Introduction à XPath

Code: xml-xpath

Originaux

url: **http://tecfa.unige.ch/guides/tie/html/xml-xpath/xml-xpath.html**

url: **http://tecfa.unige.ch/guides/tie/pdf/files/xml-xpath.pdf**

Auteurs et version

- Daniel K. Schneider

- Version: 0.4 (modifié le 2/12/05)

Prérequis

Module technique précédent: **xml-tech**

Module technique précédent: **xml-xslt** (il faut un minimum de XPath pour XSLT de base)

Module technique précédent: **xml-xslt2** (il faut connaître les fonctions XPath)

Abstract

Petite introduction à XPath

Objectifs

- Eléments XPath nécessaires pour faire du XSLT niveau débutant et moyen

A faire

- Plus de fonctions
- La notion générale d'expression XPath

1. Table des matières détaillée

2. Introduction à XPath

2.1 Rôle de XPath

- Historique: XPath est le résultat d'un effort d'homogénéisation de la syntaxe et de la sémantique de fonctions communes à XSLT, XPointer, etc.
- XPath n'est pas un langage XML (utilise une autre syntaxe)

Objectifs

1. Définir la manière d'adresser des parties d'un document XML (chemins d'accès)
   - Un chemin d'accès ressemble un peu à celui des noms de fichiers (chemins), d'où le nom "XPath"
2. Traiter des chaînes de caractères, des nombres et des booléens (cela rend ce langage plus sophistiqué par rapport à l'objectif 1)

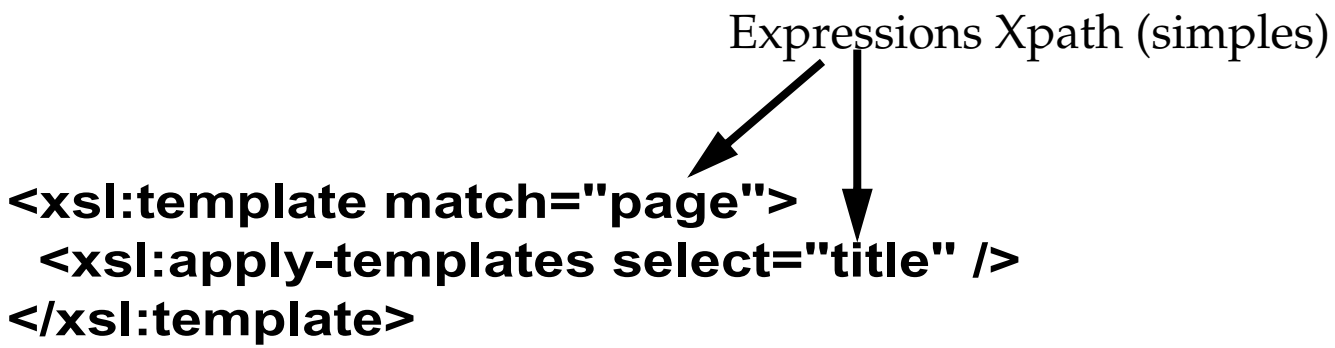
Statut

- XPath est un standard W3C:

  url: **http://www.w3.org/TR/xpath**
- XPath est utilisé dans des langages comme XSLT, XQuery, XInclude etc. ainsi que dans des librairies XML des langages de programmation
- XPath donc largement utilisé actuellement

2.2 XPath dans le contexte de XSLT

- XPath faisait d'abord partie de XSLT avant d'être formalisé comme standard à part.



- XSLT utilise XPath par ex. pour indiquer à quel élément il faut appliquer une règle
  - Cet élément (par ex. "path") est indiqué par un chemin de localisation
- Les débutants peuvent utiliser des simples noms de balises:

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

- Pour les experts, ces expressions peuvent devenir très compliquées:

```
<xsl:apply-templates select="cours/module[position()=1]/section[position()=2]"/>
```

(cela veut dire: "la 2ème section du premier module du cours")

3. Les Chemins de localisation

- Un chemin de localisation trouvera en ensemble de noeuds (éléments, attributs, commentaires, instructions de traitement, etc.)
- Chaque chemin de localisation peut être exprimé soit avec une syntaxe verbeuse soit avec une version courte (abbreviatedXYZ)

Début de la définition formelle des chemins de localisation ....

```
[1] LocationPath::= RelativeLocationPath | AbsoluteLocationPath
[2] AbsoluteLocationPath ::= '//' RelativeLocationPath?
                           | AbbreviatedAbsoluteLocationPath
[3] RelativeLocationPath ::= Step | RelativeLocationPath '//' Step
                           | AbbreviatedRelativeLocationPath
[4] Step ::= AxisSpecifier NodeTestPredicate* | AbbreviatedStep
[5] AxisSpecifier ::= AxisName '::' | AbbreviatedAxisSpecifier
[6] AxisName ::= 'ancestor' | 'ancestor-or-self' | 'attribute' | 'child' |
'descendant' | 'descendant-or-self' | 'following' | 'following-sibling' |
'namespace' | 'parent' | 'preceding' | 'preceding-sibling' | 'self'
```

Note

- Ce début de la définition formelle montre qu'il s'agit ici de qqch. de complexe
- Il y a en tout 39 clauses ...
- Nous allons donc seulement introduire les concepts les plus importants

3.1 Le modèle du document de XPath

- XPath voit un document sous forme d'arborescence.
- Chaque élément d'information (éléments XML, attributs XML, texte, etc.) est appelé "noeud" (node)
- Ce modèle ressemble au DOM (mais n'est pas identique !)

Noeuds que XPath peut voir:

- noeud racine (root node)
  - ATTENTION cette racine n'est pas forcément celle de l'élément racine du XML. XPath verrait en premier des instructions de traitement (feuille de style, etc.) ou commentaires au début de fichier !
- noeuds d'éléments et d'attributs
- noeuds de commentaires, d'instructions de traitement, de namespace.

Noeuds que XPath ne peut pas voir:

- XPath voit le document une fois qu'il construit, donc ne voit pas les entités, déclarations de type de document, etc.

La notion de contexte:

- Pour comprendre ce que fait une expression XPath il faut toujours regarder son contexte d'utilisation (cf. suite)

3.2 Quelques chemins de localisation avec la syntaxe courte

A. Quelques chemins simples: éléments enfants, parents, cousins

Noeud racine: /

retourne le premier noeud trouvé dans un arbre (pas forcément l'élément racine XML !)

Elément enfant direct:

Syntaxe: **nom_element_XML**

Elément enfant direct du noeud racine:

Syntaxe: **/nom_element_XML**

Enfant d'un enfant:

Syntaxe: **nom_element_XML/nom_element_XML**

Descendant arbitraire du noeud racine:

Syntaxe: **//nom_element_XML**

Descendant arbitraire d'un noeud:

Syntaxe: **nom_element_XML//nom_element_XML**

Un parent d'un noeud:

Syntaxe: **../**

Un cousin lointain d'un noeud:

Syntaxe: **../../nom_element_XML/nom_element_XML/nom_element_XML**

Illustrations avec un document XML exemple et quelques règles XSLT:

```
<project>
 <title>Mon project</title>
 <problem>
    <title>Voici un problème></title>
    <description>Je ne connais pas XPATH</description>
 </problem>
 <solutions>
   <item val="moyenne">Vous pouvez acheter un livre sur XSLT</item>
   <item val="forte">Vous pouvez suivre un cours et faire des exercices</item>
 </solutions>
</project>
```

Règle (1) XSLT pour la racine XML

```
<xsl:template match="/project">
  <xsl:apply-templates select="title" />
</xsl:template>
```

- xsl:template est une règle XSLT (voir XSLT pour plus d'informations !)
- L'attribut "match" dit que cette règle s'applique à l'élément **`project`** sous la racine
- Le contexte d'exécution est donc l'élément "**`project`**"
- xsl:apply-templates permet de déclencher une autre règle. Donc le "**`title`**" référencé par l'attribut sélect est "Mon projet" (parce que ce **`title`** est un enfant de project)

Règle (2) XSLT pour l'élément problem

```
<xsl:template match="/project">
  <xsl:apply-templates />
</xsl:template>

<xsl:template match="problem">
  <xsl:apply-templates select="title" />
</xsl:template>
```

- La deuxième règle sera déclenchée par la première règle, car **problem** est un élément enfant de **project.** Autrement dit, quand on se place dans le contexte "**project**", problem est un descendant direct.
- Une fois quand la règle est déclenchée on se retrouve dans le contexte "**problem**". Donc "**title**" se réfère à "Voici un problème".

B. Chemins pour chercher des attributs

Chercher un attribut d'un élément (contexte courant)

Syntaxe: **@nom_attribut**

Chercher tous les attributs "bla"

Syntaxe: //@bla

C. Chemins avec Wildcards

Chercher tous les noeuds de type élément

Syntaxe: *

Chercher tous les noeuds (y compris commentaires, etc.)

`Syntaxe: node()`

Chercher tous les attributs

`Syntaxe: @*`

D. Chemin avec prédicats

- Un prédicat est une ***expression logique qui sera vraie ou fausse*** et ***affine le résultat obtenu*** avec un chemin de recherche.

Chercher un élément qui a un attribut

`Syntaxe:` **`nom_element_XML [ @nom_attribut ]`**

Chercher un élément qui a un attribut avec une certaine valeur

`Syntaxe:` **`nom_element_XML [ @nom_attribut = 'valeur']`**

Chercher un élément qui a un autre élément comme enfant

`Syntaxe:` **`nom_element_XML [ nom_element_XML ]`**

Règle (3) XSLT pour définir un chemin en fonction de la valeur d'un attribut

```
<xsl:template match="//item[@val='forte']">
  <xsl:value-of select="." />
</xsl:template>
```

- Cette règle se déclenche dans le contexte racine
- Elle sélectionne n'importe quel élément "item" qui a comme attribut "val=forte"

3.3 Récapitulatif de chemins simples de localisations

Elément syntaxique	**(Type de chemin)**	**Exemple d'un chemin**	**Exemple d'un match réussi par rapport au chemin indiqué à gauche**
balise	nom d'élément	project	***<project> ..... </project>***
/	sépare enfants directs	project/title	<project>***<title> ... </title>***
		/	(correspond à l'élément racine)
//	descendant	project//title	<project><problem>***<title>....</title>***
		//title	<racine>...***<title>..</title>*** (n'mporte où)
*****	"wildcard"	*/title	<bla>***<title>..</title> et*** <bli>***<title>...</title>***
|	opérateur "ou"	title|head	<title>...</title> ***ou*** <head> ...</head>
		|/|@	(tous les éléments: les enfants, la racine et les attributs de la racine)
.	élément courant	.	
../	élément supérieur	../problem	<project>
@	nom d'attribut	@id	<xyz id="test">...</xyz>
		project/@id	***<project id="test" ...> ... </project>***
@attr='type'		list[@type='ol']	***<list type="ol"> ..... </list>***

4. Les fonctions XPath

- XPath définit un certain nombre de fonctions
- Chaque fonction retourne soit une valeur booléenne (vrai/faux), un nombre, un string (chaîne de caractères), ou encore une liste de noeuds
- Une fonction est identifié par le fait qu'elle a des "()" à la fin. Il existe des fonctions avec zéro, un ou plusieurs arguments, et finalement un nombre arbitraire d'arguments
- Par rapport à XSLT: Certaines fonctions peuvent être utilisées dans des chemins de localisation, d'autre seulement avec <xsl:value-of select=""./>

4.1 Fonctions pour les ensembles de noeuds

- Voici les fonctions les plus importantes

last()

La fonction last retourne le nombre de noeuds qui se trouvent dans le contexte (qui ont le même parent)

position()

La fonction position retourne le nombre de la position contextuelle (context position) d'un élément par rapport à son parent.

count(node-set)

La fonction count retourne le nombre de noeuds de l'ensemble de noeuds passés en arguments.

4.2 Choix de fonctions pour les chaînes de caractères et les nombres

- Voir un manuel ou la spécification pour les détails

starts-with(string, string)

retourne TRUE si le deuxième string se trouve au début du premier

contains(string, string)

retourne TRUE si le deuxième (!) string se trouve dans le premier

string-length(string?)

retourne la longeur d'un string

number(object?)

transforme un objet en nombre

sum(node-set)

la somme de nombres trouvés dans un ensemble de noeuds.
Effectue une conversion de strings si nécessaire, comme number()

round(number)

arrondit un nombre selon les conventions habituelles: 1.4 devient 1 et 1.7 devient 2

Exemples

```
//Etudiant[starts-with(Prenom,'Bernadette')]"
//Employee[contains(FirstName,'John')]
```

Pour les autres fonctions, voir un manuel

- il existe d'autres fonctions "strings/numbers", ainsi qu'une catégorie fonctions booléennes

4.3 Expressions

Calculs arithmétiques

- On utilise des expressions habituelles, sauf pour la division !! (**`div`** au lieu de `/`")

  `Syntaxe: + - * div mod`
- mod est intéressant pour calculer des tableaux et autres visualisations

  `5 mod 2 retourne 1, "7 mod 2" et "3 mod 2" aussi`
  - on pourrait donc utiliser cette information pour placer ces éléments dans une colonne ...

Opérateurs booléens

- XSLT utilise la notation habituelle. Selon règles de précédence des opérateurs:

  `<=, <, >=, >`
  `=, !=`
  `and`
  `or`

Exemples

- l'expression suivante choisit le titre de tous les exercices ayant une note qui dépasse 5.

  `//exercise[note>5]/title`
  - l'expression suivante additionne 2 variables
- Retourner tous les noeuds **`Participant`** ayant un contenu de **`Nom`** dépassant 7 caractères:

  `"//Participant[string-length(Nom)>=8]"`
- Retourner les 5 derniers éléments d'une liste

  `author [(last() - 4) <= position()) and (position() <= last())]`

5. Chemins de localisation non abrégés

Principe d'un chemin de localisation

- Un chemin de localisation est définit comme une séquence d'étapes de localisations
- Ces étapes de localisation sont séparées par des "/"

```
Syntaxe: (simplifiée)
RelativeLocationPath ::= Step | RelativeLocationPath '//' Step
```

Les étapes de localisation (step)

```
Syntaxe: (simplifiée)
Step ::= AxisName '::' NodeTest Predicate*
```

- Une étape de localisation (step) contient 2 éléments obligatoires et un élément à option:

1. Un axe (AxisName): définit dans quelle direction il faut chercher
2. Un noeud test (NodeTest): définit parmi quels noeuds il faut chercher
3. Un ou plusieurs prédicats à option: pour raffiner la sélection obtenu avec (1) et (2).

Exemples:

- chercher la valeur de l'attribut "val" pour l'enfant child, de l'enfant solution

```
child::solutions/child::item/@val
```

correspond à:

```
solutions/item/@val
```

Quand utiliser ?

- Lorsque la syntaxe simple ne s'avère pas assez puissant pour identifier un axe de recherche