HTMLgenerator : Comment concevoir un générateur de code HTML en php

par Alain Defrance

Date de publication :

Dernière mise à jour :

Dans cet article il sera démontré comment programmer un générateur de code HTML en PHP.

I - Introduction

Article en cours de rédaction, les modèles UML ne sont pas disponibles et n'aparaissent pas pour le moment.

Tout d'abord je pense qu'il est important d'expliquer l'interêt de généré du code (X)HTML (très facile à comprendre), par du code PHP (bien plus compliqué à mettre en oeuvre).

Le (X)HTML est une structure de donnée hierarchique (comme le XML par exemple), cela fait de lui un langage facile à écrire, cependant lorsqu'il s'agit de le faire manipuler/générer par un programme, il devient plus difficile de s'y retrouver. Le premier problème à résoudre, est de pouvoir adopter une structure de donnée différente qui sera plus facile à manipuler. Il faut faire attention à ne pas perdre à l'idée que le but est tout de même de construire du (X)HTML, il faut donc faire en sorte que le passage de notre structure de donnée soit facile à traduire en (X)HTML.

L'origine de ce générateur par du besoin d'avoir un outil permettant de construire un formulaire rapidement tout en utilisant des technologies longues à mettre en place, tel que l'utilisation d'ajax par exemple.

Afin de faciliter le travail, le générateur à été divisé en deux parties. Une première partie s'occupera de la génération (X)HTML proprement dite, une seconde partie implémentera des fonctionalités comme une vérification automatique de la saisie, génération Ajax automatique, et bien d'autres choses.

Dans cet article nous aborderons uniquement la première partie, mais une suite sera envisageable.

II - La démarche

II-A - Comment je vais faire pour organiser tout ca, le HTML c'est vaste !

Le (X)HTML étant très vaste, il est primordiale ne pas vouloir construire quelque chose de complet.

Cela peut paraitre peu ambicieux, cependant la qualitée d'un programme dépend grandement de sa capacité à évoluer.

Dans notre cas, ce critère est promordiale. Le développement web est en grande évolution et tend à grandir, proposant toujours de nouvelles possibilités.

Il est donc important de privilégier une structure souple et facile à compléter, plutot que vouloir à tout prix quelque chose de complet qui sera de tout manière obsolète dans peu de temps.

II-B - Quelle structure choisir ?

Actuellement la programmation orienté objet (POO) est très réputé pour ce type de projet.

Au travers d'une bonne structure, c'est à dire l'utilisation approprié de l'héritage, classe abstraite, et polymorphisme, la POO permet de manipuler les données, groupés ou non, avec une grande facilité.

Au cours de l'article il ne sera pas rappelé les notions de base de la POO, cependant le code sera suffisement commenté pour que les débutants en objet puisse s'y rentrouver, et découvrir ainsi un cas très concret de POO.

III - Les niveaux d'abstractions

Un niveau d'abstraction est un moyen de ne considérer seulement certains facteurs. Plus le niveau d'abstraction est bas (bas niveau) et plus nous avons besoin de savoir ce qu'il se passe concretement sur la machine.

On parle souvent de langage de bas niveau, ou par opposition, des langages de haut niveau.

Prenons l'exemple du C++ qui est un langage ou gestion de la mémoire est possible (au travers de pointeurs), il n'en est pas de même pour le Java, ou on fait abstraction de la gestion de la mémoire, en effet nous ne savons pas comment elle se fait, et ca ne nous interesse pas.

Dans la programmation web, nous sommes très haut dans les niveau d'abstraction, et toute cette gestion de la mémoire est transparente pour nous.

Cette vision des niveaux d'abstraction va nous aider à avancer sur le projet. Nous allons développer petit à petit ce générateur en commencant par les couches les plus basses (dans notre cas ca sera la création des attributs, des balises) Puis ferons de plus en plus abstraction de la facon dont le code est généré pour ce concentrer sur l'utilisation du générateur.

Nous allons donc commencer par manipuler des attributs, des balises, puis rapidement nous manipulerons des élements plus concret comme des champ texte, ou bien des tableaux (au sens (X)HTML).

La facon de créer le code proprement dis deviendra donc avec le temps transparent pour nous, et c'est cela tout l'interêt de la POO.

IV - Développement de la base du système

Nous allons enfin attaquer la conception de la librairie. L'important tout au long du développement, est de fragmenter le plus possible le code, afin de pouvoir le manipuler petit bout par petit bout.

Fini les longues fonctions de 600 lignes de code, ici nous ferons du code court et simple.

La partie du modèle expliqué dans cet article est disponible **plus bas**.

IV-A - Organisation des fichiers

Toujours dans un but de pérénité du code, nous allons créer un fichier par class.

Ces fichiers suivrons le nommage suivant : nom.class.php

IV-B - Par ou commencer ?

La première question à se poser est : quelle structure de donnée vais-je adopter ?

Nous allons beaucoup utiliser la polymorphie afin de rendre transparent la génération du code. Chaques classes devras pouvoir générer son propre petit bout de code. Ainsi le conteneur apellera les méthodes générant le code de tout les éléments qu'il contient.

Plus concretement nous aurons des instances d'attribut qui génèrerons seulement leurs propre code (par exemple name="pseudonyme"), puis une instance de balise qui génèrera son propre code, et demandera aux instances d'attribut qu'elle possède de renvoyer leurs code.

Nous avons trois parties principales à traiter, les valeurs, les attributs, et les balises.

Pourquoi faire une classe à part pour stocker une valeur qui n'est autre qu'une vulgaire chaine de caractère ?

Et bien nous avons dit que chaques éléments de notre structure devra être capable de générer son propre bout de code. Il est donc pratique d'utiliser une classe valeur qui possèderas une methode générant le code html de la valeur.

Actuellement il se trouve que le code (X)HTML correspond exactement à la valeur, cependant si un jour les normes changent par exemple, il sera très aisé de faire évoluer le générateur.

IV-C - Les valeurs

Certains d'entre vous se demande peut-être pourquoi nous utilisons des listes de valeurs par attribut.

Et bien certains attributs peuvent posséder plusieurs valeurs (comme class par exemple). Il sera pratique de pouvoir ajouter des valeurs avec une simple méthode addValue() par exemple.

L'implémentation de cette classe est plutot simple

valeur.class.php

```php
<?php

class hgValeur
{
 private $strValue; //Notre valeur

 public function __construct($pStrValue) // On impose de renseigner la valeur
 {
  $this->strValue = $pStrValue;
 }

 private function getValue() { return $this->strValue; } // Permet de récupérer la valeur.

 public function makeCode() { return getValue(); } // Pour le moment identique à la valeur.
};

?>
```

IV-D - Les attributs

Les attributs possèdent un libéllé, et une liste de valeurs Voici une implémentation basique :

attribut.class.php

```php
<?php

include_once('valeur.class.php');

abstract class hgAttribut
{
 private $strLibelle; // Libéllé de la balise, par exemple "name"
 private $listValeur; // Tableau contentant toutes les valeurs de cet attribut

 function __construct($pStrLibelle, $pStrValeur = NULL) //Le libéllé est obligatoire, mais la
 valeur peut être définie plus tard.
 {
  // Initialisation des propriétés.
  $this->listValeur = array();
  $this->strLibelle = $pStrLibelle;

  // Si on saisie une valeur par défaut on a la rajoute a la liste.
  if(!is_null($pStrValeur))
  {
   array_push($this->listValeur, new hgValeur($pStrValeur));
  }
 }

 //Permet d'ajouer apres l'instanciation, de nouvelles valeurs
 public function addValue($pStrValeur)
 {
  array_push($this->listValeur, new hgValeur($pStrValeur));
 }

 //Methode générant le code de l'attribut, par exemple name="monNom"
 public function makeCode()
 {
  $intNbValeur = sizeof($this->listValeur);
  $strValues = '';
  for($i = 0; $i < $intNbValeur;)
  {
   $strValues .= $this->listValeur[$i]->makeCode();
```

attribut.class.php

```
      if(++$i < $intNbValeur)
      {
       $strValues .= ' ';
      }
     }
     return $this->strLibelle.'="'.$strValues.'"';
    }
   };

   ?>
```

Certains d'entre vous se demanderons pourquoi cette classe est abstraite.

Dans l'état actuel un attribut doit avoir son libéllé renseigné, chose indispensable mais pas pratique a l'utilisation.

En effet, si a chaque instance d'attribut il faut définir de quel attribut il s'agit, l'utilisation de la librairie deviendrais vite laborieuse.

De plus n'importe quel libéllé peut être saisie, ce qui laisse une possibilité d'erreur pour l'utilisateur. Il est donc préférable de ne pas laisser l'instanciation directement possible d'un attribut. Afin de garder un contrôle plus important, nous allons dériver cette classe pour chaques attributs que nous vondrons gérer.

Ici nous allons créer un type "name" par exemple, qui renseignera automatiquement son libéllé.

name.class.php

```
<?php

include_once('attribut.class.php');


class hgName extends hgAttribut //on hérite d'attribut
{
 // Tout comme sa mère, il n'est pas nécessaire de donner une valeur a l'instanciation
 // dans cette classe le libéllé est automatiquement attribué à l'instanciation.
 // L'utilisateur ne peut plus se tromper.
 public function __construct($pStrValeur = NULL)
 {
  parent::__construct('name', $pStrValeur); // Appel du constructeur de la classe mère.
 }
};

?>
```

IV-E - Les balises

La gestion des balises est un peut différente puisqu'il éxiste deux types de balise

- Les balises auto-fermantes, c'est à dire qu'elle s'ouvrent et se ferment en meme temps, comme la balise br.
- Les balises conteneurs, qui peuvent contenir de l'information, comme la balsie div.

Il va donc falloir faire une distinction entre les deux types. C'est à dire que même si elles possèdent des informations communes (les attributs par exemple), elles possèderons des différences.

Une balise conteneur devras par exemple contenir d'autres balises. Une baliste auto fermante n'aura pas besoin de cela.

Voici l'implémentation de la partie commune.

balise.class.php

```php
<?php

abstract class hgBalise
{
 protected $strLibelle;
 protected $listAttribut;

 public function __construct($pStrLibelle)
 {
  $this->strLibelle = $pStrLibelle; // le nom de la balise, "div" par example.
  $this->listAttribut = array(); //les attributs qu'elle possède.
 }

 // On propose la possibilité d'ajouter le nombre d'attribut que l'on désire.
 public function addAttribut($pObjAttribut)
 {
  array_push($this->listAttribut, $pObjAttribut);
 }

 // Par souci de clareté du code, une methode servant à appeler la génération du code de tout les
 attribut contenu à été faite.
 // Ainsi qu'elle soit conteneur ou auto fermante, le code concernant les attribut sera deja
 construit.
 protected function makeCodeAttribut()
 {
  $intNbAttribut = sizeof($this->listAttribut);
  if($intNbAttribut > 0) { $strAttribut .= ' '; }
  for($i = 0; $i < $intNbAttribut;)
  {
   $strAttribut .= $this->listAttribut[$i]->makeCode();
   if(++$i < $intNbAttribut)
   {
    $strAttribut .= ' ';
   }
  }
  return $strAttribut;
 }
};

?>
```

Notez que ici aussi, la classe est abstraite. Une simple balise sans préciser s'il s'agis d'une balise auto-fermante, ou conteneur, n'as aucune sens.

Il conviens donc d'interdir son instanciation.

IV-D-1 - Balise auto fermante

Maintenant que la base d'une balise est faite, il va falloir la spécialiser afin qu'elle devienne utilisable.

Une balise autofermante ne possède pas de contenu, il s'agis donc de simplement construire la balise proprement dite.

bautoclose.class.php

```php
<?php

include_once('balise.class.php');

// On hérite de balise
abstract class hgBAutoClose extends hgBalise
{

 public function __construct($pStrLibelle) //il est indispensable de connaitre son nom (div par exemple)
 {
  parent::__construct($pStrLibelle);
 }

 public function makeCode() // son code sera donc une simple concaténation du code de ses attributs avec son libéllé
 {
  return "\n\t".'<'.$this->strLibelle.$this->makeCodeAttribut().'/>';
 }
};

?>
```

Notez comme la génération du code deviens transparente, on ne se préoccupe déja plus de comment un attribut est généré, il se débrouille tout seul.

IV-D-2 - Balise conteneur

Pour une balise conteneur c'est un peu plus délicat, en plus du fait que la syntaxe du code généré soit légèrement différente, il va falloir gérer les contenus de cette balise.

Une balise devras pouvoir contenir une liste de balise.

bautoclose.class.php

```php
<?php

include_once('bnautoclose.class.php');

abstract class hgBNAutoClose extends hgBalise
{
 private $listContent; // La liste des balises se trouvant dans la balise.

 public function __construct($pStrLibelle) // Le libéllé toujours indispensable
 {
  parent::__construct($pStrLibelle);
  $this->listContent = array();
 }

 // Tout comme la génération du code des attribut, on construit une methode
 // ayant pour rôle d'appeler les méthodes de création de toutes les balises présente a l'intérieur de notre balise.
 private function makeCodeContent()
 {
  $intNbContent = sizeof($this->listContent);
  for($i = 0; $i < $intNbContent;)
  {
   $strContent .= $this->listContent[$i]->makeCode();
   if(++$i < $intNbContent)
```

bautoclose.class.php

```
      {
       $strContent .= ' ';
      }
     }
     return $strContent;
    }

    // Il est important de laisser la possibilité de rajouter du contenu apres l'instanciation.
    public function addContent($pObjContent)
    {
     array_push($this->listContent, $pObjContent);
    }

    // La génération du code est la même que pour les auto fermante, mis a part le rajout du contenu.
    public function makeCode()
    {
     return
    "\n".'<'.$this->strLibelle.$this->makeCodeAttribut().'>'.$this->makeCodeContent()."\n".'</'.$this-
    >strLibelle.'>';
     }
    }

    ?>
```

IV-E-3 - Et maintenant on en fait quoi ?

Encore une fois, sans connaitre quel balise on veux utiliser, ca n'as aucune sens d'instancier "une balise". C'est pour cela qu'une fois de plus les deux types de balises sont des classes abstraites.

Cependant maintenant, tout les balises possibles sont rapidement réalisable. I suffira d'hériter le bon type de balise, afin d'avoir à disposition toutes les fonctionnalitée d'un balise autofermante, ou bien conteneur.

Ainsi si nous voulons créer la balise div par exemple il suffira seulement de faire comme ceci :

input.class.php

```
<?php

include_once('bnautoclose.class.php');

//div peut contenir de l'information alors on hérite une balise conteneur
class hgDiv extends hgBNAutoClose
{

 public function __construct()
 {
  parent::__construct('div'); // on a juste besoin de préciser son libéllé.
 }
}

?>
```

Il est maintenant possible d'implémenter nimporte quoi assès facilement.

Voici un diagramme de classe avec quelques exemples d'implémentation supplémentaires:

V - Un niveau d'abstraction supérieur ?

IV-A - Tableau

IV-B - Les Formulaires

VI - Implémentation d'une couche évenementielle

VI - Le diagramme de classe

VIII - Conclusion

IX - Remerciements

X - HTMLgenenrator

VIII-A - Participer au développement

VIII-B - Téléchargement