# FIGDICE, UN TEMPLATING SYSTEM EFFICACE ET ORIGINAL

*par Gabriel Zerbib [Militant séparatiste des couches applicatives]*

Parmi les nombreux systèmes de gabarits pour applications web [WIKI], certains comme Smarty pour PHP ou Django pour Python sont devenus très célèbres. Mais chacun a sa particularité qui le rend unique, et le choix d'un Templating System est structurant pour l'intégralité du projet web : mieux vaut opter pour le bon avant d'entreprendre le développement. Étudions ici un système de gabarits facile offrant de nombreux atouts par son approche originale : FigDice.

## 1 Introduction

### 1.1 Rappel de concept

(Si vous êtes familier avec le principe des systèmes de Templating, vous pourrez passer directement au 1.2)

Un *Templating System* (système de gabarits dans la langue de Molière qui pourtant n'y comprendrait pas grand chose) est une brique logicielle qui permet l'isolement de la couche de présentation au sein d'un programme (principalement web).

La couche de présentation est celle qui produit le rendu visuel que reçoit l'utilisateur ayant fait appel à un service (ou une page). La forme que revêt la réponse peut dépendre de plusieurs paramètres, qui n'auront d'influence que sur la forme elle-même et non pas sur le contenu. Par exemple, une même page visitée par un navigateur de bureau ou par un terminal mobile : le rendu sera différent, les informations affichées seront adaptées tant en quantité qu'en format, pourtant les mêmes couches fonctionnelles sous-jacentes sont invoquées.

Les *Templates* sont un moyen de mettre en œuvre cette abstraction : le projet comporte d'un côté les couches fonctionnelles (qui s'appuient elles-mêmes sur les composants techniques de plus bas niveau comme l'accès à la base de données) ; et de l'autre les couches de présentation dont le rôle est de produire une enveloppe exploitable par l'acteur effectuant la demande (qu'il soit une machine attendant des données structurées, ou un humain attendant des belles pages en couleur).

### 1.2 Principaux avantages de FigDice

Dans bon nombre de Templating System, l'adhérence avec le langage sous-jacent est très forte : on y utilise des mots-clés pour contrôler le flot d'exécution, des boucles, des fonctions du langage de programmation, des variables directement partagées avec l'application. Pire, certains moteurs permettent l'accès direct à la base de données, ou l'exécution de code arbitraire ayant le même pouvoir que l'applicatif lui-même !

Pourtant, les rôles sont radicalement disjoints. Un très bon développeur PHP peut se révéler un médiocre designer HTML ; et un excellent artiste (n'ayons pas peur des mots) maniera avec brio le HTML et les CSS, mais risquera de provoquer des accidents graves s'il utilise des instructions PHP dans ses documents, etc.

C'est dans ce contexte que peuvent s'apprécier les atouts de FigDice. Ce Templating System offre, entre autres avantages, des gabarits qui :

- sont en **XML** : la validité syntaxique est assurée par la plupart des éditeurs ;
- sont du **HTML** avec des attributs étendus : ils peuvent s'afficher directement dans un navigateur presque sans dégradation visuelle ;
- utilisent les données au moyen d'**expressions simples** et robustes, indépendantes du langage de programmation ;
- prennent en charge nativement l'**internationalisation** au moyen de fichiers de ressources (en XML aussi) ;

- gèrent les **inclusions**, les **boucles**, les **conditions** par une syntaxe non intrusive dans le document ;
- peuvent se concevoir avec des éditeurs WYSIWYG **sans programmation** ;
- présentent une **étanchéité** totale avec les « mondes inférieurs » des autres couches applicatives ;
- offrent l'**inversion de contrôle** pour l'apport des données, que les vues peuvent tirer à elles sans que ce soit nécessairement les contrôleurs qui les y poussent.

Nous allons étudier quelques aspects qui, je l'espère, vous donneront envie de faire vos premiers pas avec FigDice. Toute la documentation est disponible (en anglais) sur http://www.figdice.org/en/manual.html.

## 1.3 Installation et mise en œuvre

Dans sa version actuelle (1.3.3 lors de la rédaction de cet article), FigDice se présente sous la forme d'une bibliothèque de classes écrites en PHP. L'archive est téléchargeable sur le site du projet : http://www.figdice.org/en/download.html.

Les sources contiennent un grand nombre de fichiers, mais il suffit d'ajouter le dossier qui les contient dans le `include_path` de votre programme PHP :

```
set_include_path( get_include_path() . PATH_SEPARATOR . $dossier_FigDice );
```

Le mode opératoire est relativement simple : on instancie un objet `FIG_View`, on lui indique le nom de fichier du template à exécuter, et... on est presque prêt pour lancer le rendu et en récupérer le résultat.

```
//Le dossier figdice se trouve au premier niveau sous votre include_path
require_once 'figdice/FIG_View.php';

//Créer un objet FIG_View
$figView = new FIG_View();

//Charger un fichier XML de Template
$figView->loadFile('mes_templates/hello_world.xml');

//Afficher le résultat (ou le capturer dans une variable)
echo( $figView->render() );
```

Le code ci-dessus est opérationnel, mais il ne présente encore que peu d'intérêt, car le template ainsi exécuté ne reçoit aucune information de l'extérieur. Voyons dans le chapitre suivant comment apporter des données à un template FigDice.

# 2 Fournisseurs de données : l'inversion de contrôle

Une des principales particularités de FigDice, et qui lui confère toute son originalité, est la technique par laquelle les données sont apportées au template.

Prenons un exemple, soit l'affichage HTML suivant (extrait) :

```
<body>
  <div> Bonjour <b>Gabriel</b>, bienvenu dans FigDice </div>
</body>
```

Schématiquement, quel que soit le système choisi, on « templatiserait » cette sortie de la façon suivante (nous aborderons plus loin la syntaxe FigDice) :

```
<body>
  <div> Bonjour <b>(firstname)</b>, bienvenu dans FigDice </div>
</body>
```

où `(firstname)` représente l'utilisation d'une valeur qu'il s'agirait de faire passer à notre template lors de son exécution.

Habituellement, dans la plupart des Templating Systems, le programme « appelant » (conceptuellement, il s'agit généralement du Contrôleur) injecte des données dans le template pour que ce dernier puisse s'en servir. C'est également possible dans FigDice avec la syntaxe suivante :

```
//Une fois la vue instanciée, on peut lui injecter des valeurs
$figView->mount( 'firstname', 'Gabriel' );
```

Pourtant, si cette approche reste disponible, ce n'est pas celle que FigDice privilégie. Au contraire, c'est ici la vue elle-même (le fichier de template) qui a la capacité (la responsabilité, même !) de tirer à elle les données dont elle a besoin, au moyen d'un objet appelé **Feed**.

Le Feed est l'articulation entre le template et le monde qui lui est extérieur. Chaque classe dérivée de la classe abstraite `FIG_Feed` prend en charge la collecte d'un ensemble cohérent de données structurées (par exemple, les données personnelles relatives à un compte utilisateur), et les retourne à la vue appelante, sur qui repose la responsabilité de les exploiter comme bon lui semble.

Reprenons notre exemple (idem, la syntaxe sera détaillée plus loin) :

```
<body>
  <fig:feed class="UserInfoFeed" target="userInfo" />
  <div> Bonjour <b fig:text="/userInfo/firstname"> </b>, bienvenu dans FigDice </div>
</body>
```

Plusieurs éléments doivent être expliqués ici.

Tout d'abord, nous voyons la balise `<fig:feed>` qui invoque une classe arbitraire précisée en attribut, et récupère le résultat dans une « variable » nommée `userInfo`. Cette balise, comme toutes celles du *namespace* `fig:`, ne produit pas d'affichage, elle est une directive d'exécution du template et est absorbée au moment du rendu (au même titre que l'instruction d'inclusion par exemple : voir en 3.3).

Ensuite, l'attribut `fig:text` de la balise `<b>`, qui mentionne la donnée à afficher en guise de contenu.

Étudions de plus près.

## 2.1 Aperçu de l'exploitation

On peut imaginer que ce `UserInfoFeed`, qui dérive de la classe abstraite `FIG_Feed`, charge depuis la base de données un ensemble d'ojets tels que : l'identité de l'utilisateur, une liste de centres d'intérêt, etc. Ces données peuvent prendre une forme arbitraire (tableaux, hiérarchie d'objets, scalaires) et nous aborderons succinctement au paragraphe 2.2 la syntaxe FigDice pour les parcourir. Dès lors, ces données deviennent disponibles au template (et à tous ceux qu'il inclura) sous le nom spécifié par l'attribut `target` : ici `userInfo`.

Avant d'approfondir le fonctionnement de la classe `FIG_Feed`, admettons momentanément que l'appel à notre fournisseur `UserInfoFeed` par la balise `<fig:feed>` provoque le montage en mémoire dans le Template, sous le nom `userInfo`, d'un dictionnaire comme celui-ci :

```
array(
  'firstname' => 'Gabriel',
  'country' => 'France',
  'hobbies' => array ( 'GNU/Linux', 'PHP' )
)
```

Penchons-nous maintenant sur l'exploitation des valeurs dans le template.

Comme introduit plus haut, l'attribut `fig:text` appliqué à toute balise HTML classique a pour effet d'en remplacer le contenu par le résultat de l'expression fournie. Dans notre exemple, nous avons :

```
<b fig:text="/userInfo/firstname"> ce contenu sera remplacé </b>
```

où `/userInfo` fait référence à notre dictionnaire répondu par le fournisseur `UserInfoFeed`, et `/userInfo/firstname` accède à la clé `firstname` de ce tableau associatif.

Dans FigDice, cette expression s'appelle un **Path**.

Nous disposons de la même manière des *paths* suivants :

- `/userInfo/country`
- `/userInfo/hobbies/0`
- `/userInfo/hobbies/1`

où l'on entrevoit notamment comment accéder aux éléments d'un tableau par leur index, qui est une clé comme une autre.

**Remarque**

Dans les expressions FigDice, tous les accès aux propriétés et éléments indexés sont sécurisés : l'appel à une propriété qui n'existe pas, ou à un index dépassant les bornes d'un tableau, retourne simplement la chaîne vide. Pas de risque de faire échouer un template !

Avant d'aller plus loin dans la présentation des **Path** et de l'analyseur syntaxique des expressions FigDice, attardons-nous un peu plus sur ce module d'alimentation en données que constitue le **Feed**.

## 2.2 Anatomie du fournisseur de données

Le Feed implémente la méthode abstraite `run()`, invoquée par le moteur. Elle retourne le dictionnaire de données (tableau associatif, ou objet du genre « bean ») qu'il monte dans la vue.

Chaque balise `fig:feed` donne lieu à l'instanciation d'un nouvel objet, même si la même classe a déjà été invoquée dans le template.

On peut passer directement depuis la vue des paramètres à l'instance Feed au moyen d'attributs dans la balise `fig:feed` :

```
<fig:feed class="MonFeed" target="pointDeMontage"
   monParam="expr1" unAutre="exprN" />
```

Tout attribut de la balise `fig:feed`, en dehors de `class` et `target`, est considéré comme un paramètre pour l'objet Feed. Leurs valeurs sont des **expressions** FigDice (voir section 3.2) que le moteur évalue avant d'invoquer la méthode `run()` du Feed.

L'objet récupère ses paramètres en appelant sa méthode `getParameter($paramName)`. Les valeurs peuvent être de simples scalaires, aussi bien que des arbres obtenus par exemple en réponses à d'autres Feeds.

## 2.3 Factory : le lien avec le contrôleur

Bien souvent, les réponses à retourner proviennent d'une base de données, en y apposant des filtres issus de valeurs qui ne sont bien sûr pas codées en dur. Pour ce faire, le `Feed` a besoin de connaître les détails de connexion, une référence à la session HTTP en cours, etc.

En outre, avant même de parler d'instance de Feed, et puisque la directive `fig:feed` informe le moteur sur le nom de la classe à instancier, il faut bien que notre PHP sache où la trouver !

C'est là le rôle de la **Factory** : ce *design pattern* bien connu permet de passer au moteur un objet qu'il connaît bien (par son interface, à qui il demandera d'instancier des Feed), mais dont nous maîtrisons totalement le code, de sorte que notre code conserve la responsabilité d'effectuer en fin de compte l'instanciation. La méthode-clé de l'interface `FIG_FeedFactory` est :

```
/**
 * @return FIG_Feed, null if the specified FIG_Feed
 * subclass is not handled by this factory
 */
public function create ( string $className, array $attributes )
```

Le cycle peut se décrire ainsi :

- Vous écrivez une classe `Factory` responsable de l'instanciation d'un ensemble de Feed.

- Vous instanciez un objet de cette classe `Factory`, que vous pouvez paramétrer selon vos besoins (lien à base de données, référence au contrôleur, etc.).
- Vous créez une vue `FIG_View`, vous chargez son template.
- Vous enregistrez votre Factory auprès de la vue, et vous lancez le rendu.
- Par suite, lorsque le moteur FigDice doit traiter une balise `fig:feed`, il s'en remet à votre instance de Factory : si celle-ci est en mesure d'instancier la classe demandée, elle en retourne une instance, sinon le moteur FigDice poursuit avec d'éventuelles autres Factories, ou une erreur.

L'inversion de contrôle par laquelle la vue attire à elle les données qui l'intéressent montre toute sa puissance lorsque l'on cherche à écrire des contrôleurs génériques, et que les vues sont des assemblages de petits modules utilisant des briques de présentation réutilisables à travers les gabarits, dédiées à l'agrégation de données d'une zone applicative réduite et bien définie.

# 3 Mise en œuvre

Nous aborderons maintenant le formalisme de mise en œuvre des templates FigDice, c'est-à-dire de quoi sont faits les templates et de quels outils dispose le concepteur.

## 3.1 Notation XML

Tout d'abord, rappelons qu'un template FigDice est un document XML dont la vocation est d'annoter un document HTML (ou un extrait de HTML) avec des directives à l'intention du moteur de rendu. Ces directives restent, par définition, compatibles avec la nature XML du document.

Ainsi, le principe fondamental du formalisme FigDice est que les directives du template (les conditions, les répétitions, etc.) s'expriment sous la forme d'attributs sur des balises, et s'appliquent donc à des nœuds et tous leurs descendants. Par exemple, une condition FigDice est en fait un attribut sur un tag, qui indique dans quelle condition le tag tout entier (et tout ce qu'il contient) doit être présent dans le document sortant, ou au contraire supprimé.

Il en résulte un certain nombre de contraintes liées à cette nature XML rigoureuse.

En particulier, même si en bout de chaîne les navigateurs sont permissifs quant à la syntaxe HTML, le template FigDice est contraint, lui, de correctement fermer chaque balise dans le bon ordre, de ne pas laisser de balise orpheline (sans sa fermeture), de ne pas dupliquer un même nom d'attribut sur une balise, etc.

En outre, les `<!-- commentaires XML -->` que vous placerez dans vos templates sont considérés comme tels par le moteur de rendu, en sorte que vous ne les retrouverez pas dans votre HTML résultant !

Mais, avec un minimum de discipline, il en résulte des documents d'autant plus lisibles, avec le bénéfice de la validation de structure XML que beaucoup d'éditeurs intègrent.

Du reste, les messages d'erreur levés par le moteur lors du rendu d'un template invalide sont explicites et permettent de retrouver rapidement le nœud fautif.

## 3.2 Expressions et univers

Le template est composé majoritairement d'éléments qui se retrouveront tels quels dans le flux résultant, à l'exception des directives FigDice dans lesquelles on importe des données (les Feeds vus plus haut) ou l'on évalue ces données au moyen d'**expressions**.

Tout ce qui est importé par les Feeds se retrouve ancré (« monté », selon le vocabulaire du projet) sous un certain nom à la racine de l'ensemble des données disponibles au template, à l'instar du *filesystem* Linux. Cet ensemble de données s'appelle l'**univers**, et le template peut ainsi parcourir les différents dictionnaires (ou assimilés) qu'il monte, par l'utilisation des **path** comme on parcourrait les répertoires. Nous verrons plus loin, à propos des boucles, comment l'univers s'organise en contextes absolus et relatifs.

Mais FigDice dispose également d'un puissant moteur d'expressions, et celles-ci peuvent comporter calculs arithmétiques, concaténations de chaînes, appels à des fonctions, opérateurs logiques et de comparaison : tout ce à quoi on peut être habitué dans les langages de programmation.

Les expressions se placent donc sur toutes sortes de nœuds XML, comme valeur de différents attributs du namespace **fig:** dont nous donnerons quelques exemples dans la suite.

Voici pour le moment quelques utilisations typiques des expressions FigDice (elles sont ici placées entre guillemets car c'est là le sort des attributs XML) :

```
<!-- concaténation de trois chaînes
  les chaînes littérales se passent entre apostrophes
  et le symbole de concaténation est le même que
  celui de l'addition -- >
"/userInfo/firstname + '    ' + /userInfo/lastname"

<!-- arithmétique élémentaire pour l'âge du visiteur -- >
" 2013 - /userInfo/birthdate/year "

<!-- beaucoup de bruit pour rien -- >
" (true and false) or (3 * ' ') "

<!-- appels de fonctions sur des collections
  ici on somme les propriétés 'prix' de tous
  les éléments du tableau articles -- >
" sum( /panier/articles/prix ) "
```

**Remarque**

- À l'instar des langages de programmation, les espaces ne sont pas significatifs (hormis à l'intérieur des chaînes littérales). Il est même recommandé d'aérer son code en faisant bon usage des espaces pour la lisibilité.
- Inspiré par son langage sous-jacent d'origine (notre cher PHP), FigDice ne discerne pas de types de données. La chaîne vide est homogène au nombre zéro et au booléen false dans les conditions, et l'on peut indifféremment concaténer des nombres et des chaînes : le moteur effectuera ce qui a le plus de sens, selon la nature des opérandes.

## 3.3 Premiers pas dans la figueraie

### 3.3.1 Conditions

À la lumière de cette petite présentation des expressions, voyons à quoi peut ressembler une condition dans un template FigDice.

Rappelons tout d'abord qu'une condition n'est autre qu'un attribut sur une balise. La valeur de l'attribut est en fait l'expression dont l'évaluation donnera **true** ou **false**.

```
<div>
  <span fig:cond="/une/valeur or /une/autre ">
        ceci est soumis à condition
    </span>
  ceci sera affiché toujours
</div>
```

### 3.3.2 Silence

Nous voici pourtant embarrassés : comment faire si le texte soumis à condition ne doit pas se trouver encerclé dans une balise (comme le **** ci-dessus) ?

Il suffit d'utiliser l'attribut **fig:mute** qui, comme son nom l'indique, rend la balise muette (mais pas son contenu !) :

```
<span fig:cond="/une/expression" fig:mute="true">
    ceci est soumis à condition,
    mais l'enveloppe span n'apparaîtra jamais, même
    en cas de condition satisfaite.
</span>
```

Dans cet exemple, l'enveloppe du nœud est tue de façon systématique par la valeur **true** de l'attribut **fig:mute**. Mais cet attribut, tout comme la **fig:cond**, accepte une expression d'une complexité quelconque, dont l'évaluation doit donner finalement un booléen.

### 3.3.3 Attributs

Retenons donc que les attributs du namespace **fig:** ne se retrouvent pas dans le document résultant mais servent à guider le comportement du moteur de rendu. Nous avons vu que l'on peut rendre conditionnel l'affichage d'une balise au moyen de l'attribut **fig:cond**, mais qu'en est-il des attributs HTML ? FigDice permet de les rendre eux aussi conditionnels, ou porteurs de valeurs complexes obtenues par l'évaluation d'expressions, par l'utilisation de la base **fig:attr** :

```
<div id="maDiv">
  <fig:attr name="class" value="/ma/classe" />
</div>
```

La balise **fig:attr** sert à ajouter (ou remplacer) à la balise parente un attribut dont le nom est passé en **name**, et dont la valeur est fournie par l'expression contenue dans l'attribut **value**. Si la chaîne à passer en attribut est plus complexe que ce qu'il ne vous est possible de faire avec l'attribut **value**, vous pouvez donner sa valeur en tant que corps du tag **fig:attr**.

À présent, l'avantage principal de cette balise qui ajoute des attributs dynamiques est que l'on peut également y apposer des conditions. Par exemple :

```
<div id="maDiv">
  <fig:attr fig:cond="/une/condition" name="class" value="/ma/classe"
/>
</div>
```

Si **/une/condition** est remplie, alors la balise ** ** portera l'attribut **class**, autrement cet attribut sera omis.

Rappel : **/une/condition** fait ici référence à un tableau associatif montée sur le nom « une » et dont l'une des clés est « condition », dont la valeur associée est assimilable à **true** (donc : un nombre différent de zéro, une chaîne non vide et ne contenant pas que des espaces, et bien sûr un booléen **true**).

Pour finir, et afin de ne pas alourdir plus que nécessaire la syntaxe déjà verbeuse d'un document XHTML, sachez qu'il est possible de spécifier des attributs par la notation dite ad hoc, à l'intérieur même de la valeur d'un attribut, en plaçant une expression FigDice entre accolades `{ }` comme ceci :

```
<div id="maDiv_{/profile/id}">
  L'attribut id deviendra par
  exemple : "maDiv_1234"
</div>
```

**Remarque** **Tableaux et objets**

Notons une particularité confortable de l'interpréteur d'expressions FigDice : les données qui composent l'univers (celles retournées par les Feeds), ne se limitent pas aux tableaux associatifs. Il peut également s'agir d'objets, dont les propriétés offrent des méthodes d'accès en lecture commençant par « get ». Ainsi, un Feed peut retourner un objet présentant par exemple la méthode publique `getName()`, et dont la valeur de la propriété sous-jacente s'obtient dans une expression par le Path « monObjet/name ».

## 3.4 Contrôle d'exécution

Par souci de concision, nous n'entrerons pas dans les détails de toutes les fonctionnalités de FigDice (pour cela le manuel est assez complet). Nous nous contenterons ici d'énumérer quelques points qui suffisent à prendre la mesure des possibilités.

### 3.4.1 Inclusions

Au sein d'un template, l'inclusion d'un autre fichier s'effectue par la balise :

```
<fig:include file="filename" />
```

Une inclusion ne doit pas briser la validité de la structure XML : il n'est pas possible d'ouvrir une balise dans un fichier parent, et de la fermer dans un fichier inclus.

L'inclusion fonctionne de façon similaire au pre-processeur C : pour le template parent, tout se passe comme si vous aviez copié/collé le contenu du template inclus à l'endroit de la balise `fig:include`.

Le chemin fourni est obligatoirement relatif au template appelant.

### 3.4.2 Boucles

L'une des fonctionnalités les plus efficaces des FigDice, est sans aucun doute la gestion des itérations. Étant donné un tableau indexé se trouvant dans l'univers, la répétition d'un sous-arbre XML sur chaque élément du tableau s'effectue de la manière suivante :

```
<balise-a-repeter fig:walk="/mon/tableau" >
  <!-- code reproduit pour chaque item ->
</balise-a-repeter>
```

où `<balise-a-repeter>` est bien évidemment une balise HTML arbitraire. En particulier, on l'utilisera surtout sur `<tr>` ou `<li>` ou les autres balises de ce type.

Ainsi, on n'écrit qu'un spécimen de la partie à cloner, en indiquant par l'attribut `fig:walk` le nom du tableau indexé. Par suite, à l'intérieur de l'itération, le **contexte** d'évaluation des expressions devient local : supposons que l'univers contienne les données suivantes (j'utilise ici la notation JSON pour sa concision) :

```
{'mon' : {
    'tableau' : [
      {'valeur' : 12},
      {'valeur' : 26},
      {'valeur' : 53}
    ]
  }
}
```

alors on pourra afficher trois `<li>` contenant chacun une des valeurs, par l'itération suivante :

```
<li fig:walk="/mon/tableau" fig:text="valeur" />
```

On voit ici que le `fig:text` est simplement « valeur », sans indication de nom du point de montage, ou d'index de l'élément courant dans la boucle : c'est parce que dans toute l'exécution du sous-arbre XML constituant le spécimen, l'**univers** se résume à l'élément en cours lui-même, et l'on accède à ses valeurs par un **Path** « relatif ».

Notons au passage ici que le sous-arbre peut être d'une complexité quelconque, tout autant que se limiter à un simple tag unique. La notation relative des Paths reste valable partout à l'intérieur du sous-arbre.

Notons encore, pour préparer la section 3.5 Fonctions, qu'en dehors de toute itération, et étant donné l'exemple de données ci-dessus, l'expression suivante retournera 91 :

```
< … fig:text="sum( /mon/tableau/valeur )" ... />
```

Pourtant, `/mon/tableau` ne comporte pas directement de propriété nommée « valeur », mais contient en fait une liste d'objets ayant chacun une propriété « valeur » : c'est que la fonction `sum` est une fonction d'agrégation, et à ce titre elle sait collecter ladite propriété « valeur » de chaque élément de `/mon/tableau`.

Mais revenons aux instructions de contrôle d'exécution.

Sachez qu'il est possible d'imbriquer des itérations, et dans ce cas le contexte intérieur permet tout de même d'accéder aux données du contexte extérieur, par la notation « ../ » en début de Path, comme on le ferait pour des répertoires.

Sachez enfin que, comme pour les répertoires, l'univers entier reste accessible depuis un contexte d'itération, par la notation « absolue » en début de Path, avec le *slash* initial.

### 3.4.3 Macros

Il est possible de nommer un sous-arbre XML se trouvant à un endroit quelconque du document, sans le rendre tout de suite, et de l'exploiter ultérieurement en un nombre arbitraire d'exemplaires.

Il s'agit du mécanisme de Macro. Une macro ne produit pas de rendu au moment de sa définition. En outre, elle accepte des paramètres que le tag appelant peut lui passer, et qui s'utilisent à l'intérieur de la macro par des Paths relatifs :

```
<!-- Definition de la Macro -->
<div fig:macro="maMacro">
  <span fig:text="unParam" />
</div>

<!-- Appel Macro ->
<div fig:call="maMacro" unParam="12" />
```

**Remarque** **À retenir**

Les paramètres passés lors de l'appel à une Macro sont des expressions FigDice. Si vous souhaitez passer une chaîne littérale, pensez à l'envelopper de *single-quotes*.

### 3.4.4 Et plus...

Nous en resterons là pour les premières fonctionnalités, essentielles à la rédaction de templates efficaces et faciles à organiser et à relire. Retenez toutefois que FigDice offre d'autres outils qui méritent quelques minutes d'attention. Citons notamment :

- un système d'emplacements réservés pour pousser du contenu au moment opportun (le contraire des Macros, finalement), qui permet de réaliser ce que d'autres Templating Systems fournissent par des jeux d'héritage ;
- une couche de *logging* des erreurs, branchable (encore par l'intermédiaire de Factories) à votre système de logging préféré, quel qu'il soit ;
- des expressions dont les noms de propriétés sont elles-mêmes le résultat de calculs (l'équivalent d'un `$obj->$prop` en PHP, mais avec toute la garantie de robustesse FigDice), très pratique pour réaliser des tableaux croisés dynamiques par exemple.

## 3.5 Fonctions personnalisées

FigDice est livré avec un certain nombre de fonctions prédéfinies, telles que `count()` qui permet de compter les éléments d'une collection, `position()` pour connaître l'indice de l'élément courant au sein d'une boucle, ou encore `GET()` pour accéder aux paramètres passés en URL.

Néanmoins, le développeur a la possibilité d'enrichir à l'infini la palette de fonctions mise à disposition des concepteurs de gabarits, au moyen là encore du design pattern de la Factory. Chaque fonction personnalisée se présente sous la forme d'une classe qui implante l'interface `FIG_Function` dont la principale méthode est :

```
//Méthode principale de l'interface FIG_Function
evaluate($arity, $arguments);
```

Il suffira alors de fournir une usine de fabrication pour ces instances de fonctions, à savoir une classe qui implante l'interface `FIG_FunctionFactory` et dont la méthode principale est :

```
//Méthode principale de l'interface FIG_FunctionFactory
create($functionName);
```

Son rôle est, étant donné un nom de fonction invoquée dans une expression FigDice, d'instancier l'objet `FIG_Function` personnalisé correspondant, si toutefois cette usine la prend en charge. Une même Factory peut prendre en charge l'instanciation de plusieurs fonctions personnalisées. Elle répond `null` si on l'invoque avec un nom de fonction qu'elle ne prend pas en charge.

Enfin, on inscrit la Factory dans la vue en appelant :

```
$figView->registerFunctionFactory( new MyFunctionFactory() );
```

Il est bien sûr possible d'inscrire autant de Factories que nécessaire.

# 4 Évolution du projet FigDice

Vous avez pu à présent découvrir quelques-unes des nombreuses possibilités de ce moteur de gabarits. Beaucoup d'autres fonctions sont à découvrir sur le site du projet. Sa grande flexibilité, combinée à la robustesse de sa syntaxe, en font un système particulièrement agréable à manier, et par lequel se réconcilient la subtilité architecturale des applications modernes et les exigences des concepteurs d'interfaces visuelles.

Bien que ne bénéficiant pas encore d'une grande communauté de développeurs, le projet FigDice est déjà assez mûr. Il tourne en production sur de nombreux sites et applications, avec des performances honorables.

Pourtant des évolutions importantes sont souhaitées, telles que des fonctionnalités liées à la compilation des Templates, l'optimisation du cache, ou encore la préparation d'un *package* PEAR, le portage du moteur dans d'autres langages de programmation, un *plug-in* Eclipse de validation de syntaxe, etc.

Aussi, toutes les bonnes volontés sont les bienvenues, et si après avoir examiné ce programme, vous souhaitez apporter votre contribution dans un domaine ou un autre, n'hésitez pas à rejoindre le projet sur www.figdice.org. ■

**Références**

[WIKI] http://en.wikipedia.org/wiki/Web_template_system

. Le logo du projet est tiré de l'œuvre libre intitulée « Fig Fruit » par Gnokii : http://openclipart.org/detail/131023/fig-fruit-by-gnokii