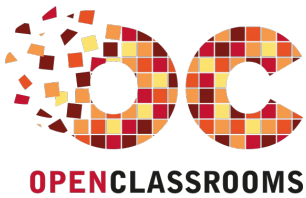
Comment parser facilement du XML

Par Crabe05



www.openclassrooms.com

Sommaire

Comment parser facilement du XML

Par  Crabe05

Mise à jour : 01/01/1970
Difficulté : Intermédiaire

Bonjour à tous.
Vous venez d'apprendre le XML, et vous ne savez pas à quoi ça va bien pouvoir vous servir ? Ou vous voulez une méthode simple pour lire un fichier XML comme si il s'agissait d'un fichier XHTML ?
Eh bien vous êtes au bon endroit !
Nous allons apprendre à « transformer » du XML très facilement.
En avant !
Sommaire du tutoriel :

Définitions

Dans cette rubrique, nous allons aborder la théorie, et LA technique utilisée pour parser simplement du XML.

Le XML

Je ne vais pas parler du XML, considérant que vous avez déjà lu et compris ce tuto et celui-ci (au moins la première partie).

La technique

Nous allons utiliser un fichier pour parser (= transformer en un autre fichier) notre fichier XML. Oui, c'est vrai, ça peut paraître un peu bizarre de transformer un fichier en un autre fichier par un fichier...

Ce fichier sera écrit dans un langage que vous devriez bien connaître, puisqu'il s'agit du... XML. En effet, ce langage nommé « *stylesheet* » (« feuille de style » en anglais), ou XSLT (*eXtensible Stylesheet Language Transformation*) est totalement basé sur le XML.

Le XSLT est un langage définit par le W3C et qui sert tout spécialement à la transformation de fichiers XML en... d'autres fichiers XML... Or, comme certains langages que vous devez connaître (comme le xHTML par exemple) sont eux aussi basés sur le XML...

Pour « jumeler » la feuille de style avec un document XML, on place le code suivant juste après la toute première ligne XML :

Code : XML

```
<?xml version="1.0" encoding="ISO-8859-1" ?><!--Première ligne XML--
>
<?xml-stylesheet type='text/xsl' href='nomStylesheet.xsl'?><!--Ligne
```

```
de jumelage-->
<!--Code-->
```

Ici, le fichier XML sera couplé avec le fichier « nomStylesheet.xsl ».

Les *stylesheets* ont leur extension en .xsl, pensez-y lorsque vous enregistrerez votre fichier !

Quelques informations supplémentaires

Vous serez plus loin appelés à manier les signes « < » et « > ». Or, ceux-ci sont déjà utilisés pour les balises. Veillez donc à les remplacer par « < » pour « < » (*Lower Than* ; plus petit que), et « > » pour « > » (*Greater Than* ; plus grand que).
Ainsi, un « <= » se transforme en « <= ». Pensez-y : c'est la source de beaucoup d'erreurs !

Pour ceux qui sont déçus

Oui, je n'utilise pas le PHP. Mais il y a un tuto écrit justement pour ça.

Les bases

Comme tout langage, il y a des bases. Autant vous dire qu'elles ne sont pas très difficiles. Sachez que nous utiliseront un *namespace* nommé « xsl ».

Le code de base

Il y a un code de base sans lequel le fichier ne servirait strictement à rien :

Code : XML

```
<?xml version="1.0" encoding="ISO-8859-1"?><!--Vous devez connaître
:)-->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <!--Vous vous en doutez : on mettra le code ici-->
</xsl:stylesheet>
```

Analysons tout ça...
D'abord, la balise XML que vous devez connaître.
Ensuite, une balise XML qui permet de dire qu'il s'agit de XSLT (avec sa version) et de faire un lien vers le site W3C par lequel passera notre document pour être transformé.

Allons un peu plus loin : en effet, si vous compilez même avec une *stylesheet* correcte un fichier XML... ça ne marchera pas. Pourquoi ? Parce qu'on n'a pas défini les sorties !

Code : XML

```
<xsl:output
  method="html"
  encoding="ISO-8859-1"
  doctype-public="-//W3C//DTD XHTML//EN"
  doctype-system="http://www.w3.org/TR/2001/REC-xhtml11-20010531"
  indent="yes" />
```

Je ne vais bien sûr pas détailler tous les paramètres. Sachez simplement que « *method* » est l'attribut de la méthode de transformation. Ici, c'est « html », mais cette valeur peut vouloir dire aussi « XHTML ». Vous pouvez tout mettre tant que vous connaissez le *doctype* correspondant (le lien de l'attribut 'doctype-system'). « *encoding* », c'est le type de caractère. Enfin, « *indent* » dit si les indentations du XHTML doivent être respectées (pour les paragraphes, les listes, etc.).

Les balises en elles-mêmes

Bon, nous y voilà...
Commençons par ce que je juge être l'essentiel :

Code : XML

```
<xsl:template match="nom_balise">
    <!--Ici on mettra le code qui remplacera la balise et son
contenu-->
</xsl:template>
```

Cette ci s'activera à chaque fois que, dans la lecture du fichier XML, le fichier rencontrera la balise <nom_balise></nom_balise>ou <nom_balise />. À ce moment, le fichier va écrire dans le fichier de sortir le code qu'il y a entre les balises (en l'interprétant bien-sûr).

Mais si je veux récupérer le contenu des attributs et balises ?

Petits impatients : La suite arrive !
C'est vrai qu'avec ce code, le *parser* ne se pose pas de question : il remplace, un point c'est tout. Cependant, nous pouvons faire en sorte d'écrire le texte des attributs et du contenu de la balise :

Code : XML

```
<xsl:value-of select="truc" />
<!--Ne vous amusez pas à écrire ça si vous n'avez pas lu la suite
!!!-->
```

Là il y a un truc (hahem). Entre les guillemets, il faut mettre une requête XPATH. Nous verrons ça en détail plus loin, en attendant voici quelques requêtes simples :

- `select="."` : récupère le contenu de la balise en cours de *parsage*
- `select="@attribut"` : récupère la valeur de l'attribut nommé « attribut ».

Enfin, il nous reste une dernière balise à aborder dans cette partie :

Code : XML

```
<xsl:apply-templates select="nom_balise2" />
```

Cette balise permet d'appeler le *parsage* d'une autre balise, la balise <nom_balise2></nom_balise2> (ou <nom_balise2 />), tant que cette balise est fille de la balise en cours de *parsage*. Sachez que vous devez mettre là aussi une requête XPATH. Pour appeler le *parsage* de toutes les balises filles, mettez simplement `select="*"`.

En fait, à chaque fois que dans une balise XSLT vous voyez l'attribut « <italique>select</itlaique> », c'est que vous devez mettre une requête XPATH.

Bon, un petit exemple ?

Le XML à *parser* :

Code : XML

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet type='text/xsl' href='parser.xsl'?>
<A>
    <B>aaa</B>
    <C>
        <D att="1"/>
        <D att="2"/>
    </C>
    <B>bbb</B>
</A>
```

et sa *stylesheet* :

Code : XML

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
  method="html"
  encoding="ISO-8859-1"
  doctype-public="-//W3C//DTD XHTML//EN"
  doctype-system="http://www.w3.org/TR/2001/REC-xhtml11-20010531"
  indent="yes" />

<xsl:template match="A">
    <html><body>
     <xsl:apply-templates select="*" />
    </body></html>
</xsl:template>

<xsl:template match="B">
 <xsl:value-of select="." /><br />
</xsl:template>

<xsl:template match="C">
 <xsl:apply-templates select="D" />
</xsl:template>

<xsl:template match="D">
 <xsl:value-of select="@att" /><br />
</xsl:template>

</xsl:stylesheet>
```

Ici, la *stylesheet* est nommée « parser.xsl ».
Vous remarquerez que j'ai mis des balises XHTML dans la *stylesheet*. On a tout à fait le droit ! C'est quand même notre premier but de transformer du XML en XHTML...

J'espère que vous avez tout reconnu !

Pour lancer un document XML comme s'il s'agissait d'une page XHTML, il suffit de double-cliquer sur l'icone de votre document XML. Si le lien est bien fait, vous n'aurez aucun problème, sauf peut-être des erreurs dans votre code, qui seront alors indiquées à la place du document.

Bon, un petit exercice... 

Voilà mon code :

Code : XML

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!--Ici, la ligne de jumelage. Je vous laisse la faire-->
<tirelire nom="Ma tirelire">
    <billet5 nombre="10">Allemandes</billet5>
    <billet10 nombre="5">Autrichiennes</billet10>
    <piece2 nombre="12">Françaises</piece2>
    <piece2 nombre="4">Espagnoles</piece2>
    <cent50 nombre="6">Italiennes</cent50>
</tirelire>
```

Je veux que vous obteniez cela :

Citation : Ce que vous devez obtenir (ne tenez pas compte de la police, elle est automatique)

Tirelire : Ma tirelire
Billets de 5 euros : nombre = 10, nationalité = Allemandes
Billets de 10 euros : nombre = 5, nationalité = Autrichiennes
Pièces de 2 euros : nombre = 12, nationalité = Françaises
Pièces de 2 euros : nombre = 4, nationalité = Espagnoles
Pièces de 50 cents : nombre = 6, nationalité = Italiennes

Et pas de tricherie ! Je veux un code qui *parse* quand même si on change le XML !!! 

Au travail !

...

...

C'est bon ? Voilà la correction :

Secret (cliquez pour afficher)

Code : XML

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
  method="html"
  encoding="ISO-8859-1"
  doctype-public="-//W3C//DTD XHTML//EN"
  doctype-system="http://www.w3.org/TR/2001/REC-xhtml11-20010531"
  indent="yes" />

<xsl:template match="tirelire">
 <html><body>Tirelire : <xsl:value-of select="@nom" /><br />
```

```
 <xsl:apply-templates select="*" /><br />
 </body></html>
</xsl:template>

<xsl:template match="billet10">
 Billets de 10 euros : nombre = <xsl:value-of select="@nombre" />,
nationalité = <xsl:value-of select="." /><br />
</xsl:template>

<xsl:template match="billet5">
 Billets de 5 euros : nombre = <xsl:value-of select="@nombre" />,
nationalité = <xsl:value-of select="." /><br />
</xsl:template>

<xsl:template match="piece2">
 Pièces de 2 euros : nombre = <xsl:value-of select="@nombre" />,
nationalité = <xsl:value-of select="." /><br />
</xsl:template>

<xsl:template match="cent50">
 Pièces de 50 cents : nombre = <xsl:value-of select="@nombre" />,
nationalité = <xsl:value-of select="." /><br />
</xsl:template>

</xsl:stylesheet>
```

Il faut avouer qu'il était simple, mais répétitif. Attention cependant, ce n'est pas la seule solution qui existe !

Programmez dans vos stylesheets !

Cette partie n'est pas vraiment indispensable, mais je vous la recommande vivement ! En effet, il est possible de programmer dans vos *stylesheets*... Allons-y ! 

La structure if et choose

Il existe dans ce langage des structures conditionnelles : *if* et *choose*. Ne vous méprenez pas, il n'y a pas de *else*...

La structure if

Voici d'emblée le code :

Code : XML

```
<xsl:if test="a == b">
    <!--Le code-->
</xsl:if>
```

If (= « si » en anglais) permet de tester une opération. Ici, il s'agit de « a=b ». Les opérations booléennes sont les mêmes que pour n'importe quel langage de programmation :

- '<' pour « si ... inférieur à ... ».
- '>' pour l'inverse.
- '<=' pour « ... inférieur ou égal ... ».
- '>=' pour « ... supérieur ou égal ... ».
- '==' pour « ... égal ... » (attention il y a deux signes « égal »).
- '!=' pour « ... différent de ... ».

Cette structure est pratique, si on utilise une fonction pour la canaliser. On apprendra ça plus tard.

La structure choose

Étant donné que le « *else* » (sinon) n'existe pas dans la structure *if*, une autre structure a été développée. Il s'agit d'une structure où il existe des « cas » selon la valeur d'entrée :

Code : XML

```
<xsl:choose>
    <xsl:when test="test = a">
        <!--Code-->
    </xsl:when>
    <xsl:when test="test = b">
        <!--Code-->
    </xsl:when>
            ...
</xsl:choose>
```

Ici, le code ne sert à rien. Il est encore une fois préférable de l'utiliser dans une fonction.

Notez qu'il existe une condition nulle, c'est-à-dire appelée si aucun des cas ne correspond à la valeur d'entrée. Cette balise est :

Code : XML

```
<xsl:otherwise>
    <!--Code-->
</xsl:otherwise>
```

Les fonctions

Enfin les voilà !
Vous allez voir, c'est très simple. Pour déclarer une fonction, on utilise cette balise :

Code : XML

```
<xsl:template name="nomFonction">
    <!--Le code de votre fonction-->
</xsl:template>
```

Alors, ça ne vous rappelle rien ?
Et pour l'appeler, on utilise celle-là :

Code : XML

```
<xsl:call-template name="nomFonction" />
```

Dans les deux cas, « nomFonction » désigne le nom de votre fonction (vous pouvez le changer si vous le voulez). Je pense que ça paraît logique, mais il faut **toujours** déclarer une fonction avant de l'appeler.

Faites attention : *call-template* n'est pas pareil que *apply-templates*, ne vous trompez pas ou vous aurez de graves

 erreurs !

Bon ben on a fini, salut.

 Eh attends ! Où sont les paramètres ?

Il vous faut toujours quelque chose !
Bon allez... Je suis sympa.
Il suffit juste de modifier un peu le code de la fonction :

Code : XML

```xml
<xsl:template name="nomFonction">
   <xsl:param name="parametre" select="0" />
</xsl:template>
```

Et pour l'appeller :

Code : XML

```xml
<xsl:call-template name="NomFonction">
    <xsl:with-param name="parametre" select="375" />
</xsl:call-template>
```

Pour utiliser une variable dans une fonction, on inscrit (dans une chaîne de caractères) le symbole « $ » juste devant le nom de la variable, comme cela :

Code : XML

```xml
<xsl:template name="afficherVariale">
    <xsl:param name="variable" />

    <xsl:if test="$variable < 25">
        <xsl:value-of select="$variable" /><!--On a le droit de
faire ça !-->
    </xsl:if>
</xsl:template>
```

Et voilà le travail !  En plus ça vous montre comment utiliser « xsl:if » dans une fonction. Une précision toutefois, la balise « xsl:with-param » peut s'écrire de deux façons :

Code : XML

```xml
<xsl:with-param name="parametre" select="123" />
<!--Ou alors-->
<xsl:with-param name="parametre">123</xsl:with-param>
```

C'est toujours pratique si on veut mettre un « <xsl:value-of /> » entre les balises !

La balise *for-each*

Cette balise est un peu spéciale... Il s'agit d'une boucle, mais ça n'en est pas vraiment une. En fait, cette balise va appeler le *parsage* de toutes les balises présentes dans la requête de l'attribut du *for-each*...
Ouais je sais : c'est difficile à avaler. Le plus dur, c'est cette histoire de requête XPATH... on verra ça plus loin...

Conclusion

Il faut garder à l'esprit qu'il existe sûrement d'autres balises dans le XSLT. Je vous ai présenté celles que je juge être les plus importantes. Abordons un point plus difficile maintenant : le XPATH.

Le XPATH

Le XPATH n'est pas *vraiment* un langage. C'est juste une manière d'écrire pour accéder aux divers n%u0153uds d'un arbre XML. Tout d'abord, un arbre est un document XML, dans le sens où il y a des balises mères et des balises filles. On peut donc construire un arbre avec toutes ces balises. Dans ce code :

Code : XML

```
<AAA>
    <BBB></BBB>
    <CCC></CCC>
    <BBB>
        <HHH att="1" />
        <HHH att="2" />
        <HHH fromage="guda" />
        <HHH />
        <HHH />
    </BBB>
    <BCC>
        <BCD>
            <CDB />
        </BCD>
    </BCC>
    <DDD>
        <EEE />
        <FFF />
        <EFG />
    </DDD>
    <GGG>
        <BBB></BBB>
    </GGG>
    <III>
        <JJJ nom="jjj" />
        <JJJ nom=" jjj" />
    </III>
</AAA>
```

Si on part de la balise BCC :

- On peut dire que BCC est un enfant.
- On peut dire que BCC possède deux descendants : BCD et CBD, mais il ne possède qu'un enfant : BCD.
- On peut dire que BCC est parent de BCD.
- On peut dire que BCC possède un ancêtre : AAA.
- On peut dire que BCC a 6 frères : BBB, CCC, BBB, DDD, GGG et III.

Et grâce au XPATH, on peut se déplacer de balise en balise à l'aide de ces notions.
C'est ce code que j'utiliserai durant tout le reste de cette sous-partie.

Faites attention, le code XPATH se rapporte toujours à la balise en cours de *parsage* !

Les requêtes

Le XPATH, comme le SQL, utilise la notion de requêtes : on demande quelque chose.
Les voici :

La syntaxe de base

On peut apparenter le XPATH aux chemins de fichiers comme C:\Blable\Blibli\...
Ainsi, la syntaxe /nomBalise correspond à la balise nommée « nomBalise » qui se trouve être mère mais pas fille. Dans le fichier XML ci-dessus, si on fait /AAA, alors une sorte de curseur (virtuel bien sûr) va se positionner dans la balise AAA. S'il y a plusieurs balises du même nom, il y aura plusieurs curseurs, chacun traité dans l'ordre.

Attention : dans les chemins de fichiers, on utilise l'antislash : \. Ici, seul le slash : / est autorisé !

À partir de là, on peut accéder à des sous-balises simplement comme avec /AAA/CCC, qui placera le curseur dans la balise CCC qui se trouve dans la balise AAA. On peut continuer ainsi de balise en sous-balise...

Si il y a un *slash* avant le AAA, c'est pour commencer de la racine, c'est-à-dire le document qui « héberge » l'arbre.

Cela signifie que s'il y a une balise CCC dans une balise BBB, elle ne sera pas prise en compte !

Tous les éléments d'un type

Avec le double *slash* (//), toute balise du nom suivi seront prises en compte : mère comme fille. Donc si je fais //BBB, toutes les balises « BBB » seront prises en compte, que ce soit dans la balise AAA, ou dans les autres balises.

Donc avec //DDD/EEE, toutes les balises EEE filles de DDD seront sélectionnées.

L'astérisque

L'astérisque (*) après un chemin désigne toutes les balises se trouvant après le début du chemin. Si je fais /AAA/DDD/*, alors les balises EEE, FFF et EFG seront prises en compte.

Et ça marche dans l'autre sens !
Avec ce code : /*/*/EEE, je sélectionne toutes les balises s'appelant EEE et qui ont deux ancêtres (mère et grand-mère) ! Cela veut dire qu'une balise EEE qui a un ou trois ou plus d'ancêtres ne sera pas prises en compte.

Encore une astuce : //*, si on est logique, ce code sélectionne toutes les balises !

Les crochets

Une expression suivie d'un numéro entre crochets désigne l'énième élément d'une balise. Ainsi, l'élément /AAA/BBB/HHH[1] sera la première balise HHH dans une balise BBB, elle-même dans une balise AAA.

Notez que /AAA[last()] sélectionne le dernier élément nommé AAA.

Arobase

L'arobase (@) désigne un attribut d'une balise ou d'un document en général. Si je fais //@att, alors tous les attributs nommés « att » seront sélectionnés. Si je veux toutes les balises possédant un attribut nommé « att », il suffit de mettre l'arobase et le nom de l'attribut entre crochets : /AAA/HHH[@att], ce code va sélectionner toutes les balises HHH possédant un attribut nommé « att ».

Si on suit la logique, //HHH[@*] désigne toutes les balises HHH possédant un attribut. Et encore une astuce : //HHH[not(@*)] désigne toutes les balises ne possédant pas d'attribut.

On peut sélectionner encore plus durement : en indiquant que l'on ne considère que les balises HHH possédant un attribut fromage égal à « guda » comme ici : //HHH[@fromage='guda'].

Notez ici que l'on utilise des *quotes* et non des guillemets !

Il y a bien sûr un problème : si je fais //JJJ[@nom='jjj'], alors seule la balise <JJJ nom="jjj" /> sera sélectionnée mais pas l'autre ! Une seule fonction dans ce cas : la fonction « *normalize-space* » : elle va enlever les espaces avant et après la valeur de l'attribut, et va confondre les espaces consécutives : //JJJ[normalize-space(@nom)='jjj'], ce code sélectionne toutes les balises JJJ possédant un attribut « nom » qui contient « jjj » une fois qu'on lui a enlevé les espaces avant et après (si toutefois il y en a).

Combinons des chemins

Avec la barre (|), on peut combiner des chemins :
//CCC|//BBB, ce code sélectionne toutes les balises nommées BBB ou CCC. Et vous pouvez mettre autant de barres que vous voulez, avec des dizaines de chemins différents !

Quelques fonctions

Nous avons vu précédemment des fonctions comme « not() » ou « normalize-space() ». En voilà quelques autres pour affiner vos requêtes.

La fonction count()

Pour ceux qui ne le sauraient pas, « *count* » veut dire « compte » en anglais. Cette fonction compte le nombre de balises qui ont l'enfant mis en paramètre, et le nombre d'enfant(s) mis après le symbole « égal » :
//*[count(JJJ)=1], ce code va sélectionner toutes les balises ayant un seul et unique enfant JJJ.
//*[count(*)=5], ce code va sélectionner toutes les balises ayant 5 enfants.

La fonction name()

Voilà une fonction intéressante : cette fonction va sélectionner toutes les balise du nom passé en paramètre et les balises enfantes du chemin :
//*[name()='GGG'], ce code va sélectionner toutes les balises GGG.
Plus fort : //*[start-with(name(), 'B')], ce code va sélectionner toutes les balises dont le nom commence par un B.
Encore plus fort : //*[contains(name(), 'G')], ce code sélectionne toutes les balises dont le nom contient un G.

La fonction string-length()

Cette fonction va sélectionner des balises avec des nombres de caractères.
//*[string-length(name())=3], ce code sélectionne toutes les balises dont le nom comporte 3 caractères.
Vous pouvez utiliser inférieur à (<) et supérieur à (>), mais en respectant les normes W3C.

Les "axes"

En XPATH, il existe des « axes ». Il s'agit en fait d'une sorte de sélection de balises qui repose uniquement sur la notion

d'enfants, d'ancêtres, de descendants, etc.

On marque un axe comme ceci :
//AAA/ancestor::CCC, où « *ancestor* » est le nom d'un axe.

L'axe « descendant »

L'axe « *descendant* » sélectionne tous les descendants de la balise précédente du chemin. Si je fais /AAA/descendant::*, alors je sélectionne tous les éléments descendants de AAA. Si je fais /AAA/descendant::BBB, alors je sélectionne tous les éléments descendants de AAA, et nommés BBB.

L'axe « parent »

L'axe « *parent* » sélectionne les parents directs de la balise précédente du chemin, si toutefois il y en a un. Si je fais //BBB/parent::*, alors je sélectionne tous les parents directs de BBB, comme AAA, et GGG. Si je fais //BBB/parent::GGG, alors je sélectionne tous les parents de BBB nommés GGG.

L'axe « ancestor »

L'axe « *ancestor* » sélectionne tous les éléments ancêtres de la balise précédente dans le chemin. Si je fais //BCD/ancestor::*, alors toutes les balises ancêtres de BCD seront sélectionnées, comme BCC et AAA. Je ne vous explique pas ce qui se passe si on indique une balise à la place de l'astérisque (pour rappel, la signification de l'astérisque, qui est aussi valable ici, est détaillée plus haut )...

Les axes « following » et « preceding »

L'axe « *following* » sélectionne tous les éléments suivant la balise précédente dans le chemin, sauf les éléments descendants de celle-ci. Si je fais /AAA/DDD/following::*, alors tous les éléments situés après l'élément DDD seront sélectionnés, sauf les descendants de DDD.

L'axe « *preceding* », quant à lui, sélectionne tous les éléments précédant la balise précédente dans le chemin, sauf les éléments descendants de celle-ci. Si je fais /AAA/DDD/preceding::*, alors tous les éléments situés avant l'élément DDD seront sélectionnés, sauf les descendants de DDD.

Les axes « following-sibling » et « preceding-sibling »

L'axe « *following-sibling* » sélectionne tous les éléments frères suivant la balise précédente dans le chemin. Si je fais /AAA/BCC/following-sibling::*, alors je ne sélectionne que les balises DDD, GGG, et III.

L'axe « *preceding-sibling* », quant à lui, sélectionne tous les éléments frères précédant la balise précédente dans le chemin. Si je fais /AAA/BCC/preceding-sibling::*, alors je ne sélectionne que les balises BBB, CCC et un autre élément BBB.

L'axe « self »

L'axe « *self* » sélectionne tous les éléments du même nom que celui de la balise précédente dans le chemin. Si je fais /AAA/BBB/self::*, alors tous les éléments nommés BBB seront sélectionnés.

Quelques précisions

Le XPATH n'est peut-être pas un langage, cela n'en fait pas une branche facile du XML. Retenez les termes utilisés pour un arbre, comme ancêtre, enfant, fille, etc.
Gardez à l'esprit que l'on peut mettre un nom de balise à la place de l'astérisque, et que cela signifie que les éléments sélectionnés porteront tous le nom de la balise en question.

Comment utiliser tout ça...

Un exemple vaut mieux qu'un laïus, alors :

Code : XML

```
<AAA>
    <BBB></BBB>
    <CCC></CCC>
    <BBB>
        <HHH att="1" />
        <HHH att="2" />
        <HHH fromage="guda" />
        <HHH />
        <HHH />
    </BBB>
    <BCC>
        <BCD>
            <CDB />
        </BCD>
    </BCC>
    <DDD>
        <EEE />
        <FFF />
        <EFG />
    </DDD>
    <GGG>
        <BBB></BBB>
    </GGG>
    <III>
        <JJJ nom="jjj" />
        <JJJ nom=" jjj" />
    </III>
</AAA>
```

Oui je sais : je suis très original...

Imaginons que je veuille *parser* seulement les balises JJJ avec un attribut « nom » dont la valeur est « jjj », sans compter les espaces, et filles de III lorsque je rencontre la balise AAA :

Code : XML

```
<xsl:template match="AAA">
    <xsl:for-each select="//III/JJJ[normalize-space(@nom)='jjj']">
        <em>Attribut : </em><xsl:value-of select="@nom" />
    </xsl:for-each>
</xsl:template>
```

Mais t'as pas parlé du point !

C'est vrai. Mais vous savez à quoi il sert. Donc pourquoi rabâcher ce que je vous ai déjà dit. Vous êtes des Zéros, certes, mais des Zéros intelligents (enfin j'espère ! ) !

Q.C.M.

Le premier QCM de ce cours vous est offert en libre accès.
Pour accéder aux suivants

Connectez-vous Inscrivez-vous

Quel est le *namespace* employé dans le XSLT ?

- ◯ xslt
- ◯ stylesheet
- ◯ xml
- ◯ xsl
- ◯ il n'y a pas de namespace

Quel est le groupe de balises délimitant une *stylesheet* ?

- ◯ <stylesheet></stylesheet>
- ◯ <xslt:stylesheet></xslt:stylesheet>
- ◯ <xsl:stylesheet></xsl:stylesheet>
- ◯ <xsl></xsl>
- ◯ <xsl:xslt></xsl:xslt>

Quelles balises sont utilisées pour délimiter un *parsage* ?

- ◯ <xsl:template match="nomBalise"></xsl:template>
- ◯ <xslt:template match="nomBalise"></xslt:template>
- ◯ <template match="nomBalise"></template>

Il existe <xsl:if></xsl:if>. Existe-t-il <xsl:else></xsl:else> ?

- ◯ ça existe, xsl:if ?
- ◯ oui
- ◯ non
- ◯ peut-être...

Quelle(s) balise(s) permet(tent) d'appeler une fonction ?

- ◯ <xslt:call-template name="fonction"></xslt:call-template>
- ◯ <xsl:call-template name="fonction"></xsl:call-template>
- ◯ <xsl:call-function name="fonction"></xsl:call-function>
- ◯ <xslt:call-function name="fonction"></xslt:call-function>

Quels seront les attributs des balises sélectionnées dans le code ci-dessous et avec le code XPATH : /AAA/BBB

Code : XML

```xml
<AAA>
    <BBB att="1" />
    <BBB name="bbb" />
    <CCC>
        <BBB ake="tux" />
    </CCC>
</AAA>
```

- ◯ BBB
- ◯ AAA, BBB et CCC
- ◯ att, name et ake

- ◯ ake
- ◯ att et name
- ◯ C'est une question piège !

Dans le XPATH : /AAA/BBB/HHH[last()]//BBB[@*], quelles seront les valeurs des attributs des balises sélectionnées (code ci-dessous) ?

Code : XML

```xml
<AAA>
    <BBB>
        <HHH id="ak" />
        <HHH id="ek" />
        <HHH id="ouk" />
        <HHH />
        <HHH id="crabe" />
    </BBB>
    <CCC>
        <BBB id="ka" />
        <BBB id="ko" />
        <BBB id="koua" />
    </CCC>
</AAA>
```

- ◯ ka, ko, ak et ek
- ◯ crabe et ak
- ◯ id, id et id
- ◯ ouk et koua
- ◯ ak et ek
- ◯ ka, ke, koua et crabe
- ◯ Je peux téléphoner à un ami ?

Allez, une dernière pour la route. Je vous demande la valeur des attributs des balise(s) sélectionnée(s). Voici le XPATH : /AAA/DDD/following-sibling::*|/AAA/BBB/HHH[3]//BCC/descendant::CDB|//EFG/ancestor::*

Code : XML

```xml
<AAA arbe="mon super-arbre multi-fonctions">
    <BBB id="tux"></BBB>
    <CCC id="86"></CCC>
    <BBB id="a">
        <HHH att="1" />
        <HHH att="2" />
        <HHH fromage="guda" />
        <HHH />
        <HHH />
    </BBB>
    <BCC vert="00ff00">
        <BCD att="26">
            <CDB tux="piano" />
        </BCD>
    </BCC>
    <DDD quatre="4">
        <EEE id="e" />
        <FFF id="f" />
        <EFG id="g" />
    </DDD>
    <GGG nom="ggg">
        <BBB a="abc"></BBB>
    </GGG>
    <III belfort="90">
```

```
            <JJJ nom="jjj" />
            <JJJ nom=" jjj" />
        </III>
    </AAA>
```

- ◯ Il n'y a pas de réponse
- ◯ ggg, 00ff00, jjj, abc
- ◯ mon super-arbre multi-fonctions
- ◯ 86, 1, 2, 26, 4, 90, et le numéro complémentaire, le 25
- ◯ euh...
- ◯ 4, e, f, g, ggg, abc, jjj
- ◯ tux, guda, piano, saucisse, salade, choucroute
- ◯ ggg, 90, guda, piano, 4, mon super-arbre multi-fontions

Correction !

Statistiques de réponses au QCM

Et voilà ! 
Vous êtes devenus (presque) maîtres dans l'art de la *stylesheet*. Alors à vos marques, prêts, *parsez* !


Ce tutoriel a été corrigé par les zCorrecteurs.