XML

**Table des matières**

[1 Introduction 2](#_Toc481447009)

[2 Structure d’un document xml 3](#_Toc481447010)

[3 Les espaces de noms 4](#_Toc481447011)

[4 Traitement des fichiers xml en .net 5](#_Toc481447012)

[4.1 XmlSerializer 5](#_Toc481447013)

[4.2 LINQ To XML 8](#_Toc481447014)

# Introduction

XML (eXtensible Markup Language) est un langage de description de données standardisé par le W3C

Il présente les données au format texte sous forme d’une arborescence de balises. On crée nous-même toutes les balises qu’on souhaite. Il n’y a pas de balises prédéfinies.

Il est lisible à la fois par une machine et un humain.

**Utilisations**

Les données étant représentées au format texte, XML peut être utilisé de façon universelle pour l’échange de données, notamment avec les protocoles de communication du web (http).

Il est utilisé pour décrire des informations de tous types, pourvus qu’elles puissent être représentées au format texte :

* Fichiers de configuration
* Descriptions de schémas (ex : UML)
* Description d’interfaces visuelles (ex : html, xaml, svg…)
* Descriptions de documents (ex : formats ooxml et odf)
* …

**Inconvénient**

Le principal inconvénient du langage xml est qu’il est assez verbeux. Json, un langage équivalent de xml qui s’est beaucoup développé ces dernières années avec l’utilisation massive de JavaScript, est par exemple moins verbeux, mais aussi moins facile à lire.

**Traitement**

Un document XML doit respecter la syntaxe XML (on dit alors qu’il est **valide**), pour être traitable par un **processeur XML**.

Un processeur XML est un module logiciel qui lit le fichier XML et permet de manipuler son contenu. Il existe plusieurs types de processeurs xml, comme nous allons le voir dans ce document.

# Structure d’un document xml

Exemple de fichier xml :

<?xml version="1.0" encoding="utf-8"?>

<!-- Elément racine (généralement une collection) -->

<CollectionBD Nom="Les aventures de Tintin">

<Auteurs>

<!-- Elément simple -->

<Auteur>Hergé</Auteur>

</Auteurs>

<!-- Elément de type collection -->

<Genres>

<Genre>Aventure</Genre>

<Genre>Policier</Genre>

<Genre>Science-fiction</Genre>

</Genres>

<Albums>

<!-- Eléments avec attributs, et balise auto-fermante -->

<Album Id="1" Titre="Tintin au pays des Soviets" Année="1930" />

<Album Id="2" Titre="Tintin au Congo" Année="1931" />

<Album Id="3" Titre="Tintin en Amérique" Année="1932" />

<Album Id="4" Titre="Les Cigares du pharaon" Année="1934" />

<Album Id="5" Titre="Le Lotus bleu" Année="1936" />

<Album Id="6" Titre="L'Oreille cassée" Année="1937" />

<Album Id="7" Titre="L'Ile Noire" Année="1938" />

<Album Id="8" Titre="Le Sceptre d'Ottokar" Année="1939" />

<Album Id="9" Titre="Le Crabe aux pinces d'or" Année="1941" />

<Album Id="10" Titre="L'Étoile mystérieuse" Année="1942" />

<Album Id="11" Titre="Le Secret de La Licorne" Année="1943" />

<Album Id="12" Titre="Le Trésor de Rackham le Rouge" Année="1944" />

<Album Id="13" Titre="Les Sept Boules de cristal" Année="1948" />

<Album Id="14" Titre="Le Temple du Soleil" Année="1949" />

<Album Id="15" Titre="Tintin au pays de l'or noir" Année="1950" />

<Album Id="16" Titre="Objectif Lune" Année="1953" />

<Album Id="17" Titre="On a marché sur la Lune" Année="1954" />

<Album Id="18" Titre="L'Affaire Tournesol" Année="1956" />

<Album Id="19" Titre="Coke en stock" Année="1958" />

<Album Id="20" Titre="Tintin au Tibet" Année="1960" />

<Album Id="21" Titre="Les Bijoux de la Castafiore" Année="1963" />

<Album Id="22" Titre="Vol 714 pour Sydney" Année="1968" />

<Album Id="23" Titre="Tintin et les Picaros" Année="1976" />

<Album Id="24" Titre="Tintin et l'Alph-Art" Année="1986" />

</Albums>

</CollectionBD>

Un fichier XML est composé de 2 parties :

* Le prologue, qui peut contenir lui-même :
  + La déclaration XML (version xml et encoding utilisé)
  + Des commentaires
  + Des instructions de traitement pour le processeur XML
* L’élément racine (ou document)

Les règles de syntaxe du xml sont décrites en détail sur la page suivante : <https://www.w3schools.com/xml/xml_syntax.asp>

# Les espaces de noms

Comme dans les autres langages, les espaces de noms permettent en xml d’éviter les conflits de noms. Il est ainsi possible d’avoir des noms de balises et d’attributs identiques, pour peu qu’ils figurent dans des espaces de noms différents.

Un espace de noms est déclaré avec un attribut du type :

xmlns:xx

…où xx représente le nom de l’espace de noms.

Voici un exemple de contenu d’un document au format ooxml :

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

<w:document xmlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessingCanvas"

xmlns:cx="http://schemas.microsoft.com/office/drawing/2014/chartex"

xmlns:cx1="http://schemas.microsoft.com/office/drawing/2015/9/8/chartex"

xmlns:o="urn:schemas-microsoft-com:office:office"

xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"

xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math"

xmlns:v="urn:schemas-microsoft-com:vml"  
...

<w:body>

<w:p w:rsidR="00B10DAA" w:rsidRDefault="00B10DAA">

<w:r>

<w:t>Voici un exemple de document Word</w:t>

</w:r>

<w:bookmarkStart w:id="0" w:name="\_GoBack"/>

<w:bookmarkEnd w:id="0"/>

</w:p>

<w:sectPr w:rsidR="00B10DAA">

<w:pgSz w:w="11906" w:h="16838"/>

<w:pgMar w:top="1417" w:right="1417" w:bottom="1417" w:left="1417"

w:header="708" w:footer="708" w:gutter="0"/>

<w:cols w:space="708"/>

<w:docGrid w:linePitch="360"/>

</w:sectPr>

</w:body>

</w:document>

Pour utiliser un élément d’un espace de noms, on met le préfixe d’espace de noms au début de sa balise.

Il peut y avoir un espace de noms sans préfixe, qui est l’espace de noms par défaut. Ex :

<?xml version="1.0" encoding="utf-8"?>

<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" Condition="Exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props')" />

# Traitement des fichiers xml en .net

**La sérialisation** consiste à générer un fichier xml à partir d’une arborescence d’objets .net

**La désérialisation** est l’opération inverse : elle consiste à charger une arborescence d’objets .net à partir des données lues dans un fichier xml.

.Net fournit plusieurs ensembles de classes pour lire, écrire et parcourir du xml :

* La classe XmlSerializer et l'utilisation de classes décorées d'attributs pour sérialiser et désérialiser un flux xml
* Les classes XmlReader et XmlWriter pour lire et écrire des fichiers xml avec le maximum de contrôle sur les données, pour les transformer comme on souhaite.
* La classe XmlDataProvider pour charger de façon simple un fichier XML et afficher son contenu dans un ItemsControl WPF
* L’API LINQ To XML, pour charger un flux xml en mémoire, et effectuer des opérations de recherche et/ou de modifications avec un syntaxe Linq.
* …

Il convient d’utiliser les bonnes classes selon l’objectif visé et les contraintes auxquelles on est confronté. Exemple de scénarios classiques :

* Pour charger un fichier xml complet dans une collection d’objets et l’enregistrer en base de données, on pourra utiliser le XmlSerializer
* Si la structure du fichier est très différente de celle des classes dans lesquelles on veut le charger, l’utilisation du XmlReader sera dans ce cas nécessaire.
* Pour rechercher et/ou modifier une information particulière dans un fichier de configuration, on pourra utiliser LINQ to XML
* …

Nous allons voir maintenant plus en détail la classe XmlSerializer, ainsi qu’un exemple d’utilisation de LINQ to XML.

## XmlSerializer

Cette classe permet de sérialiser et désérialiser du xml de façon simple et très concise. Elle s’appuie sur des attributs qui décorent les classes, et qui permettent de préciser les règles de (dé)sérialisation.

Voyons comment l’utiliser pour traiter le fichier xml suivant :

<?xml version="1.0" encoding="utf-8"?>

<CollectionsBD>

<CollectionBD Nom="Les aventures de Tintin">

<Auteurs>

<Auteur>Hergé</Auteur>

</Auteurs>

<Genres>

<Genre>Aventure</Genre>

<Genre>Policier</Genre>

<Genre>Science-fiction</Genre>

</Genres>

<Albums>

<Album Id="1" Titre="Tintin au pays des Soviets" Année="1930" />

<Album Id="2" Titre="Tintin au Congo" Année="1931" />

<Album Id="3" Titre="Tintin en Amérique" Année="1932" />

</Albums>

</CollectionBD>

<CollectionBD Nom="Astérix">

<Auteurs>

<Auteur>René Goscinny</Auteur>

<Auteur>Albert Uderzo</Auteur>

<Auteur>Jean-Yves Ferri</Auteur>

<Auteur>Didier Conrad</Auteur>

</Auteurs>

<Genres>

<Genre>Aventure</Genre>

<Genre>Humour</Genre>

</Genres>

<Albums>

<Album Id="1" Titre="Astérix le Gaulois" Année="1961" />

<Album Id="2" Titre="La Serpe d'or" Année="1962" />

<Album Id="3" Titre="Astérix et les Goths" Année="1963" />

</Albums>

</CollectionBD>

</CollectionsBD>

Nous souhaitons charger les données dans une liste d’objets décrits par la classe suivante :

public class CollectionBD

{

   public string Nom { get; set; }

   public List<Auteur> Auteurs { get; set; }

   public List<Genre> Genres { get; set; }

   public List<Album> Albums { get; set; }

}

Remarquons que dans le fichier xml, certaines informations sont présentées sous forme d’éléments et d’autres sous forme d’attributs. Ceci doit être retranscrit dans les attributs qui décorent les classes de la façon suivante :

public class CollectionBD

{

   // Par défaut les propriétés sont sérialisées en éléments xml

   // Pour les sérialiser en attributs, il faut les décorer de [XmlAttribute]

   [XmlAttribute]

   public string Nom { get; set; }

   public List<Auteur> Auteurs { get; set; }

   public List<Genre> Genres { get; set; }

   public List<Album> Albums { get; set; }

}

public class Auteur

{

   // XMLText désigne le texte à l'intérieur d'un élément

   [XmlText]

   public string Nom { get; set; }

}

public class Genre

{

   [XmlText]

   public string Libelle { get; set; }

}

public class Album

{

   [XmlAttribute]

   public int Id { get; set; }

   [XmlAttribute]

   public string Titre { get; set; }

   [XmlAttribute]

   public int Année { get; set; }

}

Il y a ici une correspondance parfaite entre les noms d’éléments et attributs du fichier xml, et les noms des propriétés des classes C#. Si on voulait utiliser des noms différents, il suffirait par exemple de préciser ces noms de la façon suivante :

[XmlRoot("ComicBook")]

public class Album

{

   public int Id { get; set; }

   [XmlAttribute("Title")]

   public string Titre { get; set; }

   [XmlAttribute("Year")]

   public int Année { get; set; }

}

Les liens entre les éléments et attributs xml, et les classes et propriétés C# étant décrits, le chargement des données est ensuite très simple :

public static List<CollectionBD> ImporterXml()

{

   List<CollectionBD> listCol = null;

   XmlSerializer deserializer = new XmlSerializer(typeof(List<CollectionBD>),

      new XmlRootAttribute("CollectionsBD"));

using (var sr = new StreamReader(@"..\..\CollectionsBD.xml"))

   {

      // Deserialize renvoie un type object, qu'il faut transtyper

      listCol = (List<CollectionBD>)deserializer.Deserialize(sr);

   }

   return listCol;

}

On précise dans le constructeur du XmlSerializer :

* Le type de l’objet qui représente l’élément racine du document xml (ici, une liste de CollectionBD)
* Le nom de cet élément racine

La désérialisation est faite en appelant la méthode Deserialize, qui utilise un objet StreamReader pour lire le fichier. Le résultat est retourné sous forme d’object, qu’il suffit de transtyper.

La sérialisation est très similaire :

public static void ExporterXml(List<CollectionBD> listCol)

{

   // On crée un sérialiseur, en spécifiant le type de l'objet à sérialiser

   // et le nom de l'élément xml racine

   XmlSerializer serializer = new XmlSerializer(typeof(List<CollectionBD>),

                              new XmlRootAttribute("CollectionsBD"));

   using (var sw = new StreamWriter(@"..\..\CollectionsBD.xml"))

   {

      serializer.Serialize(sw, listCol);

   }

}

On appelle cette fois la méthode Serialize du XmlSerializer, en lui passant un StreamWriter pour écrire le fichier.

## LINQ To XML

Cette API remplace avantageusement les API DOM et XPath, qui sont les API de base pour du code non managé (c’est-à-dire avec du code n’utilisant pas .net, tel que C++, JavaScript…).

LIQ To XML fournit un moyen puissant de manipuler un flux xml pour en extraire des informations, ou les modifier, en s’appuyant sur la syntaxe LINQ.

Le fichier xml est entièrement chargé en mémoire ; Il peut être modifié et enregistré au moyen de la méthode XDocument.Save.

### Requêtes LINQ

Voici un exemple d’extraction et de modification d’information sur le fichier xml présenté plus haut :

public static void ExemplesLINQtoXML()

{  
 // Chargement du fichier xml   
 XDocument doc = XDocument.Load(@"..\..\CollectionsBD.xml");

   // Récupération de tous les titres d'albums

   var titres = doc.Descendants("Album").Attributes("Titre").Select(t => t.Value);

   // Récupération de l'album dont le titre est "Le Lotus bleu"

   var al = doc.Descendants("Album").Where(a => a.Attribute("Titre").Value == "Le Lotus bleu").FirstOrDefault();

   //...puis modification de son année de sortie

   al.Attribute("Année").Value = "1937";

   //... et enregistrement du fichier

   doc.Save(@"..\..\CollectionsBD.xml");

}

Pour avoir une liste exhaustive de toutes les techniques et syntaxes pour l’extraction d’informations depuis un flux xml avec LINQ, on se référera à la page MSDN suivante : <https://msdn.microsoft.com/en-us/library/mt693117.aspx>

### Sérialisation d’une collection avec LINQ

Bien que LINQ To XML ne soit pas la technique la plus simple pour sérialiser une collection d’objets arborescente, voici un exemple de code qui permet de le faire. L’objectif étant surtout d’illustrer la syntaxe d’écriture de flux xml avec LINQ.

public static void ExporterXmlLINQ(List<CollectionBD> listCol)

{

   XDocument doc = new XDocument(SerialiserCollections(listCol));

   using (var sw = new StreamWriter(@"..\..\CollectionsBDLINQ.xml"))

   {

      doc.Save(sw);

   }

}

private static XElement SerialiserCollections(List<CollectionBD> listCollections)

{

   XElement elt = new XElement("CollectionsBD");

   foreach (var c in listCollections)

      elt.Add(new XElement("CollectionBD", new XAttribute("Nom", c.Nom),

                  SerialiserAuteurs(c.Auteurs),

                  SerialiserGenres(c.Genres),

                  SerialiserAlbums(c.Albums)));

   return elt;

}

private static XElement SerialiserAuteurs(List<Auteur> listAuteurs)

{

   XElement elt = new XElement("Auteurs");

   foreach (var a in listAuteurs)

      elt.Add(new XElement("Auteur", a.Nom));

   return elt;

}

private static XElement SerialiserGenres(List<Genre> listGenres)

{

   XElement elt = new XElement("Genres");

   foreach (var g in listGenres)

      elt.Add(new XElement("Genre", g.Libelle));

   return elt;

}

private static XElement SerialiserAlbums(List<Album> listAlbums)

{

   XElement elt = new XElement("Albums");

   foreach (var a in listAlbums)

      elt.Add(new XElement("Album",

            new XAttribute("Id", a.Id),

            new XAttribute("Titre", a.Titre),

            new XAttribute("Année", a.Année)));

   return elt;

}

Ce code produit exactement le même résultat que le XmlSerializer vu plus haut. Il illustre la manière dont on peut imbriquer les objets de façon à créer l’arborescence du flux XML souhaité.