();
```

Si dans Destroy() on a ajouté du code permettant de stocker les valeurs saisies quand la boite était affichée, alors on obtient une boite de dialogue simple à utiliser, plus proche d'une programmation orientée objet et permettant à l'utilisateur de renseigner des informations.

La deuxième utilisation des pseudo-classes nous sert à ranger les informations saisies par l'utilisateur. Comme il s'agit principalement de liste de données, l'utilisation de tableaux simple aurait signifié des tableaux de tableaux de etc… . Il s'agit ici de stocker des listes d'attributs css avec leur valeur saisie par l'utilisateur (il en va de même pour les listes de propriétés de balises). Plutôt que cela, nous avons créé deux pseudo-classes : Une permettant de stocker les deux valeurs dans un même objet et la deuxième permettant de manipuler ces objets à la manière des ArrayList que l'on trouve dans différents langages de programmation évolués.

La classe permettant de stocker le nom de la propriété et sa valeur dans un seul objet :

```
function ListNode(_node, _val){
	this.node = _node;
	this.val = _val;
}

// Utilisation
var mon_noeud = new ListNode("background-color", "#FF0000");

alert(mon_noeud.node);
alert(mon_noeud.val);
```

La classe ArrayList :

```
function ArrayList(){
	this.list = new Array();
}
ArrayList.prototype = {
	Add:function(_data){
	},
	Remove:function(_id){
	},
	Contains:function(_id){
	},
	GetNode:function(_id){
	}
}
```

Le dernier point important dans le script écrit réside dans les trois listes de propriétés, créées pour être affichées dans les boites de dialogue qui seront affichées à l'utilisateur. La première, css_properties, correspond à la liste des propriétés css qu'il est possible de trouver dans la doc du W3School. Cette version concerne le CSS2. Le CSS3 permettant d'utiliser des propriétés propres au navigateur, il est plus difficile d'en faire une liste exhaustive. La deuxième liste correspond à la liste des tags xhtml 1.0 Strict. La dernière liste correspond aux attributs qu'il est possible de mettre sur une balise xhtml (id, class, href, etc …).

3. Mootools

Mootools est une librairie JavaScript compact, modulaire, orienté objet. Grâce à un ensemble de classes et de fonctions compatibles avec les navigateurs web les plus utilisés, Mootools offre une réponse aux problématiques du développement des clients riches. En ce qui concerne le choix de cette librairie, les différents tests que l'on peut trouver en ligne et mon expérience personnelle montre que Mootools allie légèreté des sources, légèreté d'écriture et rapidité d'exécution.

Le principal objet de Mootools utilisé est l'objet **Element**, qui permet de créé un élément dans le DOM de façon remarquablement rapide. Avec le code ci-desous, on a créé un objet **Element** dans le DOM, qui correspond à une balise de lien, à laquelle on a ajouté plusieurs propriétés, des propriétés de style et des événements, le tout en 17 lignes. Voici l'exemple :

```
var myAnchor = new Element('a', {
    'href': 'http://mootools.net',
    'class': 'myClass',
    'html': 'Click me!',
    'styles': {
        'display': 'block',
        'border': '1px solid black'
    },
    'events': {
        'click': function(){
            alert('clicked');
        },
        'mouseover': function(){
            alert('mouseovered');
        }
    }
});
```

L'autre objet utilisé est l'objet **Request**, qui permet d'utiliser un objet XHR, avec encore une fois, une rapidité et une simplicité d'écriture notable.

```
new Request({
	url: "work.pl",
	method: 'post',
	onSuccess: function (Txt, XML) {
		alert('requete terminée');
	},
	onFailure: function (XHR) {
		alert('une erreur est survenue');
	},
}).send("action=edit&id="+_node+"&value="+prop_string);
```

Un des points les plus important dans l'utilisation de cet objet **Request** est le fait de ne plus se préoccupé du type du navigateur. Rien que pour cela, l'atout de **Request** est tout trouvé face à une écriture en JavaScript. Outre les attributs obligatoires d'une requête vers le server, on a ici en très peu de lignes une gestion d'état de la requête allégée en écriture.

b. Coté server

Comme dit plus haut, le server est développé en PERL. Son fonctionnement est basé sur un fichier XML, qui est rempli et édité au fur et à mesure des requêtes de l'utilisateur, en fonction de ses ajouts, modifications, suppressions. On aurait pu placer les informations à stocker par le server dans une base de données, mais le fait d'utiliser un XML permet une portabilité maximum. La DTD du fichier xml est la suivante :

```
<!ELEMENT projet (nodes)>
<!ELEMENT nodes (node*)>
<!ELEMENT node (node*, style?, properties?, text?)>
<!ATTLIST node
	id NMTOKENS  #REQUIRED
	type NMTOKENS #REQUIRED>
<!ELEMENT styles (style*)>
<!ELEMENT style>
<!ATTLIST style
	name NMTOKENS  #REQUIRED
	value NMTOKENS #REQUIRED>
<!ELEMENT properties (property*)>
<!ELEMENT property>
<!ATTLIST property
	name NMTOKENS  #REQUIRED
	value NMTOKENS #REQUIRED>
<!ELEMENT text (#PCDATA)>
```

Un nœud <node> représente une balise dans le fichier de sortie. Exemple, un <node type="a"> créera une balise de lien sur le rendu final. Les nœuds enfants de <styles> représentent les propriétés de style qui seront affectées au nœud en question. Les nœuds enfants de <properties> représentent les propriétés d'une balise (class, href, id, etc …).

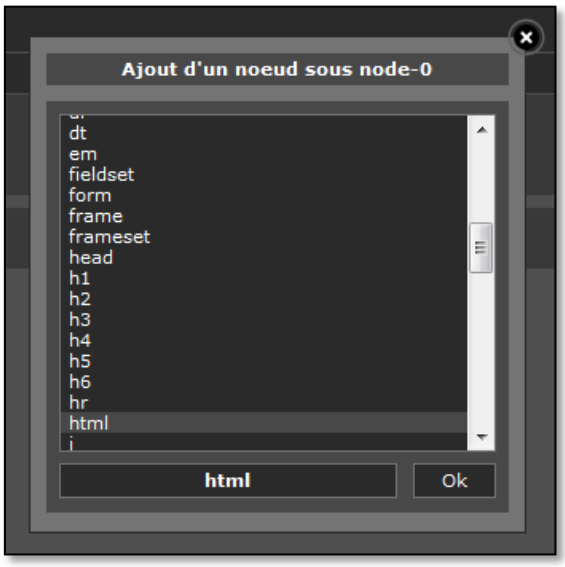
Donc la grosse partie du script server est de détecter le type de la requête venant de l'utilisateur (ajout, modification, suppression, etc …). Voici la liste des actions possibles venant du client :

- init : initialise le fichier xml.
- add : Ajout d'un nœud.
- del : Suppression d'un nœud.
- edit_prop : Modification des propriétés d'un nœud.
- edit_style : Modification du style d'un nœud.
- edit_text : Modification du texte d'un nœud (textnode).
- render : Générer les pages.

Toutes les actions sauf la dernière vont altérer le fichier xml. La dernière va générer les pages xhtml et css correspondant au contenu du xml. Pour observer un exemple du fichier xml et les pages générées à partir de celui-ci, voir en annexe.

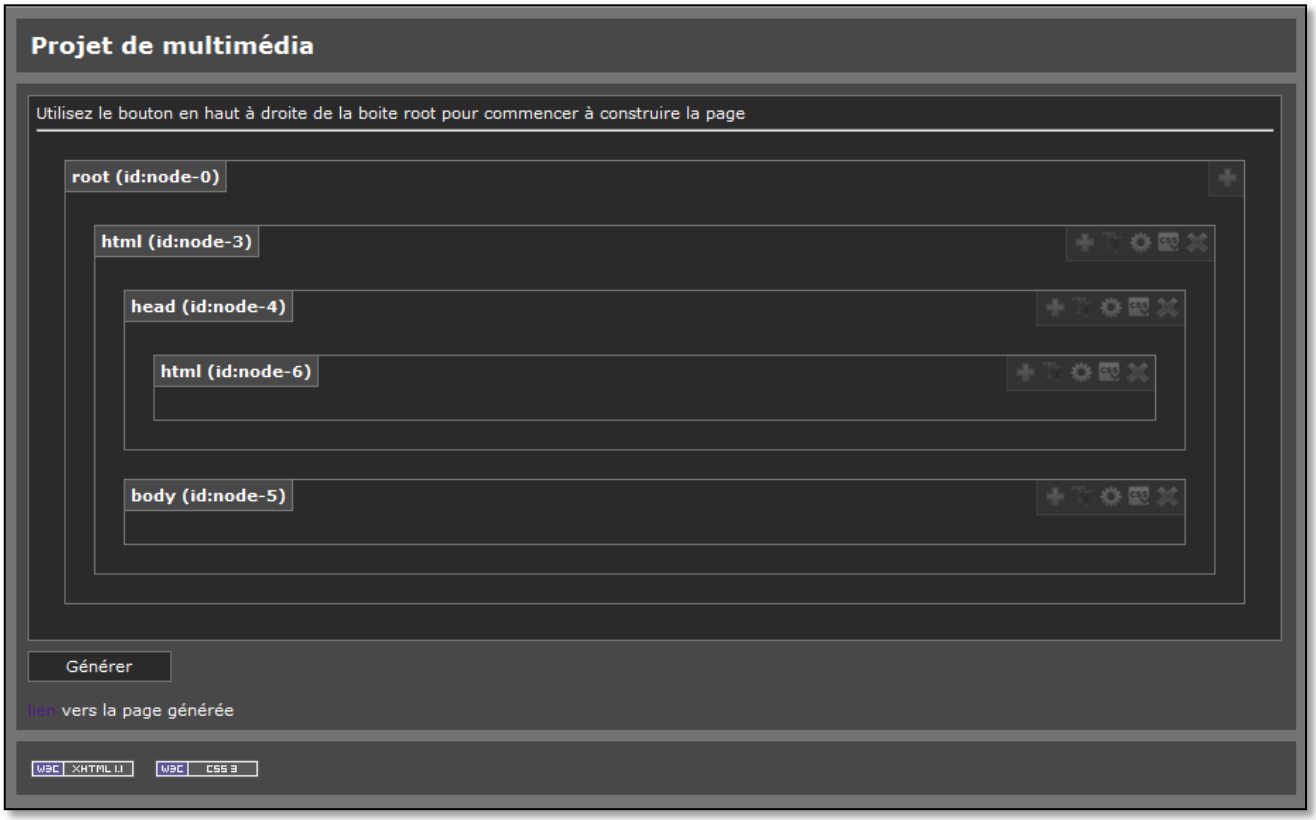
IV. Aperçus de l'application

Voici la page de démarrage de l'application. Il y a trois parties principales, la partie haute et la partie basses sont des parties " visuelles ", qui sont juste la pour améliorer globalement le design de la page. La partie centrale quand à elle est le morceau actif que la page. La seule action logique au démarrage est de cliquer sur le plus en haut à droite du cadre root. Cliquer maintenant sur [Générer] ou [lien vers la page générée] fonctionnera mais lancera une génération vide et montrera une page vide. En cliquant sur le [+], on obtient la boite suivante :



Cette dernière permet de sélectionner le type de balise que l'on veut rajouter. Une fois le type sélectionné et un clique sur ok, on obtient le visuel suivant :

Le nœud est créé. On observe aussi en haut à droite une notification sur l'action qui vient d'être effectuée. La notification, si elle annonce une réussite, disparaitra d'elle-même au bout de quelques secondes.

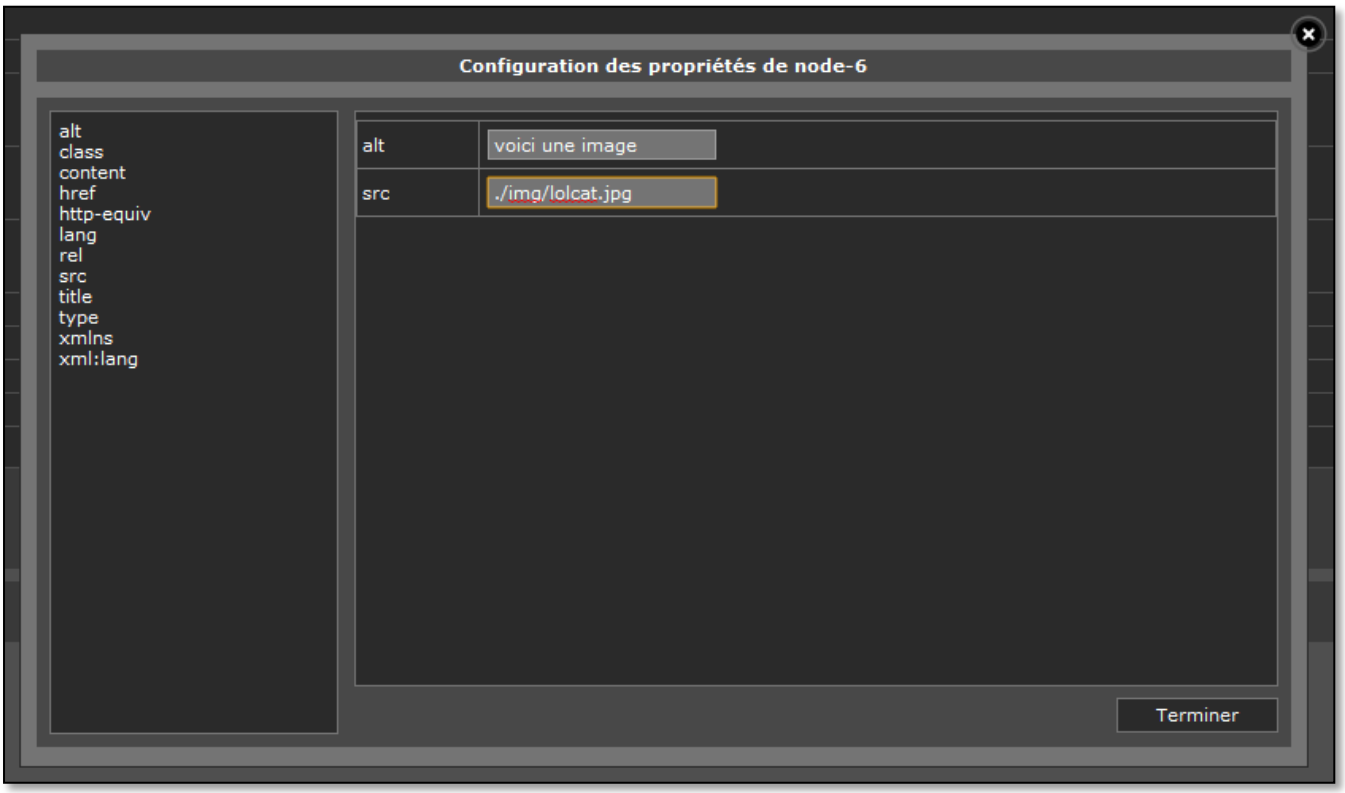


Et on ajoute autan de nœud que l'on veut, où on veut. Une liberté totale est laissée à l'utilisateur. D'ailleurs, la validité de la page finale dépend des connaissances de l'utilisateur. Par exemple, dans l'image précédente, l'utilisateur à placé une balise html dans une balise head ce qui n'a pas de sens.

En regardant bien l'image, on voit qu'à partir du premier nœud, une liste d'outils apparait en haut à droite de chaque nœud ajouté.

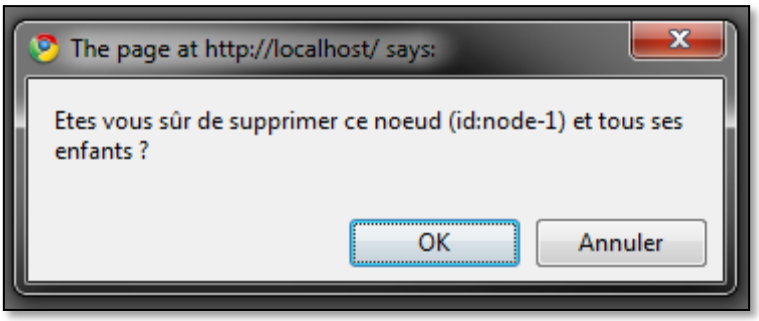
Le premier outil est déjà connu, il s'agit de l'ajout d'un nœud. Le deuxième permet d'éditer le textnode du nœud. Le troisième permet d'éditer les propriétés de balises (href, src, etc …). Le quantième permet d'éditer le style du nœud et enfin le dernier détruit le nœud et ses enfants.

On ne s'arrête pas sur l'édition du textnode car il s'agit juste d'une zone de saisie, avec un bouton ok. Les boites de propriétés et de styles sont plus intéressantes. Pour la boite des propriétés, prenons par exemple le cas d'une image, il faut au minimum renseigner un attribut alt e t un attribut src. Nous allons donc créer ces attributs :



On voit ici l'affichage de la boite d'édition des propriétés. En cliquant sur une des propriétés de gauche, une ligne se crée dans l'espace de droite, permettant à l'utilisateur de saisir une valeur. Quand on a terminé, ces valeurs sont enregistrées pour être utilisées par la suite. La boite de style, qui correspond au 4ème bouton de la barre d'outils, fonctionne de la même façon.

Le dernier bouton permet de supprimer le nœud et tous ses enfants. Une fenêtre de type confirmation est alors affichée :



V. Conclusion et perspectives

La liste qui suit constitue un inventaire des améliorations auxquelles nous avons pensé :

- L'édition du css pourrait être contextuelle, c'est-à-dire ne proposer que les propriétés qu'il est possible de configurer pour une balise donnée.

- Actuellement, il est impossible de déplacer une balise déjà créée. L'intégration d'un système de drag and drop pourrait être intéressante.

- En marge de cela, nous avions d'abord pensé à un système séparé pour la création d'objet d'un coté et le placement de l'autre. C'est-à-dire que l'on pourrait créer des éléments de manière séparée, ne dépendant pas du tout les uns des autres et les rager dans une sorte de panier. Ensuite, on placerait les éléments en drag and drop en les rangeant dans un arbre.

- Un concepteur de composant complexe serait une bonne idée. Dans l'application, on crée un nœud après l'autre. Si c'est une bonne solution pour des enchainements comme div->p->span, ce n'est pas la même chose pour la création d'un tableau par exemple, qui est lourde et répétitive. Donc sans forcement passé par une solution de design, il sera intéressant de proposer des prototypes tout fait pour des composants comme les tableaux ou les listes déroutantes.

- Actuellement, on ne peut éditer qu'une seule page. Il est possible d'ajouter une fonctionnalité permettant de scanner l'intérieur du dossier de rendu des pages et ainsi renvoyer la liste des fichiers déjà créé. Ainsi, l'utilisateur pourrait choisir le fichier à éditer.

- Avec l'ajout de la fonctionnalité d'éditer plusieurs fichiers, il peut devenir intéressant de pouvoir télécharger une sélection de ceux-ci grâce à une interface de sélection et un zip pas exemple.

La réalisation de ce projet nous a permis de réaliser une partie JavaScript assez poussée, en accord avec les concepts d'AJAX, avec l'utilisation de librairies (auxquelles il faut se faire avant de les utiliser) et de pseudo-classes (approche de la programmation objet, chose qui n'est pas spécialement intuitive en JavaScript). Du coté server, on utilise on objet CGI pour récupérer les informations passées par le client, et on stocke au fur et à mesure les informations en cours d'utilisation dans un XML.

Autrement dit, avec ce projet, on concentre donc toutes les caractéristiques d'une RIA (Rich Internet Application).