Perl et les fichiers XML

Analyse et création des fichiers XML

par djibril (site personnel) Nicolas Vallée (Home page)

Date de publication : 2009-07-08

Dernière mise à jour : 2011-03-25

Cet article vous explique comment lire et écrire des fichiers XML en utilisant certains modules existants sur le site du CPAN.

I - Introduction aux fichiers XML

I-1 - Qu'est-ce qu'un fichier XML ?

XML (e**X**tensible **M**arkup **L**anguage pour *« langage extensible de balisage »*), est le standard défini par le **W3C** (World Wide Web Consortium) pour représenter des données dans des documents balisés.
Markup Language: langage de balise, comme le bien connu HTML.
eXtensible: *a contrario* du HTML, le nombre, le nom et la fonction des balises ne sont pas prédéfinis.
La sémantique d'une balise est définie par l'auteur du document XML. C'est un langage informatique de balisage générique qui sert essentiellement à structurer des données. Cela peut avoir de nombreuses applications comme la définition d'un protocole de communication en mode texte (WebServices, etc.), la représentation texte d'opérations qui seront interprétées par la suite (XSL, XSQL...), la définition d'un nouveau langage de représentation à balises (XHTML, XUL...).

Un fichier XML se présente sous la forme suivante :

1. une information d'encodage et la norme XML utilisée (ex: **<?xml version="1.0" encoding="ISO-8859-1"?>**) ;
2. une éventuelle déclaration de doctype (ex: **<!DOCTYPE annuaire SYSTEM "exemple.dtd" >**) ;
3. des données structurées sous forme arborescence et bien formée, c'est-à-dire :
   - une unique balise racine ;
   - toute balise doit être fermée, via les formes *<balise/>* ou *<balise>...</balise>* ;
   - les déclarations des balises ne peuvent se chevaucher, *<a1><a2></a1></a2>* est donc interdit.

Pour plus d'informations, **http://haypo.developpez.com/tutoriel/xml/introduction/**

I-2 - Les fichiers de spécifications de format en général

Depuis 2001, le W3C recommande l'utilisation de langages de description de format pour spécifier les formats XML utilisés par son application. Les formats les plus utilisés sont **DTD** et **XSD**. Une **DTD** (**D**ocument **T**ype **D**efinition) est un fichier permettant de vérifier qu'un document XML est valide et respecte un modèle donné par le créateur du XML (nom, ordre et nombre d'occurrences autorisées des sous éléments, etc.). La norme XML n'impose pas l'utilisation d'une DTD pour un document XML, mais elle impose par contre le respect exact des règles de base de la norme XML.

Un fichier doctype est un fichier texte structuré (SGML) servant à décrire un format XML. Vous y trouverez deux balises importantes :

- <!ELEMENT balise (contenu)> ;
- <!ATTLIST balise attribut1 type1 (contrainte1?) ... attributN typeN (contrainteN?).

Pour plus d'informations, **http://zvon.developpez.com/tutoriels/dtd/**

XSD (**X**ML **S**chema **D**ocument) ou **WXS** ((**W3C**) XML Schemas) désigne la norme en **XML Schemas** du W3C. Il a la particularité d'être lui-même au format XML, et donc analysable par les mêmes outils. Comparé à son prédécesseur DTD, il a l'avantage d'intégrer la notion de domaine de validité, et donc d'imposer des contraintes plus fines sur son format XML.

Un schéma XML est un fichier XML servant à décrire un format XML

- sa balise racine est forcément **<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">** ;
- tout d'abord, vous pouvez déclarer des types via les balises **xs:simpleType** et **xs:complexType** et leur donner un nom pour les réutiliser ultérieurement ;
- ensuite, il vous faudra de manière arborescente décrire la structure du format XML, et vous contraindre à déclarer les attributs (**xs:attribute**) d'un élément (**xs:element**) après ses éventuels fils.

Pour le déclarer dans votre fichier XML, il faudra ajouter ceci au niveau de la balise racine, exemple :

```
<NOM_BALISE_RACINE xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="ExempleXSD.xsd">
```

Pour plus d'informations, **XML Schema**

Il diffère donc du fichier DTD sur plusieurs points :

- il permet de définir des domaines de validité pour la valeur d'un champ, alors que cela n'est pas possible dans une DTD ;
- il ne permet pas de définir des entités comme le fait une DTD ;
- il est lui-même un fichier XML alors qu'une DTD est un document SGML (*Standard Generalized Markup Language* pour *langage normalisé de balisage généralisé*) et donc non XML en comparaison à XSD ;
- il est possible d'utiliser et de faire réagir très facilement des XSD entre eux ;
- il est impossible avec une DTD de définir deux éléments portant le même nom mais avec des contenus différents ;
- XSD permet d'avoir une définition locale d'un élément portant le même nom qu'un autre élément déclaré dans un autre contexte.

Rien ne vous empêche d'employer un fichier DTD et un fichier XSD pour le même document XML, une DTD définissant les entités et un XSD pour la validation.

Pour en savoir plus, **XML Schemas** et **DTD Tutorial from W3schools**

N.B. **XHTML** est un langage de balisage servant à écrire des pages pour le **W**orld **W**ide **W**eb. Censé être le successeur du **HTML**, il se fonde sur la syntaxe définie par **XML**. Il arrive souvent que le format ne soit pas respecté, mais qu'on ait besoin de traiter les données. Dans ce cas, Il est recommandé d'utiliser des parseurs spécifiques tels les modules **HTML::Parser** et **HTML::TreeBuilder** en Perl.

I-3 - Contexte d'utilisation des fichiers XML

Le XML est de nos jours utilisé dans différents contextes. Il permet d'afficher les données en fonction de la demande, à transmettre les données selon un certain "schéma" que l'on peut définir selon ses propres besoins. Par exemple, il peut permettre à deux applications complètement étrangères d'interagir par l'intermédiaire d'un format **SOAP** prédéfini.

I-4 - Comment analyser et manipuler des fichiers XML ?

L'analyse syntaxique et la manipulation des fichiers XML sont faisables via la plupart des langages de programmation (JAVA, PHP, C#, **Perl**...). Le but de notre article est bien entendu de vous montrer comment s'y prendre en utilisant le langage Perl.

II - Perl et XML

II-1 - Qu'est-ce qu'un parseur Perl de fichier XML ?

Un parseur XML est un outil d'analyse syntaxique. Il permet d'une part d'extraire les données d'un document XML (on parle d'analyse du document ou de parsing) ainsi que de vérifier éventuellement la validité du document. On distingue des parseurs validant (vérifiant qu'un document XML est conforme à sa DTD, XSD...) et non validant (vérifiant que le document respecte la syntaxe XML de base).

II-2 - Parseurs Perl

Les débutants en Perl ont généralement la mauvaise habitude de vouloir analyser un fichier XML à coups d'expressions régulières. C'est une très mauvaise idée car il existe de nombreux modules sur le CPAN le faisant proprement et rapidement. De plus ces modules sont utilisables sur toutes les plates-formes.

Les parseurs XML utilisent deux méthodes pour analyser les fichiers, **DOM** et **SAX**.

II-2-a - DOM (Document Object Model)

C'est une spécification de la norme W3C (World Wide Web Consortium) afin de structurer un document XML sous forme d'arbre afin de simplifier l'accès aux éléments, au contenu...

II-2-b - SAX (Simple API for XML)

La méthode SAX est basée sur un modèle événementiel, qui transforme un document XML en un flux d'évènements déclenchés par la lecture d'éléments syntaxiques XML (balise ouvrante, balise fermante...). Elle permet de manipuler un nœud du document XML à la fois, ce qui facilite le traitement de gros fichiers XML.

II-2-c - Comparaison entre SAX et DOM

Les méthodes SAX et DOM adoptent chacune une stratégie d'analyse différente. La méthode DOM charge l'intégralité d'un document XML dans une structure de données qui peut alors être manipulée facilement. **L'inconvénient** ? Le fichier entier doit être stocké en mémoire, ce qui peut poser des problèmes si celui-ci est volumineux. La méthode SAX apporte alors une alternative dans les cas de figure où les documents XML sont de taille très importante. Elle nous permet de traiter uniquement ce qui est nécessaire. Notez que la méthode SAX peut avoir aussi un inconvénient ! En fait, il se peut qu'on ait besoin d'informations liées mais ne se trouvant pas dans les mêmes nœuds. La mise en place du parseur peut donc devenir très compliquée.

N.B. Notez qu'en Perl, il existe un module très intelligent qui est capable de gérer très facilement les deux  .

II-2-d - Listes de parseurs XML Perl

Voici une liste non exhaustive des modules Perl nous permettant d'analyser et modifier des fichiers XML.

Nous ne les citerons pas tous car il en existe des centaines  , mais voici les plus utilisés !

- **XML::Simple**

Il a été conçu par *Grant McLean*, son but premier était de lire et mettre à jour des fichiers de configuration écrits en XML. Mais il peut être utilisé pour parser des fichiers XML simples. Tout est chargé en mémoire dans un hash. Notez bien que le mot "**Simple**" (de XML::Simple) signifie des fichiers avec une architecture simple. Il ne supporte pas des contenus mixés, exemple :

```
<document>
    <para>This is <em>mixed</em> content.</para>
</document>
```

Pour en savoir plus, veuillez vous referer à la FAQ **XML::Simple**

- **XML::XPath**

Il a été conçu par *Matt Sergeant* et est le premier module Perl implémentant la méthode DOM et supportant XPath. Nous ne le vous recommandons pas car il n'est plus maintenu et est à ce jour moins efficace que d'autres modules comme XML::LibXML utilisant DOM et XPath.

- **XML::DOM**

Il a été conçu par *Enno Derksen* et est le premier module Perl implémentant DOM. Il est à peine maintenu par *T.J. Mather*. XML::LibXML devrait être utilisé à sa place.

- **XML::SAX**

Il est maintenu par *Grant McLean*. Il comprend un certain nombre de méthodes dont vous pouvez avoir besoin si vous travaillez avec SAX.
Les auteurs de la spécification SAX Perl et des modules qui l'implémentent sont *Ken MacLeod, Matt Sergeant, Kip Hampton et Robin Berjon*.

- **XML::Parser**

Il a été conçu par *Larry Wall* et est maintenu actuellement par *Clark Cooper, Matt Sergeant*. Il est rapide, et de bas niveau. Il est basé sur la librairie Expat. Néanmoins, il est vraiment très difficile à prendre en main et à ce jour, il est préférable d'utiliser l'un des deux modules suivant :

1. XML::LibXML plus rapide ;
2. XML::Twig facilement utilisable.

- **XML::Rules**

Il est maintenu par *Jan Krynicky*. Ce module est basé sur XML::Parser. Il vous permet de spécifier des actions à exécuter une fois qu'une balise est entièrement analysée (y compris les sous-balises) et transmet les données produites par les actions à ceux qui sont attachés aux balises mères.

- **XML::LibXML (recommandé)**

Il a été conçu par *Matt Sergeant* et *Christian Glahn* et est activement maintenu par *Petr Pajas*. Il est rapide, complet et stable. Il peut être utilisé dans un contexte validant ou non validant et utilise DOM sous le support XPath. Sa méthode DOM et la gestion de la mémoire ont été écrites en C (il utilise libxml2 écrit en C) offrant ainsi des performances intéressantes. Il permet également de faire très facilement des transformations XSL en la combinant au module XML::LibXSLT (qui utilise la librairie écrite en C libxslt).

- **XML::Twig (recommandé)**

Il a été conçu par *Michel Rodriguez*. Il est stable, performant, simple d'utilisation et activement maintenu. Il fait partie des modules intelligents en ce sens que l'on peut faire du DOM en chargeant tout en mémoire, ou même parser uniquement des portions d'un fichier en ne chargeant en mémoire que ces dernières ! Il est également recommandé pour tous les fichiers XML, surtout les très gros ! Il comprend énormément de méthodes (une centaine !) permettant de jouer avec le fichier XML. Il peut également parser facilement des fichiers xhtml. Il est basé sur XML::Parser, lui-même basé sur Expat. C'est un parseur non validant.

III - Scripts - Exemples de parseurs XML en Perl

III-1 - Enoncé de l'exercice

Pour vous donner quelques exemples de codes vous permettant d'utiliser ces modules, nous utilisons trois fichiers (XML, un DTD et un fichier XSD).

ExempleXML.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE annuaire SYSTEM "ExempleDTD.dtd" >
<annuaire xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="ExempleXSD.xsd">
    <personne miseajour="2009-06-18T20:57:57">
        <prenom>djibril</prenom>
        <role>rédacteur</role>
        <email>djibril@monmail.com</email>
        <telephone>+33102030405</telephone>
```

ExempleXML.xml

```
        <langage name="perl">
          <ModulePreferes>XML::Twig</ModulePreferes>
          <ModulePreferes>XML::Writer</ModulePreferes>
          <ModulePreferes>Config::Std</ModulePreferes>
        </langage>
        <commentaire>
          Personne assez active sur le forum, et
          responsable d'un forum.
        </commentaire>
     </personne>

    <personne miseajour="2009-06-14T20:57:57">
        <prenom>gorgonite</prenom>
        <role>rédacteur</role>
        <email>gorgonite@mon.mail.com</email>
        <telephone>+33112131415</telephone>
        <langage name="perl">
          <ModulePreferes>XML::Simple</ModulePreferes>
        </langage>
        <commentaire>
          Personne assez active sur le forum, et
          rédacteur d'articles.
        </commentaire>
     </personne>

    <personne miseajour="2009-06-18T21:30:12">
        <prenom>stoyak</prenom>
        <role>rédacteur</role>
        <email>stoyak@monmail.com</email>
        <langage name="perl">
          <ModulePreferes>XML::LibXML</ModulePreferes>
          <ModulePreferes>DBI</ModulePreferes>
        </langage>
        <commentaire>
          Personne assez active sur le forum, et
          en charge de corrections d'articles.
        </commentaire>
    </personne>

    <!-- Il n'aime rien et/ou ne connait rien -->
    <personne  miseajour="2009-06-20">
        <prenom>jean</prenom>
        <role>lecteur</role>
        <email>jean@monmail.com</email>
        <commentaire>
          Personne assez..., bah en fait, ne fait rien :-) .
          Et oui malheureusement, ça existe.
        </commentaire>
    </personne>
</annuaire>
```

ExempleXSD.xsd

```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
<!-- http://xformsinstitute.com/essentials/email.xsd -->
   <xs:simpleType name="my-email-type">
      <xs:restriction base="xs:string">
         <xs:pattern value="[A-Za-z0-9!#-'\*\+\-/=\?\^_`\{-~]+(\.[A-Za-z0-9!#-'\*\+\-/=\?\^_`\{-
~]+)*@[A-Za-z0-9!#-'\*\+\-/=\?\^_`\{-~]+(\.[A-Za-z0-9!#-'\*\+\-/=\?\^_`\{-~]+)*" />
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="my-phone-number-type">
      <xs:restriction base="xs:string">
         <xs:pattern value="[+]?\d+" />
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="date-or-datetime">
      <xs:union memberTypes="xs:date xs:dateTime" />
   </xs:simpleType>
```

ExempleXSD.xsd

```
    <xs:element name="annuaire">
       <xs:complexType>
          <xs:sequence>
             <xs:element name="personne" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                   <xs:sequence>
                      <xs:element name="prenom" type="xs:string" maxOccurs="1" />
                      <xs:element name="role" type="xs:string" maxOccurs="1" />
                      <xs:element name="email" type="my-email-type" maxOccurs="1" />
                      <xs:element name="telephone" type="my-phone-number-
type" minOccurs="0" maxOccurs="1" />
                      <xs:element name="langage" minOccurs="0" maxOccurs="unbounded">
                         <xs:complexType>
                            <xs:sequence>

  <xs:element name="ModulePreferes" type="xs:string" minOccurs="0" maxOccurs="unbounded" />
                            </xs:sequence>
                            <xs:attribute name="name" type="xs:string" use="required" />
                         </xs:complexType>
                      </xs:element>
                      <xs:element name="commentaire" type="xs:string" maxOccurs="1" />
                   </xs:sequence>
                   <xs:attribute name="miseajour" type="date-or-datetime" />
                </xs:complexType>
             </xs:element>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
</xs:schema>
```

ExempleDTD.dtd

```
<!ELEMENT annuaire (personne*) >
<!ATTLIST annuaire
  xmlns:xsi CDATA #FIXED "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation CDATA #IMPLIED
>
<!ELEMENT personne (prenom,role,email,telephone?,langage?,commentaire)>
<!ATTLIST personne miseajour CDATA #IMPLIED>
<!ELEMENT langage ((ModulePreferes*)? | EMPTY)>
<!ATTLIST langage name CDATA #REQUIRED>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT role (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT commentaire (#PCDATA)>
<!ELEMENT ModulePreferes (#PCDATA)>
```

III-2 - Solutions

Le but est d'analyser ces fichiers XML et d'afficher un résultat dans un fichier texte qui ressemblera à celui-ci :

resultat.txt

```
Personne : djibril 18/06/2009 20:57:57
Role : rédacteur
Email : djibril@monmail.com
Telephone : +33-1-02-03-04-05
Langage prefere : perl
   - XML::Twig
   - XML::Writer
   - Config::Std
Commentaire :
          Personne assez active sur le forum, et
          responsable d'un forum.


==========
```

resultat.txt
```
Personne : gorgonite 14/06/2009 20:57:57
Role : rédacteur
Email : gorgonite@monmail.com
Telephone : +33-1-12-13-14-15
Langage prefere : perl
   - XML::Simple
Commentaire :
          Personne assez active sur le forum, et
          rédacteur d'articles.


==========

Personne : stoyak 18/06/2009 21:30:12
Role : rédacteur
Email : stoyak@monmail.com
Telephone :
Langage prefere : perl
   - XML::LibXML
   - DBI
Commentaire :
          Personne assez active sur le forum, et
          en charge de corrections d'articles.


==========

Personne : jean 20/06/2009
Role : lecteur
Email : jean@monmail.com
Telephone :
Commentaire :
          Personne assez..., bah en fait, ne fait rien :-) .
          Et oui malheureusement, ça existe.


==========
```

III-2-a - XML::Simple

Ce module permet de charger entièrement le fichier en mémoire sous forme d'arbre. En fait, il est stocké dans un hash. Les quelques lignes suivantes suffisent pour charger le fichier XML

```
my $parser = XML::Simple->new( KeepRoot => 1 );
my $doc = $parser->XMLin('ExempleXML.xml');
```

Si vous souhaitez voir l'architecture du hash, vous pouvez utiliser le module **Data::Dumper**. C'est important, car en fonction du fichier XML, l'architecture n'est pas la même.

```
use Data::Dumper;
print Dumper $doc;
```

Voici le script complet

ParseurXMLSimple
```
#! /usr/bin/perl
use strict;
use warnings;

use XML::Simple;

my $parser = XML::Simple->new( KeepRoot => 1 );

# Création du fichier résultat
my $FichierResulat = 'resultat.txt';
```

ParseurXMLSimple

```
open( my $FhResultat, '>', $FichierResulat )
  or die("Impossible d'ouvrir le fichier $FichierResulat\n$!");

my $doc = $parser->XMLin('ExempleXML.xml');

# Tout le fichier XML est dans $doc sous forme d'arbre
foreach my $personne ( @{ $doc->{annuaire}->{personne} } ) {
  print {$FhResultat} 'Personne : ';
  print {$FhResultat} $personne->{prenom};
  print {$FhResultat} "\t";
  print {$FhResultat} $personne->{miseajour};
  print {$FhResultat} "\nRole : ";
  print {$FhResultat} $personne->{'role'};
  print {$FhResultat} "\nEmail : ";
  print {$FhResultat} $personne->{email};
  print {$FhResultat} "\nTelephone : ";
  print {$FhResultat} str_if( $personne->{'telephone'}, '' );

  if ( $personne->{langage} ) {
    print {$FhResultat} "\nlangage preferes : ";
    langage_process( $personne->{langage} );
  }
  print {$FhResultat} "\nCommentaire : ";
  print {$FhResultat} $personne->{commentaire};
  print {$FhResultat} "\n", "=" x 10, "\n\n";
}

# Fermeture du fichier
close($FhResultat);

#==============
# Procédures
#==============
sub is_array {
  my ($var) = @_;
  return ( ref($var) eq 'ARRAY' );
}

sub str_if {
  my ( $str, $default ) = @_;
  return $default unless $str;
  return $str;
}

sub langage_process {
  my ($langage) = @_;
  print {$FhResultat} str_if( $langage->{name}, '' );
  return unless $langage->{ModulePreferes};
  if ( is_array( $langage->{ModulePreferes} ) ) {
    foreach my $module ( @{ $langage->{ModulePreferes} } ) {
      module_process($module);
    }
  }
  else {
    module_process( $langage->{ModulePreferes} );
  }
}

sub module_process {
  my ($module) = @_;
  print {$FhResultat} "\n\t- $module";
}
```

En lisant ce script, vous pouvez remarquer qu'il suffit de parcourir le hash pour lire les informations. C'est tout ! Nous vous recommandons ce module pour les fichiers XML très simples, sans attribut.

```
<balise nom="attribut">text</balise>
```

Dans le cas contraire, vous pourrez avoir de mauvaises surprises dans l'architecture du hash. Par exemple, supposons le fichier suivant :

```
<data>
  <toto name="att1">data1</toto>
  <toto name="att2">data2</toto>
</data>
```

XML::Simple aura pour contenu de hash ceci :

```
$VAR1 = {
          'toto' => {
                      'att1' => {
                                  'content' => 'data1'
                                },
                      'att2' => {
                                  'content' => 'data2'
                                }
                    }
        };
```

Si votre fichier ressemble à ceci

```
<data>
  <toto name="att1">data1</toto>
</data>
```

Vous obtiendrez

```
$VAR1 = {
          'toto' => {
                      'content' => 'data1',
                      'name' => 'att1'
                    }
        };
```

Vous constatez que le rendu est différent. Donc faites attention.
Si pour une raison quelconque vous souhaitez créer un fichier XML, vous pouvez utiliser la méthode **XMLout** du module. Voici un exemple de code

```
#! /usr/bin/perl
use strict;
use warnings;

use XML::Simple;

my $parser = XML::Simple->new( NoAttr=>1, RootName=>'data' );
my $doc = $parser->XMLout(
  {
  toto => {
    tutu => 1
  },
  'country'=>'Angleterre',
  'capital'=>'Londres',
  }
);
print $doc;
```

résultat en console

```
<data>
  <capital>Londres</capital>
  <country>Angleterre</country>
  <toto>
    <tutu>1</tutu>
  </toto>
```

résultat en console
```
</data>
```

Voilà, pour en savoir plus sur ce module, CPAN est votre ami !
N.B. Ce module à ma connaissance est incapable de faire une vérification DTD ou XSD Donc, il ne se préoccupe pas de l'information fournie dans le fichier XML.

III-2-b - XML::LibXML

Ce module est vraiment le plus puissant, le plus rapide et le plus complet. Il vous permet de vérifier votre fichier XML en fonction de la DTD renseignée. Il est également possible de faire une vérification en fonction d'un fichier XSD. Vous le remarquerez dans le script d'exemple. Voici le script d'exemple

```
#!/usr/bin/perl
use strict;
use warnings;

use XML::LibXML;

my $FichierXML = 'ExempleXML.xml';
my $parser     = XML::LibXML->new();

# Activation validation DTD du fichier XML avant le parsing
$parser->validation(1);

my $tree = $parser->parse_file($FichierXML);

# Création du fichier résultat
my $FichierResulat = 'resultat.txt';
open( my $FhResultat, '>', $FichierResulat )
  or die("Impossible d'ouvrir le fichier $FichierResulat\n$!");

# Racine du document XML
my $root = $tree->getDocumentElement;

# Validation XSD du fichier XML
# Récupérons le fichier XSD dans la balise annuaire
my $FichierXSD = $root->getAttribute('xsi:noNamespaceSchemaLocation');
my $schema = XML::LibXML::Schema->new( location => $FichierXSD );
eval { $schema->validate($tree) };
die "[XSD] Le fichier $FichierXML est non valide.\n$@" if $@;

# Balise personne
my @personne = $root->getElementsByTagName('personne');

foreach my $childid (@personne) {

  print {$FhResultat} "Personne : ",
    $childid->getElementsByTagName('prenom')->[0]->getFirstChild->getData, " ",
    $childid->getAttribute('miseajour'), "\n";

  print {$FhResultat} "Role : ",
    $childid->getElementsByTagName('role')->[0]->getFirstChild->getData, "\n";

  print {$FhResultat} "Email : ",
    $childid->getElementsByTagName('email')->[0]->getFirstChild->getData, "\n";

  foreach my $telephone ( $childid->getElementsByTagName('telephone') ) {
    print {$FhResultat} "Telephone : ", $telephone->getFirstChild->getData,
      "\n";
  }

  foreach my $langage ( $childid->getElementsByTagName('langage') ) {
    print {$FhResultat} "Langage prefere : ", $langage->getAttribute('name'),
      "\n";
  }

  foreach my $modules ( $childid->getElementsByTagName('ModulePreferes') ) {
```

```
    print {$FhResultat} "\t- ", $modules->getFirstChild->getData, "\n";
  }

  # commentaire
  print {$FhResultat} "Commentaire : ",
    $childid->getElementsByTagName('commentaire')->[0]->getFirstChild->getData,
    "\n";

  print {$FhResultat} "\n", "=" x 10, "\n\n";
}

close($FhResultat);
```

Le code est assez commenté pour être compris ! Vous remarquerez juste qu'on a activé la validation du fichier DTD en une ligne.

```
$parser->validation(1);
```

Il est important de mettre cette ligne avant le parsing du XML

```
my $tree = $parser->parse_file($FichierXML);
```

Si vous la mettez après, la vérification ne sera pas faite !
En ce qui concerne la vérification XSD de notre fichier XML, le module le fait très bien également. Il nous suffit de récupérer le nom du fichier XSD dans le fichier XML, puis de faire une vérification. C'est très simple :

```
# Validation XSD du fichier XML
# Récupérons le fichier XSD dans la balise annuaire
my $FichierXSD = $root->getAttribute('xsi:noNamespaceSchemaLocation');
my $schema = XML::LibXML::Schema->new( location => $FichierXSD );
eval { $schema->validate($tree) };
die "[XSD] Le fichier $FichierXML est non valide.\n$@" if $@;
```

Comme cela a été dit, ce module peut même vous permettre de convertir rapidement votre fichier XML en un fichier XSL. Il vous suffit de voir la documentation du module **XML::LibXSLT**.

III-2-c - XML::Twig

C'est un module vraiment simple à utiliser. Il est très riche et permet de jouer avec le XML. Le seul inconvénient (qui n'en est pas un), c'est qu'il est non validant, c'est-à-dire qu'il est impossible via XML::Twig de valider votre fichier XML via un fichier DTD ou XSD. De toute façon, il vaut mieux le valider avant en utilisant un autre module avant même de commencer à le parser.
Voici les scripts complets utilisant **XML::Twig** en simulant DOM ou non.

ParseurXMLTwigDom.pl

```
#!/usr/bin/perl
use strict;
use warnings;
use XML::Twig;

my $FichierXML = 'ExempleXML.xml';

# Parsing façon DOM, tout en mémoire
my $twig = new XML::Twig;

# Création du fichier résultat
my $FichierResulat = 'resultat.txt';
open( my $FhResultat, '>', $FichierResulat )
  or die("Impossible d'ouvrir le fichier $FichierResulat\n$!");

# Création d'un objet twig
$twig->parsefile($FichierXML);
```

ParseurXMLTwigDom.pl

```
# racine du XML
my $root = $twig->root;

# Chaque personne
foreach my $TwigPersonne ( $root->children ) {

  # prenom et mise à jour
  print {$FhResultat} "Personne : ", $TwigPersonne->field('prenom'), ' ',
    $TwigPersonne->att('miseajour'), "\n";

  # role, email, téléphone, etc.
  print {$FhResultat} "Role : ",      $TwigPersonne->field('role'),      "\n";
  print {$FhResultat} "Email : ",     $TwigPersonne->field('email'),     "\n";
  if ( my $Telephone = $TwigPersonne->field('telephone') ) {
    print {$FhResultat} "Telephone : $Telephone\n";
  }

  if ( my $TwigLangage = $TwigPersonne->first_child('langage') ) {
    print {$FhResultat} "Langage prefere : ", $TwigLangage->att('name'), "\n";

    # Module preferes
    foreach my $TwigModule ( $TwigLangage->children('ModulePreferes') ) {
      print {$FhResultat} "\t- ", $TwigModule->text, "\n";
    }
  }

  # commentaire
  print {$FhResultat} "Commentaire : ",
    $TwigPersonne->field('commentaire'), "\n";

  print {$FhResultat} "\n", "=" x 10, "\n\n";
}

close($FhResultat);
```

ParseurXMLTwig.pl - A la twig intelligent

```
#!/usr/bin/perl
use strict;
use warnings;
use XML::Twig;

my $FichierXML = 'ExempleXML.xml';

# Parsing façon intelligente en ne chargeant que le strict nécessaire en mémoire
# Utilisation des handlers
my $twig = new XML::Twig( Twig_handlers => { 'personne' => \&personne, }, );

# Création du fichier résultat
my $FichierResulat = 'resultat.txt';
open( my $FhResultat, '>', $FichierResulat )
  or die("Impossible d'ouvrir le fichier $FichierResulat\n$!");

# Création d'un objet twig
$twig->parsefile($FichierXML);

# Fermeture du fichier
close($FhResultat);

sub personne {
  my ( $twig, $TwigPersonne ) = @_;

  # prenom et mise à jour
  print {$FhResultat} "Personne : ", $TwigPersonne->field('prenom'), ' ',
    $TwigPersonne->att('miseajour'), "\n";

  # role, email, telephone, etc.
  print {$FhResultat} "Role : ",      $TwigPersonne->field('role'),      "\n";
  print {$FhResultat} "Email : ",     $TwigPersonne->field('email'),     "\n";
  if ( my $Telephone = $TwigPersonne->field('telephone') ) {
```

ParseurXMLTwig.pl - A la twig intelligent

```
    print {$FhResultat} "Telephone : $Telephone\n";
  }

  foreach my $TwigLangage ( $TwigPersonne->children('langage') ) {
    print {$FhResultat} "Langage prefere : ", $TwigLangage->att('name'), "\n";

    # Module preferes
    foreach my $TwigModule ( $TwigLangage->children('ModulePreferes') ) {
      print {$FhResultat} "\t- ", $TwigModule->text, "\n";
    }
  }

  # commentaire
  print {$FhResultat} "Commentaire : ",
    $TwigPersonne->field('commentaire'), "\n";

  print {$FhResultat} "\n", "=" x 10, "\n\n";

  # vide de la mémoire le contenu de la balise personne
  $twig->purge;
  return;
}
```

Comme vous pouvez le constater, nous avons fait deux scripts différents. Le premier charge tout le fichier en mémoire. Vous n'avez ensuite plus qu'à travailler sur l'objet twig (mais de façon beaucoup plus propre qu'avec **XML::Simple**). Tout est correctement mis à votre disposition de façon simple et logique.
Dans le script 2 (ParseurXMLTwig.pl), nous utilisons une petite merveille de Twig, **Twig_handlers**. Avec les Twig_handlers, il est possible de dire à Twig, d'effectuer certaines tâches lorsqu'il se trouve dans certaines balises. C'est le cas dans notre exemple. Il parse le contenu des balises **"personne"** que l'on supprime de la mémoire grâce à la méthode **purge**. Il y a une multitude de méthodes pour indenter du code, stopper le parsing, etc. Consultez la documentation du CPAN et surtout le site du créateur du module. Vous y trouverez des exemples de scripts et des explications.

IV - Créer des fichiers XML en Perl

Pour créer des fichiers XML, il existe un module très simple à prendre en main : **XML::Writer**. Il est simple à installer et à utiliser. Notez qu'il est aussi possible de créer des fichiers XML à l'aide des modules de parsing recommandés ci-dessus (XML::Twig, XML::LibXML...).

IV-1 - XML::Writer

Voici un exemple de script.

XMLWriter.pl

```
#!/usr/bin/perl
use XML::Writer;
use IO::File;

my $output = new IO::File(">output.xml");

my $writer = new XML::Writer(
    OUTPUT      => $output,
    DATA_INDENT => 3,             # indentation, trois espaces
    DATA_MODE   => 1,             # changement ligne.
    ENCODING    => 'utf-8',
);


$writer->xmlDecl("UTF-8");
$writer->startTag("data");
$writer->startTag( "greeting", "class" => "simple" );
$writer->characters("Hello, world!");
$writer->endTag("greeting");
```

XMLWriter.pl

```
$writer->startTag( "toto", "name" => "att1" );
$writer->characters("data1");
$writer->endTag("toto");
$writer->comment("Un commentaire");
$writer->emptyTag( 'toto', 'name' => 'att2.jpg' );

$writer->endTag("data");
$writer->end();
$output->close();
```

output.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<data>
   <greeting class="simple">Hello, world!</greeting>
   <toto name="att1">data1</toto>
   <!-- Un commentaire -->
   <toto name="att2.jpg" />
</data>
```

Voilà, ce n'est pas compliqué !

V - Valider ses fichiers XML

Pour valider ces fichiers XML, certaines personnes vous diront qu'il faut utiliser un parseur XML. En effet, tout bon parseur XML vérifiera que votre fichier respecte la syntaxe XML. Mais, il ne sera pas obligé de valider la conformité avec sa/ses DTD et/ou à son/ses XSD (s'il y en a ). On parlera dans ce cas de parseurs non validants. Quoi qu'il en soit, il existe des modules vous permettant de vérifier qu'un fichier XML est conforme à une **DTD**, un **XSD** ou même **Schematron**.

V-1 - Valider une DTD

Les modules les plus utilisés sont **XML::Checker::Parser** et **XML::LibXML** déjà évoqués ci-dessus. Voici un exemple de script utilisant XML::Checker::Parser.

```
#!/usr/bin/perl
use XML::Checker::Parser;

$XML::Checker::FAIL = \&my_fail;
my $FichierXML = 'ExempleXML.xml';
my $xp         = new XML::Checker::Parser();

$xp->parsefile($FichierXML);

sub my_fail {
    my $code = shift;
    die("[WARNING] Le fichier $FichierXML est non valide\n\t@_\n");
}
```

Vous pouvez créer une procédure avec ce code afin de valider à la volée vos fichiers XML.

V-2 - Valider une XSD

Les modules les plus utilisés sont **XML::Validator::Schema** et **XML::LibXML** déjà évoqués ci-dessus. Voici un exemple de script utilisant XML::Validator::Schema.

```
#!/usr/bin/perl
use XML::SAX::ParserFactory;
```

```
use XML::Validator::Schema;

my $FichierXSD = 'ExempleXML.xsd';
my $document   = 'ExempleXML.xml';

my $validator = XML::Validator::Schema->new( file => $FichierXSD );
my $parser = XML::SAX::ParserFactory->parser( Handler => $validator );

eval { $parser->parse_uri($document); };
die $@ if $@;

print "Le fichier $document est valide\n";
```

Vous pouvez créer une procédure avec ce code afin de valider à la volée vos fichiers XML.

VI - Conclusion

Voilà, c'est déjà fini !! Nous espérons que cet article vous a permis de constater qu'il est simple de dompter les fichiers XML en Perl. Vous avez sans doute remarqué qu'il existe beaucoup de modules faisant bien leur travail et que parmi eux, certains le font remarquablement bien (XML::LibXML et XML::Twig). Ne boudez pas votre plaisir en les utilisant !

Pour télécharger les fichiers (scripts, XML...) de cet article, rendez-vous **ici**.
N'hésitez pas à faire des remarques, corrections ou appréciations ici (), elles seront prises en compte et permettront d'améliorer cet article.

VII - Références utilisées

1 **XML, DTD et XSD**
- 1 **Introduction XML**
- 2 **http://www.xml.com**
- 3 **Notion DTD**
- 4 **DTD Tutorial from W3schools**
- 5 **XML Schema**
- 6 **XML Schemas**
- 7 **Les cours XML de developpez.com**

2 **PERL**
- 1 **La FAQ Perl XML officielle**
- 2 **Perl fondation**
- 3 **XML::Twig (CPAN)**, **Site du créateur**
- 4 **XML::LibXML**
- 5 **XML::Simple**

VIII - Remerciements

Nous remercions **GrandFather**, **stoyak**, **ClaudeLELOUP** et l'équipe de developpez.com pour la relecture et les remarques.