Devenir un meilleur codeur avec C#et XML

Introduction

Paris, le 8 aout 2012

« Ah martin tant que je te tiens, le client m'a fait un retour très négatif sur vos développements sur l'appli MachinTruc. Le logiciel prend une éternité pour charger le fichier XML, et pour l'écriture c'est encore pire ! Peux-tu corriger ça vite s'il te plait ? »

Ce qui vient d'arriver à Martin peut arriver à n'importe quel autre développeur C#. On lui a confié une tache assez simple, à savoir, lire et afficher des données provenant d'un fichier XML. Le logiciel doit aussi autoriser la modification de ces données via cet affichage.

Ce n'est pas une tâche très complexe et Martin l'a bien entendu testé avec de petits fichiers. Malheureusement, même si le code qu'il a implémenté fonctionne, il est inutilisable en tant que tel car la méthode qu'il a utilisée n'est pas adaptée pour travailler avec des fichiers d'au moins 1 Mo.

Le XML (*eXtensible Markup Language*) est un langage de transport et de stockage de données devenu incontournable sur le Web et de manière générale dans le domaine du logiciel.

Cependant il y a très peu de ressources qui expliquent comment travailler avec des fichiers XML en C#. La plupart des livres (même les meilleurs) n'en parlent tout simplement pas. Parfois, il y a un petit chapitre sur le XML ; en général c'est le 26ème ou le 28ème du livre que peu de développeurs lisent

et il ne contient qu'une ou deux des techniques pour travailler avec du XML, souvent XmlDocument ou XmlTextReader , qui sont devenues largement obsolètes !

C'est pourtant loin d'être la meilleure méthode pour travailler efficacement avec du XML. Mais alors

Comment lire un fichier XML rapidement, simplement avec du code maintenable ?

De même, quelles sont les best practices pour écrire un fichier XML?

Quelle solution choisir en fonction de la taille des fichiers à traiter pour avoir les meilleures performances ?

Comment extraire une partie des données de ce fichier ?

C'est à ces questions que cet ebook va répondre.

A l'issue de la lecture de cet ebook vous connaitrez les best practices pour traiter du XML avec du C# à l'heure actuelle (Framework 4.0).

A qui s'adresse ce livre ?

L'objectif de cet ouvrage n'est pas une description détaillée de toutes les manières de traiter des fichiers XML en .NET.

Le but est de donner

- Une vision synthétique des solutions disponibles
- Les clés pour utiliser ces solutions, au moins avec des exemples assez simples.
- Du code qui vous expliquera comment utiliser chacune de ces solutions
- Un comparatif des avantages inconvénients de ces solutions
- Leurs performances temporelles en fonction de la taille de fichier traité.
- Quelques outils pour booster sa productivité avec du XML.

Ce livre est donc pour tous les développeurs .NET, qui comme moi ont eu à gérer des fichiers XML et se sont noyés dans les solutions trouvées sur le web.

A l'issue de la lecture de ce livre vous pourrez briller au café devant vos collègues quand la discussion dérivera un jour, forcément, sur la manière de gérer du XML.

Ce que vous ne trouverez pas dans ce livre

- Des paragraphes sans fin décrivant les 15 000 champs et méthodes de chaque solution pour vous endormir le soir.
- La dernière discussion entre 2 pseudos starlettes du dernier show de téléréalité à la mode.

Fichier de travail

Tout au long de cet ouvrage nous allons travailler avec un même fichier XML :

```
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="1" Category="novel">
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
  </Book>
  <Book Id="2" Category="non-fiction">
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
  </Book>
  <Book Id="3" Category="novel">
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
  </Book>
</Library>
```

Comme vous pouvez le constater, il ne s'agit pas d'un fichier très complexe : 3 livres contenus dans une balise racine Library.

L'exemple n'est pas non plus ultra basique car le fichier contient une **collection** d'éléments sur **2 niveaux hiérarchiques** avec des **attributs**. Le traitement de ce fichier est nettement plus délicat que les exemples plus basiques que l'on trouve habituellement : il n'y a qu'un seul nœud ou seulement un seul niveau hiérarchique.

Telecharger le projet à partir du web

Tous les extraits de code sont centralisés dans un projet que vous pouvez télécharger à l'url suivante :

https://app.box.com/s/kqimm6wqhrehuhejz1z5

Pour exécuter le projet :

1) Dézippez l'archive
2) Cliquez sur le fichier XMLReadWriteDemo.sln (Visual studio 2010)
3) Allez dans le fichier **program.cs** et choisissez les opérations à réaliser en commentant ou décommentant les lignes de code d'appel aux fonctions
4) Executez le projet. Les fichiers générés se trouveront dans le répertoire bin/Debug ou bin/release suivante la configuration choisie.

Aucun livre ou ouvrage ne sera jamais parfait ou entièrement terminé. N'hésitez pas à m'envoyer vos remarques, suggestions à thomas.blotiere@gmail.com pour que je puisse continuer d'améliorer cet ebook !

XmlReader & XmlWriter

Introduction à la hiérarchie de classes du modèle DOM

Avant même de commencer à décrire les différents moyens de travailler avec des fichiers XML en .NET, il est important de discuter rapidement des namespaces et classes du modèle DOM.

Ce modèle est celui utilisé dans les versions du Framework jusqu'à la 3.5. A partir de la version 3.5 du Framework, LinqToXml s'est présentée comme une alternative à ce modèle.

Cependant, pour des raisons historiques, le modèle DOM est encore très utilisé dans les applications et il est important de comprendre son fonctionnement, du moins au niveau basique.

System.Xml

Ce namespace contient des classes majeures dans la lecture et l'écriture de fichiers XML, notamment les classes **XmlReader**, **XmlTextReader**, **XmlWriter** et **XmlTextWriter**.

Cela ne devrait pas être une grande surprise, mais **XmlReader** est une classe abstraite contenant des méthodes et des propriétés pour lire un document. La méthode *Read* lit un nœud provenant d'un stream. De plus la classe contient des méthodes pour naviguer dans l'arborescence du fichier XML (*MoveToAttribute*, *MoveToFirstAttribute*, *MoveToContent*, *MoveToFirstContent*, *MoveToElement* and *MoveToNextAttribute*)

XmlNode est une classe centrale dans la gestion du XML par .NET. Elle représente un seul nœud de l'arborescence. Elle est utilisée par de nombreuses autres classes pour insérer, supprimer des nœuds ou encore naviguer dans l'arborescence.

XmlDocument est dérivée de cette dernière. Cette classe représente un document XML et contient des méthodes de chargement (*Load* par exemple) et de sauvegarde (*Save*).

Lecture de documents avec XmlReader

Le plus ancien moyen de lire des documents XML est l'utilisation d'un **XmlReader.**

Dans l'exemple suivant on va lire un fichier contenant 3 livres et retourner une liste d'objets Book :

```
List<Book> listBooks = new List<Book>();
using (XmlReader xtr = XmlReader.Create("./SampleLibrary.xml"))
{
   Book book = new Book();
   //boucle sur chaque noeud
    while (xtr.Read())
    {
      switch (xtr.NodeType)
         {
         case XmlNodeType.Element: // le node est un element.
             while (xtr.MoveToNextAttribute()) // lire les attributs.
             {
                if (xtr.Name == "Id")
```

```
                        book.Id = xtr.Value;

                  if (xtr.Name == "Category")
                        book.Type = xtr.Value;
               }
               if (xtr.Name == "Name")
               {
                   //lit l'element Name et passe au noeud suivant
                    book.Name = xtr.ReadElementString();
               }
               if (xtr.Name == "Author")
                   book.Author = xtr.ReadElementString(); break;

          case XmlNodeType.EndElement:
          {
              if (xtr.Name == "Book")
              {// ajouter le livre quand la balise </book> est rencontrée
                   listBooks.Add(book);
                   book = new Book();
                   break;
              }
       }//Fin boucle
    }// Fin using
// utilisation de listBooks
```

La première chose à faire pour lire un fichier Xml de cette manière est donc de créer cet objet **XmlReader**,

```
        using (XmlReader xtr = XmlReader.Create ("./SampleLibrary.xml"))
         {
```

que l'on encadrera par le mot clé using afin de ne pas avoir à gérer la fermeture de cet objet.
La lecture des nœuds intervient lors de l'appel à la méthode **Read**().

```
        while (xtr.Read())
        {
               switch (xtr.NodeType)
               {
                      …
               }
        }
```

Le traitement des nœuds est différent suivant le type de ceux-ci, d'où l'utilisation de la propriété **NodeType**

Lorsqu'il s'agit d'un élément, on boucle pour récupérer les attributs s'il y en a.

```
while (xtr.MoveToNextAttribute()) // lire les attributs.
{
   if (xtr.Name == "Id")
        book.Id = xtr.Value;

  if (xtr.Name == "Category")
      book.Type = xtr.Value;
}
```

Ensuite, on s'occupe de lire la valeur du nœud :

```
if (xtr.Name == "Name")
```

```
      book.Name = xtr.ReadElementString();//lit l'element Name et passe au noeud
suivant
if (xtr.Name == "Author")
      book.Author = xtr.ReadElementString();//lit l'element Author et passe au noeud
suivant
```

La méthode **ReadElementString**() va récupérer le texte du nœud et passe au suivant.

Enfin, il faut ajouter l'objet book à la liste pour traitement ultérieur. Ceci est effectué lorsque la balise </Book> est rencontrée.

```
case XmlNodeType.EndElement:
{
   if (xtr.Name == "Book")
   {// ajouter le livre quand la balise </Book> est rencontrée
      listBooks.Add(book);
      book = new Book();
    }
 }
```

Comme vous pouvez le voir, lire du XML avec **XmlReader** est assez délicat et implique un grand niveau de couplage entre les détails du fichier XML et le modèle objet (ici la classe Book).

En 2 mots, la technique consiste à avancer nœud par nœud dans le fichier en appelant la méthode **Read**() et en extrayant les données via **ReadElementString**().

L'utilisation de **XmlReader** opère un très faible niveau d'abstraction et n'est envisageable que pour des fichiers simples.

Pourquoi ne pas utiliser XmlTextReader ?

XmlTextReader est une classe dérivant de **XmlReader.** Si vous recherchez des tutoriaux sur C# et XML sur le web vous allez tomber sur un grand nombre d'entre eux présentant des solutions à partir de **XmlTextReader** et **XmlTextWriter**.

Cependant **XmlTextReader** est aujourd'hui (08/2013) très largement obsolète et il n'est pas recommandé de l'utiliser selon Microsoft. De nombreux bugs y ont été détectés. Pour plus d'informations se reporter au lien ci-dessous :

http://blogs.msdn.com/b/xmlteam/archive/2011/10/08/the-world-has-moved-on-have-you-xml-apis-you-should-avoid-using.aspx

Si cette classe n'est pas déclarée comme obsolète c'est parce qu'elle fait partie de la norme ECMA-335 (Common Language Infrastructure).De plus résoudre ces bugs aurait cassé la compatibilité avec les versions précédentes, ce qui n'est pas souhaitable.

Ecrire un document XML avec XmlWriter

Le code complet pour écrire un fichier XML est ci-dessous :

```
try
{
    List<Book> listBooks = AddBooks();
    //settings indent sert à rajouter un retour à la ligne à la fin de chaque élément
```

```
    var settings = new XmlWriterSettings
    { Encoding = Encoding.UTF8, Indent = true, };
    using (XmlWriter writer = XmlWriter.Create(@"./Generated_XmlWriter.xml"
, settings))
    {
        writer.WriteStartDocument();
        writer.WriteStartElement("Library");
        foreach (Book b in listBooks)
        {
            writer.WriteStartElement("Book");
            writer.WriteAttributeString("Id", b.Id);
            writer.WriteAttributeString("Category", b.Category);
            writer.WriteElementString("Name", b.Name);
            writer.WriteElementString("Author", b.Author);
            writer.WriteEndElement();//</Book>
        }
        writer.WriteEndElement();//</Library>
        writer.WriteEndDocument();
    }
}
catch (Exception ex)
{
    _log.Error(ex.Message);
}
```

Les étapes pour écrire un document XML avec **XmlWriter** sont les suivantes :

```
var settings = new XmlWriterSettings { Encoding = Encoding.UTF8, Indent = true };
using (XmlWriter writer =
XmlWriter.Create(@"c:\temp\XmlWriterGeneratedSampleLibrary.xml",settings))
{
```

XmlWriter prend en argument du constructeur le chemin où le fichier sera généré. Le second argument est facultatif : il s'agit d'un objet **XmlWriterSettings** dans lequel on peut configurer certaines options telles l'encodage ou l'indentation (le xml généré doit il comporter des retours chariots ?).

Ensuite intervient l'écriture de l'élément racine, ici Library :

```
        writer.WriteStartDocument();
        writer.WriteStartElement("Library");
```

Après cette étape il ne reste plus qu'à écrire les autres éléments du fichier et clore la balise root et le document :

```
foreach (Book b in listBooks){

 writer.WriteStartElement("Book");
 writer.WriteAttributeString("Id", b.Id);
 writer.WriteAttributeString("Category", b.Category);
 writer.WriteElementString("Name", b.Name);
 writer.WriteElementString("Author", b.Author);
 writer.WriteEndElement();//</Book>
}
writer.WriteEndElement();//</Library>
writer.WriteEndDocument();
```

Le résultat est le suivant :

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="1" Category="novel">
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
  </Book>
  <Book Id="2" Category="non-fiction">
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
  </Book>
  <Book Id="3" Category="novel">
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
  </Book>
</Library>
```

Comme vous pouvez le constater pour écrire un fichier avec **XmlWriter**, on va principalement utiliser les méthodes :

WriteStartElement, **WriteEndElement** : ces méthodes permettent d'écrire les balises ouvrantes et fermantes

`Ex :<Book> et </Book>`

WriteElementString : Ecrit une balise ouvrante, du texte et une balise fermante

`Ex :<Name>the 4h work week</Name>`

WriteAttributeString : permet de rajouter un attribut à l'élément ouvert.

`Ex : <Book Id="2">`

Validation avec XmlReader/XmlWriter

Il peut être tentant de vouloir valider le fichier XML généré après écriture de celui-ci. Cela permet de détecter des erreurs grossières et d'éviter de passer pour un imbécile auprès du client qui reçoit un fichier malformé.

La validation d'un fichier XML avec **XmlWriter** n'est tout simplement pas possible directement. **XmlWriter** ne supporte pas de méthode pour faire cela.

La solution consiste à utiliser un **XmlReader** pour lire le fichier nouvellement créé et faire la validation à ce niveau.

Un développeur peut aussi vouloir valider le fichier XML en lecture. Pour cela il faut procéder de la manière suivante :

On va tout d'abord créer le schéma :

```
XmlSchemaSet sc = new XmlSchemaSet();
sc.Add("", "SampleLibraryXsd.xsd");
```

La classe `XmlSchemaSet` encapsule un fichier Xsd. Les Dtd ne sont pas supportées par cet objet.

Dans l'exemple ci-dessus, on ajoute le fichier XSD SampleLibraryXsd.xsd. Le premier argument est le nom du namespace. Cependant comme nous n'avons spécifié aucun namespace, on va laisser une string vide.

On créé ensuite un objet `XmlReaderSettings` :

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas = sc;
settings.ValidationEventHandler += new ValidationEventHandler(ValidationCallBack);
```

On choisit le type de validation, ici Schema. Les autres valeurs sont DTD et none. La propriété Schemas permet d'affecter notre XmlSchemaSet à l'objet settings.

Enfin on définit une callback qui va être appelée en cas d'erreur de validation dont le code est ci-dessous :

```
private static void ValidationCallBack(object sender, ValidationEventArgs e)
{
   Console.WriteLine("Validation Error: {0}", e.Message);
}
```

Si vous souhaitez valider le fichier avec une DTD alors il faut choisir les options suivantes :

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Parse;
settings.ValidationType = ValidationType.DTD;
settings.ValidationEventHandler += new ValidationEventHandler(ValidationCallBack);
```

L'objet settings est ensuite passé au **XmlReader** lors de sa création :

```
using (XmlReader xtr = XmlReader.Create("./SampleLibrary.xml", settings))
```

Attention car établir un mécanisme de validation et une callback ne veut pas dire que toutes les erreurs seront remontées dans la callback. Pour que la validation soit effective il faut que la lecture se soit terminée avec succès.
Certaines erreurs seront levées comme exception lors de la lecture. C'est pour cela qu'il est crucial d'entourer le code de lecture de `try-catch` et de gérer les erreurs à ce niveau également.

Prenons par exemple le fichier suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="3" Category="novel">
    <Info>
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
      <Author> test validation<Author>
    </Info>
  </Book>
</Library>
```

Il existe une balise <Author> en doublon qui n'a pas de balise fermante. Le code de lecture écrit dans la première partie de ce chapitre ne lira que la première balise <Author>, bien formée. Dans ce cas, la lecture du document se fera sans erreur et l'erreur de validation sera bien remontée dans la callback.

En revanche pour ce fichier :

```
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="3" Category="novel">
    <Info>
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry<Author>
    </Info>
  </Book>
</Library>
```

La balise <Author> n'est pas fermée. Une exception sera levée lors de la lecture de la balise et la lecture sera stoppée à ce point. On n'entrera donc pas dans la callback de validation

Conclusion

Comme vous pouvez le voir, lire du XML avec **XmlReader** est assez délicat et implique un grand niveau de couplage entre les détails du fichier XML et le modèle objet(ici la classe Book).

De même écrire un fichier XML avec **XmlWriter** est loin d'être une partie de plaisir ! L'exemple traité dans ce chapitre est un exemple simple, mais pour des fichiers XML complexes, l'écriture de ce code prend beaucoup de temps et produit un code peu maintenable.

En revanche, **XmlWriter** et **XmlReader** sont les classes les plus performants pour traiter du XML avec .NET. Si la rapidité d'exécution est primordiale pour votre application, ce sont ces 2 classes qu'il faut utiliser. La raison en est simple : toutes les autres techniques pour lire et écrire du XML utilisent **XmlWriter** et **XmlReader** !

(cf. benchmark à la fin de cet ouvrage)

XmlDocument

Lire un fichier Xml avec XmlDocument

L'objet **XmlDocument** du namespace **System.Xml** est basé sur la notion de parenté entre nœuds XML. Au lieu de parcourir séquentiellement le fichier, on sélectionne un groupe de nœuds avec **SelectNodes**() ou un seul Nœud avec **SelectSingleNode**(). Il est ensuite possible de naviguer en utilisant la propriété **ChildNodes** pour obtenir les nœuds enfants et **ParentNodes** pour les nœuds parents.

Extrait de Code 1 :

```
internal List<Book> ReadData2(string fullpath = "./SampleLibrary.xml")
        {
            List<Book> listBooks = new List<Book>();
            XmlDocument xd = new XmlDocument();
            xd.Load(fullpath);
            XmlNodeList nodelist = xd.SelectNodes("/Library/Book");
            foreach (XmlNode node in nodelist)
            {
                Book book = new Book();
                book.Id = node.Attributes.GetNamedItem("Id").Value;
                book.Category = node.Attributes.GetNamedItem("Category").Value;

                book.Name = node.SelectSingleNode("Name").InnerText;
                book.Author = node.SelectSingleNode("Author").InnerText;
                listBooks.Add(book);
            }
            return listBooks;
        }
```

On commence par initialiser un nouveau **XmlDocument** et charger le fichier à l'aide la méthode *Load* :

```
XmlDocument xd = new XmlDocument();
xd.Load(fullpath);
```

On sélectionne ensuite les nœuds qui nous intéressent, à savoir les nœuds <Book> :

```
XmlNodeList nodelist = xd.SelectNodes("/Library/Book");
```

Une fois la liste de nœuds récupérée, on va itérer sur chacun de ceux-ci :

```
foreach (XmlNode node in nodelist)
{
    Book book = new Book();
    book.Id = node.Attributes.GetNamedItem("Id").Value;
    book.Category = node.Attributes.GetNamedItem("Category").Value;

    book.Name = node.SelectSingleNode("Name").InnerText;
    book.Author = node.SelectSingleNode("Author").InnerText;
   listBooks.Add(book);
}
```

Les attributs ne sont pas des nœuds XML, ils sont récupérés via la méthode **Attributes.GetNamedItem** du nœud. Pour accéder à la balise info, fille de Book,

La méthode *SelectSingleNode* permet de récupérer les nœuds par Nom et donc de récupérer les valeurs des nœuds <Author> et <Name>.

Ce bout de code fonctionne bien mais on aurait pu s'y prendre autrement pour lire le fichier avec **XmlDocument** :

Extrait de Code 2 :

```
internal List<Book> ReadData(string fullpath = "./SampleLibrary.xml")
        {
            List<Book> listBooks = new List<Book>();
            XmlDocument xd = new XmlDocument();
            xd.Load(fullpath);

            XmlNodeList nodelist = xd.GetElementsByTagName("Book");// get all <book>
nodes
            XmlNodeList names = xd.GetElementsByTagName("Name");
            XmlNodeList authors = xd.GetElementsByTagName("Author");

            for (int i = 0; i < nodelist.Count;i++ ) // for each <testcase> node
            {
                Book book = new Book();
                book.Id = nodelist[i].Attributes.GetNamedItem("Id").Value;
                book.Category = nodelist[i].Attributes.GetNamedItem("Category").Value;
                book.Name = names[i].InnerText;
                book.Author = authors[i].InnerText;

                listBooks.Add(book);
            }
            return listBooks;
        }
```

Dans ce cas on se base sur la méthode *GetElementsByTagName* qui est une alternative pour accéder aux nœuds <Book>. On obtient une collection de nœud <Book>. Pour récupérer les nœuds <Author> et <Name> on procède de la même manière ce qui nous amène à 3 collections, une de <Book> , <Author> et <Name>. En parcourant ces 3 collections en même temps on peut alors récupérer toutes les informations nécessaires à la création de l'objet Book.

Comme écrit précédemment, vous êtes libre de choisir la manière de lire vous document avec XmlDocument, que ce soit via *SelectNodes* (Extrait de code 1) ou via *GetElementsByTagName* (extrait de code 2).

De manière générale, l'utilisation de la classe **XmlDocument** est un bon choix si vous souhaitez extraire des données de manière non séquentielle ou si vous utilisez déjà des objets **XmlDocument** ailleurs dans le code pour maintenir un peu d'homogénéité dans celui-ci.

Ecrire avec XmlDocument

Afin de créer un fichier avec **XmlDocument**, il est nécessaire de programmer à la main toute l'arborescence XML.

Ensuite l'appel à la méthode Save va créer physiquement le fichier XML.

Le fichier résultat souhaité est le suivant :

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

```
<Library>
  <Book ID="01" Category="novel">
    <Info>
      <Name>Comment je suis devenu stupide</Name>
      <Author>Martin Page</Author>
    </Info>
  </Book>
</Library>
```

On n'a pas utilisé dans ce cas, le fichier habituel de test avec nos 3 auteurs par souci de concision.

Il ne s'agit pas d'un gros fichier XML. Voyons maintenant le code nécessaire pour créer ce petit fichier :

```
XmlDocument xmlDoc = new XmlDocument();
// Ecriture de la  déclaration XML
XmlDeclaration xmlDeclaration = xmlDoc.CreateXmlDeclaration("1.0", "utf-8", null);

// Creation de element Root
XmlElement rootNode = xmlDoc.CreateElement("Library");
xmlDoc.InsertBefore(xmlDeclaration, xmlDoc.DocumentElement);
xmlDoc.AppendChild(rootNode);

// Création d'un nouveau <Book> élément et ajout du nœud
XmlElement parentNode = xmlDoc.CreateElement("Book");

// Ajout des attributs de Book et leur valeur.
parentNode.SetAttribute("ID", "01");
parentNode.SetAttribute("Category", "novel");
xmlDoc.DocumentElement.PrependChild(parentNode);

//creation du nœud info et ajout en tant qu'enfant de Book
XmlElement infoNode = xmlDoc.CreateElement("Info");
parentNode.AppendChild(infoNode);
//Creation des noeuds Name et Author
XmlElement nameNode = xmlDoc.CreateElement("Name");
XmlElement authorNode = xmlDoc.CreateElement("Author");

// recuperation du texte
XmlText nameText = xmlDoc.CreateTextNode("Comment je suis devenu stupide");
XmlText authorText = xmlDoc.CreateTextNode("Martin Page");

infoNode.AppendChild(nameNode);
infoNode.AppendChild(authorNode);

// save the value of the fields into the nodes
nameNode.AppendChild(nameText);
authorNode.AppendChild(authorText);

// Save to the XML file
xmlDoc.Save(@"C:\temp\GeneratedXmlDocumentLibrary.xml");
```

Il ne faut pas moins de 20 lignes de code pour créer un fichier XML de 9 lignes ! Créer un fichier de cette manière non seulement prend beaucoup de temps au développeur et en plus ne garantit pas une très bonne lisibilité de la structure XML décrite.

Si vous avez la chance de pouvoir utiliser une version du Framework supérieure ou égale au 3.5, nous verrons qu'il est beaucoup plus facile et lisible d'effectuer la création d'un arbre complet avec **LinqToXml**.

Validation d'un XmlDocument

Il est tout à fait possible de valider un fichier **XmlDocument**.

Pour cela il faut ajouter à la propriété **Schemas** du document un nouveau **XmlSchemaSet** ainsi qu'une callback dans l'appel de la méthode Validate().

A noter que la méthode Validate accepte aussi un XmlNode second argument : dans ce cas, seul ce nœud sera validé et non l'intégralité du document.

```
        // validation
XmlSchemaSet sc = new XmlSchemaSet();
sc.Add("", "SampleLibraryXsd.xsd");
xmlDoc.Schemas.Add(sc);
xmlDoc.Validate(new ValidationEventHandler(ValidationCallBack));
```

Ce code peut être rajouté après l'écriture d'un nouveau fichier.

Conclusion

Même si **XmlDocument** a été très utilisé dans le passé, ce n'est désormais plus une solution digne d'un développeur qui se respecte :

- Beaucoup de code pour générer un fichier
- Maintenabilité faible
- Performances Ok pour de petits fichiers, mauvaises pour des fichiers supérieures à 1Mo.

XPath

XPath est un langage pour localiser une portion de document XML. Il n'est pas spécifique à .NET et a été fournir une syntaxe et une sémantique aux fonctions communes à XPointer et XSL.

XPath a été rapidement adopté par les développeurs comme un langage d'interrogation simple d'emploi. (Extrait Wikipédia).

Contrairement aux autres solutions décrites dans cet ebook qui permettent de lire et d'écrire des fichiers XML, **XPath** ne permet que la lecture.

La principale conséquence est que cette technologie n'a de sens que si vous faites de la lecture seule. Il faudra utiliser une autre solution pour écrire du XML.

Lecture avec XPath

Nous avons voir comment utiliser XPath avec l'exemple que nous utilisons dans cet ouvrage :

```
internal List<Book> ReadData(string fullpath="./SampleLibrary.xml")
{
    List<Book> listBooks = new List<Book>();
    try
    {
        XPathDocument doc = new XPathDocument(fullpath);
        XPathNavigator nav = doc.CreateNavigator();

        XPathExpression expr = nav.Compile("//Book");
        XPathNodeIterator nodes = nav.Select(expr);
        while (nodes.MoveNext())
        {
              Book newBook = new Book();
              newBook.Id = nodes.Current.GetAttribute("Id","");
              newBook.Category = nodes.Current.GetAttribute("Category", "");
              nodes.Current.MoveToFirstChild();//  name tag
              newBook.Name = nodes.Current.Value;
              nodes.Current.MoveToNext();//author tag
              newBook.Author = nodes.Current.Value;
              listBooks.Add(newBook);
        }
    }
    catch(Exception ex)
    {
        //log error
    }
    return listBooks;
}
```

L'utilisation d'**XPath** commence par charger le document en passant son chemin complet en tant qu'argument du constructeur.

```
        XPathDocument doc = new XPathDocument(fullpath);
```

On créé un objet navigateur à partir de ce document :

```
       XPathNavigator nav = doc.CreateNavigator();
```

C'est à partir de ce navigateur que l'on va spécifier la requête. Cette requête intervient sous la forme d'un XPathExpression que l'on déclare comme ceci :

```
XPathExpression expr = nav.Compile("//Book");
```

Notre requête //Book va simplement récupérer une collection de Books dans le document.

Apres avoir spécifié la requête, encore faut-il l'exécuter :

```
XPathNodeIterator nodes = nav.Select(expr);
```

L'objet nodes contient les nœuds Book du document. Pour chaque nœud dans cette collection on va récupérer les valeurs des Id, Category, Name et Author.

```
while (nodes.MoveNext())
 {
       Book newBook = new Book();
       newBook.Id = nodes.Current.GetAttribute("Id","");
       newBook.Category = nodes.Current.GetAttribute("Category", "");
       nodes.Current.MoveToFirstChild();//  name tag
       newBook.Name = nodes.Current.Value;
       nodes.Current.MoveToNext();//author tag
       newBook.Author = nodes.Current.Value;
       listBooks.Add(newBook);
}
```

Les attributs Id et Category ne posent pas de problème car ils sont au même niveau que <Book>. Ils sont récupérés grâce à la méthode **GetAttribute**().

Pour aller récupérer le Name du Book il faut se déplacer dans l'arborescence XML en utilisant les méthodes **MoveToFisrtChild** qui permettent de se déplacer au premier nœud enfant.

La value du nœud est récupérer en utilisant la propriété Value.

Pour récupérer la valeur de<Author>, qui est un frère de <Name>, on utilise la méthode **MoveToNext**().

Il existe d'autres méthodes pour se déplacer dans l'arborescence XML ; le lecteur comprendra facilement leur usage, elles ne seront pas détaillées ici.

Quelques réflexions autour d'XPath

La force d'**XPath**, outre ses très bonnes performances en lecture, réside dans la complexité des requêtes que l'on peut effectuer. Dans notre exemple la requête est basique (//Book) .

On pourrait imaginer, si on disposait d'un fichier de données bien plus fourni, d'une requête listant les noms des livres dont la date de publication est supérieure à 01/01/2013, dont le prix est compris entre 8€ et 20€ dans la catégorie des livres animaliers, le tout en une seule ligne.

Il est également possible d'utiliser des fonctions agrégées comme un Count et de filtrer sur ces fonctions.

Nous allons voir succinctement quelques-unes des possibilités qu'offre **XPath** :

Kit de survie de la requête XPath

Dans cette section on va brièvement discuter de la syntaxe des requêtes.

Expression XPath	Résultat
`/`	sélectionne le *root element*, qui englobe tout le document sauf <?xml version="1.0"?>
`//Book`	sélectionne tous les éléments `"Book"` du document où qu'ils soient
`/Library/Book`	sélectionne tous les éléments `"Book"` qui sont directement enfants de Library
`//Book[@Id='2']`	sélectionne tous les éléments `"Book"` du document où qu'ils soient, ayant un attribut `"Id"` dont la valeur est`"2"`
`//Book[Name= 'Toto']`	sélectionne tous les éléments `"Book"` du document où qu'ils soient, ayant une balise `"Name"` dont la valeur est`"Toto"`
`Book[ count( Book/Name) >1 ]`	Sélectionne tous les “Book” ayant 2 balises Name.

L’objet de cet ouvrage n’est pas une description détaillée d’**XPath** donc nous nous arrêterons là pour la syntaxe. Si vous avez besoin d’une condition tordue pour trouver des données dans un fichier XML, vous pourrez trouver la requête **XPath** correspondante avec un peu d’efforts en cherchant sur Internet.

Ecriture avec XPath

En introduction je décrivais **XPath** comme une solution permettant de faire de lecture uniquement.

Cependant il est possible de combiner **XPath** avec d’autres solutions pour écrire des données.

Le cas le plus fréquent est d’utiliser XPath conjointement avec un **XmlDocument**. Pour insérer un noeud à un emplacement donné l’algorithme est le suivant :

```
try
{
    Book newBook = new Book{ Id = "4", Category ="Educational", Name ="Xml : Life and
Death",Author ="Whatever"};

    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.Load(fullpath);
    XPathNavigator nav = xmlDoc.CreateNavigator();

    XPathExpression expr = nav.Compile("//Book[@Id='2']");
    XPathNodeIterator nodes = nav.Select(expr);
    if (nodes.Count != 0 && nodes.MoveNext())
    {
        nodes.Current.InsertElementAfter("","Book","","");
        nodes.Current.MoveToNext(XPathNodeType.Element);
        nodes.Current.CreateAttribute("", "Id", "", newBook.Id);
        nodes.Current.CreateAttribute("", "Category", "", newBook.Category);
        nodes.Current.AppendChildElement("", "Name", "", newBook.Name);
```

```
            nodes.Current.AppendChildElement("", "Author", "", newBook.Author);
            xmlDoc.Save(fullpath);
        }
    }
    catch (Exception ex)
    {
        //log error
    }
```

Contrairement au cas de la lecture dans lequel on commence par créer un **XPathDocument**,

On va créer ici un `XmlDocument.`

```
    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.Load(fullpath);
    XPathNavigator nav = xmlDoc.CreateNavigator();

    XPathExpression expr = nav.Compile("//Book[@Id='2']");
    XPathNodeIterator nodes = nav.Select(expr);
```

On a également change la requête vers `"//Book[@Id='2'].`

L'objectif est d'insérer un nouveau nœud après le nœud Book existant dont l'Id est égale à 2.

Comme nous avons vu dans la rubrique précédente, [] permet de fixer une condition sur la valeur.

On utilise [@Id = 2] et non [Id=2] car cet Id est un attribut.

Une fois la requête exécutée on va insérer un nouveau nœud.

```
if (nodes.Count != 0 && nodes.MoveNext())
{
   nodes.Current.InsertElementAfter("","Book","","");
   nodes.Current.MoveToNext(XPathNodeType.Element);
   nodes.Current.CreateAttribute("", "Id", "", newBook.Id);
   nodes.Current.CreateAttribute("", "Category", "", newBook.Category);
   nodes.Current.AppendChildElement("", "Name", "", newBook.Name);
   nodes.Current.AppendChildElement("", "Author", "", newBook.Author);
   xmlDoc.Save(fullpath);
}
```

Pour cela, on créé un nouvel élément <Book> avec :

```
nodes.Current.InsertElementAfter("","Book","","");
```

On va se positionner sur cet élément pour créer les attributs

```
nodes.Current.CreateAttribute("", "Id", "", newBook.Id);
nodes.Current.CreateAttribute("", "Category", "", newBook.Category);
```

Le premier et le troisième argument de CreateAttribute correspond au préfixe et au namespace. Dans notre exemple ils sont mis à `""`.

On rajoute ensuite les balises <Name> et Author> et on sauvegarde le document :

```
nodes.Current.AppendChildElement("", "Name", "", newBook.Name);
nodes.Current.AppendChildElement("", "Author", "", newBook.Author);
```

```
xmlDoc.Save(fullpath);
```

Le résultat est le suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Library>
  <Book Id="1" Category="novel">
    <Info>
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
    </Info>
  </Book>
  <Book Id="2" Category="non-fiction">
    <Info>
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
    </Info>
  </Book>
  <Book Id="4" Category="Educational">
    <Name>Xml : Life and Death</Name>
    <Author>Whatever</Author>
  </Book>
  <Book Id="3" Category="novel">
    <Info>
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
    </Info>
  </Book>
</Library>
```

On a bien réussi l'insertion d'un élément après le Book avec l'Id = 2

Il est également possible de combiner **XPath** avec **LinqToXml**. Cependant les performances sont moins bonnes par rapport à l'utilisation unique de LinqToXml qui permet d'effectuer le même type d'opérations.

Conclusion

XPath est une solution en cours d'obsolescence. Avant l'arrivée de **LinqToXml**, **XPath** permettait 2 choses :

- Effectuer des requêtes très complexes sur du code XML de manière concise
- Lire des données XML très rapidement (par rapport à XmlDocument ou Dataset)

XPath nécessite un certain apprentissage pour se familiariser avec la syntaxe des requêtes, qui peuvent paraitre quelque peu ésotériques pour le non initié.

Si vous travaillez avec un Framework inférieure à 3.5, XPath est une bonne solution pour lire ou rechercher des données rapidement.

Sinon il faut mieux utiliser **LinqToXml**.

Dataset

Le Framework .NET 2.0 a introduit ADO.Net, technologie servant principalement à interagir avec une base de données. Le composant clé d'ADO.Net est un container appelé Dataset qui peut contenir le résultat d'une requête SQL sous forme de tableaux. Ainsi ce composant peut stocker des enregistrements provenant d'une ou plusieurs tables dans un même Dataset.

Il est également possible d'utiliser un Dataset pour lire du XML. Après tout un fichier XML contient des données et est parfois considéré comme une alternative à une base de données.

Lire un fichier XML avec un Dataset

Dans ce chapitre nous allons prendre un fichier de test un peu différent :

```
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="1" Category="novel">
    <Info>
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
    </Info>
  </Book>
  <Book Id="2" Category="non-fiction">
    <Info>
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
    </Info>
  </Book>
  <Book Id="3" Category="novel">
    <Info>
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
    </Info>
  </Book>
</Library>
```

C'est grosso-modo le même qu'auparavant mais on a ajouté un niveau de hiérarchie supplémentaire, la balise <Info>. La raison de ce changement est qu'à partir de 3 niveaux de hiérarchie la logique pour lire du Xml avec un Dataset est assez particulière comme nous allons le voir.

La lecture du Xml se fait de cette manière :

```
internal List<Book> ReadData(string fullpath = "./SampleLibraryDataset.xml")
        {
            List<Book> listBooks = new List<Book>();
            DataSet ds = new DataSet();
            ds.ReadXml(fullpath);
            foreach (DataRow row in ds.Tables["Book"].Rows)
            {
                Book book = new Book();
                book.Id = row["Id"].ToString();
                book.Category = row["Category"].ToString();

                DataRow[] children = row.GetChildRows("book_info"); // relation name

                book.Name = (children[0]["Name"]).ToString(); // there is only 1 row
in children
                book.Author = (children[0]["Author"]).ToString();
```

```
            listBooks.Add(book);
        }
        return listBooks;
    }
```

D'abord on initialise le Dataset :

```
Dataset ds = new DataSet();
```

Ensuite on appelle la méthode *ReadXml* de ce dernier :

```
ds.ReadXml("./SampleLibrary.xml");
```

Les données sont désormais contenues dans ce **Dataset**. Pour y accéder on peut procéder de la manière suivante :

```
foreach (DataRow row in ds.Tables["book"].Rows)
{
  Book book = new Book();
  book.Id = row["Id"].ToString();
  book.Type = row["Type"].ToString();

  DataRow[] children = row.GetChildRows("book_info"); // relation name
  // there is only 1 row in children
  book.Name = (children[0]["Name"]).ToString();
  book.Author = (children[0]["Author"]).ToString();

  listBooks.Add(book);
}
```

La récupération des valeurs fonctionne un peu comme pour un dictionnaire :

Les données de l'enregistrement sont contenues dans une DataRow accessible en spécifiant le nom de la colonne entouré de crochets.

```
row["Id"]
```

Ceci fonctionne d'ailleurs très bien pour les attributs qui sont au même niveau que l'objet en question Book dans le fichier XML.

En revanche pour lire les valeurs des propriétés Name et Author, il faut se souvenir de la structure du fichier XML :

```
<Book Id="1" Type="novel">
    <Info>
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
    </Info>
  </Book>
```

Les données ne sont pas directement dans Book mais dans une balise à l'intérieur de celui-ci, Info.

De même on ne peut accéder directement aux champs <Author> et <Name> directement dans le Dataset. Il faut accéder à la balise fille de <Book>, ce qui se fait dans le Dataset via la méthode **GetChildRows**().

Noter l'argument de cette méthode : `book_info.` Pour récupérer les valeurs contenues dans la balise interne, il suffit de séparer le nom de la balise parente par _ et d'écrire le nom de la balise enfant : <baliseparent>_<baliseEnfant>

Cette méthode renvoie un tableau de **DataRow** qui reflète la structure de la balise <Info>.

Il est ensuite facile d'accéder aux valeurs des propriétés Name et Author :

```
book.Name = (children[0]["Name"]).ToString();
book.Author = (children[0]["Author"]).ToString();
```

Comme on peut le voir, l'utilisation de Dataset pour lire du XML est assez simple.

Il y a moins de travail à fournir pour le développeur par rapport à l'utilisation d'un **XmlReader** ou un **XmlDocument**.

Cependant, cette manière de lire du XML ne fait pas abstraction de la structure XML et tout changement dans la structure du fichier XML devra être répercuté dans le code de lecture.

Ecrire un fichier à l'aide d'un Dataset

Il existe 2 méthodes pour générer du XML avec Dataset :

GetXml()

Cette méthode va renvoyer sous forme de string les données provenant de toutes les DataTables du Dataset.

WriteXml(string filepath)

La méthode Write XML va, tout comme GetXml(), récupérer les données de toutes les datatables du Dataset, mais plutôt que de renvoyer ce code sous forme de string, va directement l'écrire dans un fichier dont le math est fourni en argument.

Il est donc assez simple de transformer des données provenant d'une source non XML en un fichier XML. On peut penser par exemple à récupérer des champs d'une base de données ou d'un fichier plat.

A noter que LinqToXml propose également ce type de transformation de manière relativement simple. Si vous utilisez une version du Framework .NET supérieure ou égale au 3.5, il sera souvent préférable d'utiliser LinqToXml.

Le code ci-dessous permet d'enregistrer sous forme de fichier XML les données contenues dans le Dataset :

```
internal void WriteData2()
{
   List<Book> books = BenchmarkHelper.AddBooks();
   DataSet ds2 = new DataSet("Library");
   DataTable table1 = ds2.Tables.Add("Book");
   table1.Columns.Add("Id", typeof(int));
   table1.Columns.Add("Category", typeof(string));

   DataTable table2 = ds2.Tables.Add("Info");
   table2.Columns.Add("Id", typeof(int));
   table2.Columns.Add("Name", typeof(string));
   table2.Columns.Add("Author", typeof(string));

   table1.Columns[0].ColumnMapping = MappingType.Attribute;
   table1.Columns[1].ColumnMapping = MappingType.Attribute;
   table2.Columns[0].ColumnMapping = MappingType.Hidden;

   DataRelation customerOrders = ds2.Relations.Add("rel1",
ds2.Tables["Book"].Columns["Id"],
   ds2.Tables["Info"].Columns["Id"]);
   customerOrders.Nested = true;

   foreach (Book b in books)
   {
     table1.Rows.Add(b.Id,b.Category);
     table2.Rows.Add(b.Id,b.Name,b.Author);
    }
    ds2.WriteXml(@"./Generated_Dataset.xml");

}
```

Tout d'abord il faut créer le Dataset et définir les colonnes et le format de celles-ci. Rien de particulièrement difficile pour la table Book :

```
DataSet ds2 = new DataSet();
DataTable table1 = ds2.Tables.Add("Book");
table1.Columns.Add("Id", typeof(int));
table1.Columns.Add("Category", typeof(string));
```

A noter que les colonnes Id et Category sont à afficher sous forme d'attribut, d'où le code suivant :

```
table1.Columns[0].ColumnMapping = MappingType.Attribute;
table1.Columns[1].ColumnMapping = MappingType.Attribute;
```

On va ensuite créer une seconde table, Info. Il est nécessaire de créer une autre table sinon les liens de hiérarchies seront mal reconstitués, par exemple comme cela :

```
<Book Id="3" Category="Fiction">
<Info/>
<Name>gbmkuinuqisw</Name>
<Author>rrwhfblnsfyo</Author>
</Book>
```

Il faut bien comprendre que la manière qu'un Dataset fonctionne est proche d'une base de donnée. D'ailleurs, l'usage principal est de stocker et d'opérer des modifications offline sur des données suite à une requête en base de données. Dans une base de donnée la notion de hiérarchie de balise

n'existe pas, contrairement au fichier XML (balise Père, Enfant ..) ; l'adaptation du DataSet au XML est donc un peu tordue. Pour modéliser ces niveaux de hiérarchie, on est obligé d'utiliser des clés étrangères pour faire le lien.

Afin de faire le lien entre la table Info et la table Book on va rajouter une colonne Id qui servira uniquement pour la clé étrangère :

```
DataTable table2 = ds2.Tables.Add("Info");
table2.Columns.Add("Id", typeof(int));
table2.Columns.Add("Name", typeof(string));
table2.Columns.Add("Author", typeof(string));
table2.Columns[0].ColumnMapping = MappingType.Hidden;
```

Cette colonne ne sera pas affichée dans le XML grâce à la valeur Hidden de l'enum MappingType.

Cette colonne n'est là que pour représenter la structure hiérarchique du fichier Xml car il n'y a pas d'attribut Id dans le fichier Xml, c'est pour cela qu'elle est cachée.

Ensuite vient la création de la relation de clé étrangère liant les 2 tables :

```
DataRelation relationBookInfo = ds2.Relations.Add("rel1",
ds2.Tables["Book"].Columns["Id"],
ds2.Tables["Info"].Columns["Id"]);
relationBookInfo.Nested = true;
```

la propriété `relationBookInfo.Nested = true` est très importante. Sans elle on n'obtiendrait le XML suivant :

```
<?xml version="1.0" standalone="yes"?>
<Library>
  <Book Id="1" Category="NonFiction" />
  <Book Id="2" Category="NonFiction" />
  <Info>
    <Name>Rich dad Poor Dad</Name>
    <Author>Kiyosaki</Author>
  </Info>
  <Info>
    <Name>the 4h work week</Name>
    <Author>Feriss</Author>
  </Info>
</Library>
```

Vous remarquerez que la balise Info n'est pas incluse dans la balise Book.

Il ne reste plus qu'à peupler le Dataset et créer le fichier XML grâce à la méthode WriteXml.

```
foreach (Book b in books)
{
   table1.Rows.Add(b.Id,b.Category);
   table2.Rows.Add(b.Id,b.Name,b.Author);
}
ds2.WriteXml(@"./Generated_Dataset.xml");
```

Le fichier généré est bien celui attendu :

```
<?xml version="1.0" encoding="utf-8"?>
<Library xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Book>
    <Id>1</Id>
    <Category>NonFiction</Category>
    <Name>Rich dad Poor Dad</Name>
    <Author>Kiyosaki</Author>
  </Book>
  <Book>
    <Id>2</Id>
    <Category>NonFiction</Category>
    <Name>the 4h work week</Name>
    <Author>Feriss</Author>
  </Book>
</Library>
```

Cet extrait de code peut paraitre assez difficile à comprendre au premier abord et c'est normal. N'hésitez pas à télécharger les extraits de code en ligne et à les bidouiller pour bien comprendre le fonctionnement.

Ceci dit, ce chapitre est plus présent pour vous montrer qu'écrire du XML avec un Dataset ne parait pas ma solution la plus simple contrairement à ce qui peut être dit sur le Web ! Libre à vous de ne PAS choisir d'utiliser un DataSet pour cela.

Conclusion

L'utilisation d'un **Dataset** est encore un autre moyen pour lire ou générer du XML. Dans ce cas, on ne travaille pas directement avec les balises XML. C'est l'architecture du **Dataset** et ses propriétés qui sont utilisées pour définir la forme du fichier XML en sortie.

On remarquera que les liens de hiérarchie entre balises XML sont gérés à l'aide de clés étrangères, ce qui impose la création de colonnes factices dans le Dataset et de relations de clés étrangères.

Si générer du XML avec **XmlWriter** ou **XmlDocument** est un travail fastidieux, avec un **Dataset** cela peut vite devenir un travail très complexe. Imaginez un fichier avec une dizaine de niveaux de hiérarchies …

En termes de performances, l'utilisation d'un **Dataset** pour lire du XML est la pire solution de celles étudiées ici. Elle n'est donc conseillée que si le reste de l'application utilise ADO.NET et pour de petits fichiers. Si la performance est importante dans votre application, il faudra chercher une autre solution.

Sérialisation

Le Namespace **System.Xml.Serialization** contient des classes servant à sérialiser des objets en fichiers XML et inversement. La sérialisation est une alternative intéressante au modèle Dom et au Dataset décrits dans les paragraphes précédents. Dans ce chapitre, nous allons commencer par générer un fichier Xml ; ensuite seulement nous aborderons la lecture.

Ecrire un fichier XML en utilisant la sérialisation

Ecrire un fichier XML avec la sérialisation est une approche très différente du modèle DOM. Le modèle des données du fichier XML est contenu dans une classe .NET tout comme pour le Dataset. En revanche ce modèle est bien plus simple à utiliser que dans le cas du Dataset.

Pour rappel voici le fichier que l'on souhaite générer :

```
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="1" Category="novel">
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
  </Book>
  <Book Id="2" Category="non-fiction">
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
  </Book>
  <Book Id="3" Category="novel">
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
  </Book>
</Library>
```

Pour cela, nous allons créer la classe suivante :

```
public class Book
{
    [XmlAttribute]
    public string Id { get; set; }
    [XmlAttribute]
    public string Category { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
}
[XmlRoot("Library")]
public class Library : List<Book> { }

internal void WriteData()
        {
            try
            {
                List<Book> listBooks = AddBooks();
                Library lib = new Library();
                lib.AddRange(listBooks);

                string fullPath = @"./Generated_Serialization.xml";
```

```
                var output = new StringBuilder();
                var settings = new XmlWriterSettings { Encoding = Encoding.UTF8,
Indent = true };
                using (Stream fs = new FileStream(fullPath, FileMode.Create))
                {
                    using (var xmlWriter = XmlWriter.Create(fs, settings))
                    {
                        XmlSerializer serializer = new XmlSerializer(typeof(Library));
                        serializer.Serialize(xmlWriter, lib);
                    }
                }
            }
            catch (Exception ex)
            {
                //_log.Error(ex.Message);
            }
        }
```

Le fonctionnement est le suivant :

On créé un objet **XmlWriterSettings** qui sera un des arguments à fournir au **XmlWriter**.

```
var settings = new XmlWriterSettings { Encoding = Encoding.UTF8, Indent = true };
```

On souhaite que la sortie soit un fichier XML, il est donc nécessaire de fournir au **XmlWriter** un **Stream**. Si on avait souhaité obtenir le code XML dans un string, il aurait suffi de fournir un **StringBuilder** au **XmlWriter**.

```
using (Stream fs = new FileStream(fullPath, FileMode.Create))
```

Enfin, le code e la sérialisation en lui-même :

```
using (var xmlWriter = XmlWriter.Create(fs, settings))
{
   XmlSerializer serializer = new XmlSerializer(typeof(Library));
   serializer.Serialize(xmlWriter, lib);
}
```

La méthode *Serialize* de l'objet **XmlSerializer** permet de faire la sérialisation des objets Book contenus dans lib. Pour cela elle nécessite en paramètre un **XmlWriter** et en second paramètre l'objet à sérialiser.

Comme vous pouvez le constater, le code est très court ! La sérialisation proprement dite prend 3 lignes, sans compter les accolades.
Il ne faut que quelques lignes pour générer un fichier XML. De plus le développeur n'a pas à écrire la gestion des balises, la hiérarchie des balises entre elles etc…
C'est la classe Book qui est utilisée pour définir cela. **La complexité de la structure hiérarchique du XML est déléguée au modèle objet.** C'est le principal attrait de la sérialisation pour générer du XML.

L'exemple ci-dessus est un exemple simple de ce qui est possible de réaliser avec la sérialisation. Comme il s'agit d'une solution acceptable pour gérer des fichiers XML, nous allons creuser un peu le sujet.

Ce qui est sérialisable et ce qui ne l'est pas

Les classes, structures, et énumérations sont sérialisables. En revanche les interfaces ne sont pas sérialisables.
Tous les types à sérialiser doivent être publics. Seuls les champs et propriétés publics sont sérialisés.

Les propriétés à sérialiser ne doivent pas être en lecture seule à l'exception des collections.

Le type à sérialiser doit posséder un constructeur par défaut (public et sans paramètres) : cela est nécessaire pour que la désérialisation puisse créer une instance de ce type.

Tous les types à sérialiser doivent implémenter l'interface *IXmlSerialisable* ou être composés de types implémentant cette interface.

Attention *IDictionary* **n'est pas** sérialisable. Si vous souhaitez sérialiser un Dictionnary, il va falloir étendre la classe pour implémenter *IXmlSerialisable.* Vous trouverez sur Internet des implémentations clé en main sans trop de difficultés, par exemple ici :

http://www.codeproject.com/Questions/454134/Serialize-Dictionary-in-csharp

Sérialisation d'une collection

L'exemple ci-dessus concerne la sérialisation d'une collection qui est un cas relativement fréquent. Nous allons ici voir ce qui se passe sans la classe Library. L'utilisation de la classe Library est optionnelle, il aurait été tout à fait possible de sérialiser une List<Book>.

Le code aurait donc ressemblé à celui-ci :

```
using (var xmlWriter = XmlWriter.Create(fs, settings))
{
    XmlSerializer serializer = new XmlSerializer(typeof(List<Book>));
    serializer.Serialize(xmlWriter, listBooks);
}
```

Ce serait donc bien une List<Book> qui est passé en argument dans le constructeur **XmlSerializer**, et l'instance de la liste dans la méthode **Serialize**().

Le résultat est le suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<ArrayOfBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Book Id="1" Category="novel">
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
  </Book>
  <Book Id="2" Category="non-fiction">
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
  </Book>
  <Book Id="3" Category="novel">
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
  </Book>
</ArrayOfBook>
```

Une seule chose a changé entre le résultat du premier exemple et celui-ci : le nom de la balise racine qui est passé de Library à ArrayOfBook.
On aurait pu s'attendre à obtenir quelque chose comme ListOfBooks mais la sérialisation ne fait pas de différence entre les types de collection. Que l'on passe un tableau ou une liste la balise sera toujours <ArrayOf…>.

Quelques attributs pour customiser la sérialisation

Il existe un petit nombre d'attributs permettant de customiser la sérialisation. Nous allons nous pencher sur ceux-ci dans cette section.

XmlRoot

L'attribut XmlRoot s'applique à une classe et non à une propriété. Celui-ci permet de définir l'élément racine du fichier XML et de définir son nom :

```
[XmlRoot("Library")]
public class Library : List<Book> { }
```

La racine du fichier généré dans l'exemple ci-dessus est bien la balise <Library>.

XmlIgnore

Dans le cas où on ne souhaite pas sérialise certaines propriétés il convient de lui appliquer l'attribut XmlIgnore .

```
[XmlIgnore]
public int NbItems { get; set; }
```

XmlElement

On ne souhaite parfois pas avoir le même nom de balise dans le fichier XML que celui de la propriété. XmlElement permet, sans changer le nom de la propriété, de changer le nom de la balise du fichier généré.

```
[XmlElement("Titre")]
public string Name { get; set; }
```

le code XML généré sera alors le suivant (extrait) :

```
<Book Id="1" Category="NonFiction">
    <Titre>Rich dad Poor Dad</Titre >
    <Author>Kiyosaki</Author>
  </Book>
```

XmlArray et XmlArrayItem

XmLArray permet de modifier le nom de la balise d'une collection.

```
[XmlArray("Livres")]
public List<Book> Books { get; set; }
```

Cependant cela ne modifiera pas le nom des balises des éléments de la liste :

```
<Livres>
 <Book>
```

```
  …
 </Book>
</Livres>
```

Pour changer également le nom des éléments de la liste

```
[XmlArray("Livres")]
[XmlArrayItem("Livre")]
 public List<Book> Books { get; set; }
```

ce qui donnera :

```
<Livres>
 <Livre>
  …
 </Livre>
</Livres>
```

XmlAttribute

Jusqu'alors, toutes les propriétés étaient transformées en éléments. Cependant on peut souhaiter passer une propriété, non sous forme d'élément, mais sous forme d'attribut de son élément parent.

C'est l'utilité de la balise XmlAttribute. Nous avons d'ailleurs utilisé celle-ci dans l'exemple en début de chapitre.

```
public class Book
{
    [XmlAttribute]
    public string Id { get; set; }
    [XmlAttribute]
    public string Category { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }
}
```

Et bien obtenu Id et Category sous forme d'attributs dans le fichier résultat.

```
<Book Id="1" Category="NonFiction">
```

Format des propriétés

Supposons que notre classe contienne une propriété DateTime et que l'on souhaite afficher uniquement la date et non l'heure de celle-ci.

Il n'y a pas d'attribut XML dans ce cas. Il faut configurer ce format dans les accesseurs get ; (pour la sérialisation) et set ; (pour la désérialisation), quitte à créer une nouvelle propriété dédiée à cela.

```
[XmlIgnore]
public DateTime PublicationDate { get; set; }

public string PublicationDateFormatted
{ get { return PublicationDate.ToShortDateString();}
    set {PublicationDate = DateTime.Parse(value);
```

```
}
```
On a donc créé une nouvelle propriété PublicationDateFormatted qui ne fait que chercher la valeur de PublicationDate et de la formater proprement.

Gestion des classes dérivées

La sérialisation ne supporte pas les classes dérivées sans un peu de travail supplémentaire.

Supposons qu'on procède à la spécialisation suivante :

```
public class EBook : Book
{
    public string Format { get; set; }
}
Et le code suivant :

EBook myebook = new EBook();
myebook.Format = "MOBI";
myebook.Name = "MyEbook";
lib.Add(myebook);
```

la classe Ebook dérive de Book mais comporte une propriété supplémentaire Format.

La collection lib contient alors à la fois des objets Book et un objet EBook. Comme Ebook dérive de Book l'ajout de l'objet à la collection est valide.

Si on essaye de sérialiser cette collection on obtient :

InvalidOperationException : *Le type ArticleXmlSerialization.Ebook n'était pas attendu. Utilisez l'attribut XmlInclude ou SoapInclude pour spécifier les types qui ne sont pas connus statiquement.*

XmlSerializer ne sait pas directement gérer la sérialisation : il faut adjoindre l'attribut XmlInclude

```
[XmlInclude(typeof(EBook))]
public class Book
{
…
}
```
Dans ce cas la sérialisation fonctionne et retourne :

```
<?xml version="1.0" encoding="utf-8" ?>
<Library>
  <Book Id="1" Category="novel">
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
  </Book>
  <Book Id="2" Category="non-fiction">
      <Name>My non fiction book</Name>
      <Author>Someone</Author>
  </Book>
  <Book Id="3" Category="novel">
      <Name>The little prince</Name>
      <Author>Antoine de Saint-Exupéry</Author>
  </Book>
  <Book xsi:type="EBook">
      <Name>MyEbook </Name>
      <Format>MOBI</Format>
  </Book>
```

```
</Library>
```

Remarquez que le type réel du livre est décrit dans la balise xsi :Type

Personnalisation avancée : Implémentation de IXmlSerializable

Si les attributs décrits ci-dessus ne suffisent pas pour arriver à vos fins, tout n'est pas perdu ! Il reste un moyen de s'en sortir, l'implémentation de l'interface IXmlSerializable. Durant la sérialisation,.NET va d'abord regarder si le type à sérialiser implémente cette interface.

IXmlSerializable est composée de 3 méthodes :

- GetSchema
- ReadXml
- WriteXml

Nous ne détaillerons pas la manière d'implémenter ces 2 méthodes. Il existe de nombreuses ressources sur Internet décrivant cela et le code ressemble quelque peu à la syntaxe utilisée avec XmlDocument.

Lire du XML avec la sérialisation

La lecture d'un fichier XML est le pendant de l'écriture : Au lieu de sérialiser on va désérialiser.

```
internal Library ReadData(string fullpath = @"./SampleLibrary.xml")
{

      XmlSerializer xs = new XmlSerializer(typeof(Library));
      Library res = new Library();

      using (StreamReader sr = new StreamReader(fullpath))
      {
         using (var xmlReader = XmlReader.Create(sr))
         {
            XmlSerializer serializer = new XmlSerializer(typeof(Library));
            res = xs.Deserialize(xmlReader) as Library;
         }
      }
     return res;
}
```

Les différences entre écrire un fichier XML (sérialisation) et lire un fichier XML (Désérialisation) se concentrent à 2 niveaux :

- Utilisation d'un **StreamReader** dans le cas de la lecture, d'un **StreamWriter** pour l'écriture
- Pour désérialiser on appelle *`Deserialize`* de **`XmlSerializer`** Alors que *Serialize* est appelée pour sérialiser.

Rien de bien sorcier…

A noter toutefois que pour désérialiser vous devez avoir la classe décrivant le modèle. Dans notre exemple il s'agit de la classe Book.

Si vous n'avez pas créé ce fichier lors du développement de l'application et que vous en avez besoin juste pour désérialiser, et bien il va falloir le créer ; 2 méthodes :

- Si vous êtes naufragé sur une ile du pacifique et que, de toute manière, le prochain navire passera dans 200 ans, alors vous pouvez le créer à la main.
- Sinon, une seconde approche est d'utiliser XSD.EXE qui va générer automatiquement la/les classe(s) C# nécessaires pour la désérialisation à partir du fichier XSD.

XSD.exe

L'outil XSD.EXE est fourni avec le SDK Visual studio. Par exemple, sur mon poste il est situé à cet emplacement :

C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin

Cela ne sera peut-être pas forcément le cas sur votre machine mais je suis sûr que vous finirez par le trouver même si vous avez moins de 200 ans pour accomplir cette tâche.

Si vous n'avez pas de fichier XSD, vous pouvez le générer à partir du XML avec cette commande

```
Xsd C:\temp\<XMLFile>.xml /out:/C:\temp
```

Cela suppose que votre fichier xml se trouve dans C:\temp et que vous ayez effectué un cd vers l'emplacement du fichier xsd.exe.

Ensuite pour générer les classes à partir du xsd, il suffit d'entrer cette commande :

```
Xsd /classes C:\temp\<XsdFile>.xsd /out:/C:\temp
```

Le code généré peut paraitre assez complexe (extrait) :

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.3038")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true)]
public partial class LibraryBook {

    private LibraryBookInfo[] infoField;

    private string idField;

    private string categoryField;


    /// <remarks/>
    [System.Xml.Serialization.XmlAttributeAttribute()]
    public string Id {
        get {
            return this.idField;
        }
        set {
            this.idField = value;
        }
    }

    /// <remarks/>
    [System.Xml.Serialization.XmlAttributeAttribute()]
    public string Category {
```

```
        get {
            return this.categoryField;
        }
        set {
            this.categoryField = value;
        }
    }
}
```

Et voilà! Vous pouvez maintenant désérialiser en toute tranquillité.

Validation avec la sérialisation

Avant de désérialiser un fichier il est intéressant de procéder à une validation du document.

L'approche est en fait la même que pour valider un fichier XML avec **XmlReader** car la sérialisation utilise un tel objet et c'est à ce niveau qu'est effectuée la validation de toute manière.

```
XmlSchemaSet sc = new XmlSchemaSet();
sc.Add("", "SampleLibraryXsd.xsd");

XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas = sc;
settings.ValidationEventHandler += new
ValidationEventHandler(ValidationCallBack);

using (var xmlReader = XmlReader.Create(sr, settings))
{
….etc
```

Comme vu précédemment l'objet **XmlReaderSettings** contient les options de configuration du **XmlReader**. C'est dans cette classe que l'on associe le schéma.

Cet objet settings est ensuite passé au constructeur de la classe **XmlReader**.

Performances

Les performances de sérialisation sont très respectables pour des fichiers de taille supérieure à 1Mo.

Dans le cas de petits fichiers, c'est malheureusement la solution la moins performante. La génération d'un fichier avec beaucoup de données est réalisée en un temps acceptable.

Bien sûr, il sera toujours plus rapide de travailler avec **XmlReader**/**XmlWriter** car la sérialisation va utiliser ces classes dans le processus.

Conclusion

La sérialisation est un excellent moyen pour générer des fichiers XML. Le développeur n'a pas à écrire du code bas niveau relatif aux balises, tout est configuré dans le modèle de classes avec des attributs XML associés aux propriétés. Ces propriétés correspondent aux balises qui seront générées.

La hiérarchie est également gérée automatiquement : il ne sera pas nécessaire d'effectuer cela à la main comme c'est le cas avec **Xmlreader**/**XmlWriter** et **XmlDocument**.

La personnalisation de la sérialisation peut être poussée plus loin en implémentant IXmlSerializable afin de gérer les types non sérialisables ou d'obtenir un comportement non prévu avec les attributs de configuration XML.

Il y a cependant un revers à la médaille : la lecture d'un fichier XML, désérialisation, intervient sur l'ensemble du fichier. Afin de rechercher une balise, valeur etc. Précises dans le document, il est tout à fait envisageable d'utiliser **XPath** par exemple.

Coupler **Sérialisation** et **XPath** (ou autre) s'avère être une solution performante et maintenable.

Linq To Xml

Les technologies avant **LinqToXml** permettant de manipuler des documents XML comme **XmlDocument** ou encore **XPath** était relativement difficiles à maitriser pour le développeur et nécessitait de la motivation et du temps pour être maitrisées.

Microsoft a introduit Linq dans la version 3.5 du Framework .NET. LinqToXml se veut une API moderne, simple et complète pour traiter les fichiers XML. A elle seule elle regroupe la majorité des fonctionnalités procurées par les autres API, le tout dans un style de programmation relativement simple.

Nous allons dédier une partie importante de ce livre à LinqToXml car c'est actuellement la meilleure solution dans bien des contextes.

Avant de commencer à voir comment utiliser LinqToXml, il est intéressant de s'intéresser à la hiérarchie des classes :

XObject

La classe hiérarchiquement supérieure est la classe XObject : cette classe sert de base à la plupart des classes LinqToXml et procure la fonctionnalité d'ajout/suppression d'annotations via les méthodes

AddAnnotation()/RemoveAnnotation()

XNode

Dépendant directement de XObject, **XNode** est la classe de base pour les nœuds Elements. C'est à ce niveau qu'est implémenté les méthodes d'ajout de nœuds **AddAferSelf**, **AddbeforeSelf** et leur pendant pour la suppression.

XContainer

Encore un cran au-dessous, il y a la classe XContainer. Celle-ci est utile pour contenir des XNode contenant d'autres XNode. **XContainer** dispose de méthodes telles **Add**(), **AddFirst**(),**ReplaceNodes**(), **RemoveNodes**().

XElement

C'est la classe la plus fondamentale de LinqToXml. Cette classe représente des nœuds XML pouvant contenir d'autres nœuds XML. Xelement offre une méthode de chargement **Load**() qui permet de charger un fichier XML depuis plusieurs types de sources de données et une méthode **Parse** permettant à un nœud d'être créé à partir d'une string représentant du XML. XElement offre également la méthode **Save**() pour générer un fichier.

XDocument

XDocument représente un document XML. Cependant avec LinqToXml , il est possible créer un document XML sans utiliser cette classe : la seule classe XElement permet de créer un arbre XML

complet. En revanche XDocument sert à rajouter une déclaration XML (XDeclaration) , un type de document ou des instructions de traitements (XSS)

Lire un fichier avec LinqToXml

Lire un fichier avec LinqToXml est très simple. Regardez la ligne suivante :

```
var books = from b in XElement.Load("SampleLibrary.xml").Elements("Book")
                          select b;
```

Pour charger un fichier XML, on fait appel à la méthode Load() de XElement. La propriété `").Elements("Book")` indique que l'on ne souhaite que récupérer le contenu des balises Book du fichier XML.
La variable en sortie books est de type IEnumerable<XElement>. Il est donc possible d'itérer sur cette collection pour récupérer les valeurs des champs :

```
foreach (var b in books)
{
  Book newBook = new Book();
  newBook.Id = b.Attribute("Id").Value.ToString();
  newBook.Type = b.Attribute("Type").Value.ToString();
  newBook.Name = b.Element("Name").Value.ToString();
  newBook.Author = b.Element("Author").Value.ToString();
  listBooks.Add(newBook);
}
```

Le code ci-dessus est plutôt simple à comprendre. Une précision cependant : Que se passe-t-il si la propriété Value n'est pas utilisée ? C'est-à-dire :

```
newBook.Name = b.Element("Name").ToString();
```

Dans ce cas, la string retournée contiendra par exemple : <Name>For whom the bell tolls</Name>

Avec l'appel à la propriété Value, la string retournée sera : For whom the bell tolls.

A noter qu'avec Linq il est possible de réécrire le code ci-dessus de cette manière :

```
var listbooks2 =
(from b in XElement.Load("SampleLibrary.xml").Elements("Book")
select new Book
{
   Id =b.Attribute("Id").Value,
   Category = b.Attribute("Category").Value,
   Name = b.Element("Name").Value,
   Author = b.Element("Author").Value
}).ToList<Book>();
```

Au lieu de récupérer une collection de XElement, on utilise ces XElement pour créer directement les objets Book et retourner une liste contenant ces derniers.

Il est intéressant de noter que la méthode Load() accepte aussi une URL : il est donc possible de lire les contenus de flux RSS de cette manière.

Il est également possible d'utiliser une source de données externe pour injecter les données du fichier XML à créer. C'est un bon moyen d'éviter d'avoir à réécrire la structure XML à la main.

Ecrire un fichier avec LinqToXml

Il y a plusieurs moyens de créer un fichier avec LinqToXml.

La première approche est appelée approche fonctionnelle : Le développeur écrit à la main l'intégralité de la structure du fichier.

```
XElement xml = new XElement("Library",
   new XElement("Book",
   new XAttribute("Id", 1),
   new XAttribute("Category", "Novel"),
            new XElement("Name", "For whom the bell tolls"),
            new XElement("Author", "Ernest Hemingway"))),
   new XElement("Book",
   new XAttribute("Id", 2),
   new XAttribute("Category", "Novel"),
            new XElement("Name", "War and Peace"),
             new XElement("Author", "Tolstoi"))));

xml.Save(@"C:\temp\GeneratedXmlFile.xml");
```

Ce bout de code va créer le fichier suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<Library>
  <Book Id="1" Category="Novel">
      <Name>For whom the bell tolls</Name>
      <Author>Ernest Hemingway</Author>
  </Book>
  <Book Id="2" Category="Novel">
      <Name>War and Peace</Name>
      <Author>Tolstoi</Author>
  </Book>
</Library>
```

Cette approche est appelée fonctionnelle car on peut créer un arbre XML complet en une seule instruction. Le code ressemble plus à l'arbre généré par cette instruction que l'approche impérative qui ressemblerait au code ci-dessous :

```
XElement xml = new XElement("Library");
xml.Add(new XElement("Book"));
...
```

AddIn de génération de code automatique PasteXmlAsXelement

Ecrire ce code à la main est une tache longue et fastidieuse. Il existe cependant un plugin qui permet de transformer du code XML existant en arborescent de XElement (cf. extrait de code plus haut) via un simple copier-coller.

Le fonctionnement est le suivant :

- Sélectionner un bout de code XML existant
- Copier
- Aller dans Edit/ Paste Xml As XElement dans Visual studio.
- Le code collé est représenté sous la forme d'une arborescence de XElement.

Pour installer cet Addin PasteXmlAsLinq, aller le télécharger sur internet :

http://code.msdn.microsoft.com/windowsdesktop/PasteXmlAsLinq-fe6d0540

- Ouvrez la solution Visual studio PasteXmlAsLinq.sln
- Compiler le projet (Après une conversion si vous utilisez Visual studio 2010 ou 2012)
- Récupérer les 2 fichiers PasteXmlAsLinq.dll et PasteXmlAsLinq.AddIn
- Les copier dans: Documents\Visual Studio 2010\AddIns
- Redémarrer Visual studio et ouvrir la solution dans laquelle vous travaillez.

Dans le menu **Edit**, vous devriez voir apparaitre un nouveau menu « Paste Xml As XElement »

Maintenant que nous avons vu de manière basique comment lire et écrire du XML avec LinqToXml, il est temps de rentrer un peu plus en profondeur sur les classes disponibles et leur fonctionnement.

XElement

La classe Xelement dispose de 3 constructeurs :

Xelement(XName name)

XElement(XName name,object content)

XElement(XName name,params object[] content)

Le type de l'objet content peut être :

string

Par exemple :

```
new XElement("Name", "War and Peace")
```

LinqToXml va se charger de la creation de du nœud interne XText contenant dans l'exemple ci-dessus « War and peace ».

XText

L'objet XText peut contenir soit une string ou une valeur de CData. On utilisera cette implémentation pour Cdata uniquement. Il est plus lisible de passer une string directement sans passer par un objet XText.

XAttribute

Ajouté comme attribut

IEnumerable

L'énumérable est parcouru et traité pour chaque objet de l'itération.

Namespace

Lors de la création d'un fichier XML dans la vie réelle, il est commun d'avoir à utiliser les namespaces.

Avec LinqToXml on procède de la manière suivante :

```
XNamespace ns = "http://gogo.com";
XElement elt = new XElement(ns + "book");
```

Après la création de namespace, il suffit de réutiliser celui-ci durant la creation d'un élément.

Comment ajouter du contenu au fichier XML ?

La méthode principale pour ajouter des nœuds à un XElement est la méthode Add() qui accepte ces 2 signatures :

```
public void Add( object content)
public void Add( params object[] content)
```

On peut donc ajouter à un XElement un ou plusieurs nœuds. Par exemple :

```
XElement elt = new XElement("Library");
elt.Add(new XElement("book",
    new XAttribute("Id",1),
    new XElement("Name","La trilogie sur le temps")));
```

Ce code va rajouter au nœud Library, un nœud Book avec un attribut Id = 1 et une balise Name dont la valeur est « La trilogie sur le temps».(Ouvrage fondamental ésotérique dans lequel Paco Rabanne explique que la fin du monde aura lieu …. à une date passée désormais ; C'est encore plus drôle que les romans de Terry Prachett ;A lire absolument !)

De la même manière supprimer des nœuds est assez simple :

```
// supprime le premier livre
books.Element("book").Remove();
// supprime tous les livres
books.Elements("book").Remove();
```

Naviguer dans l'arborescence XML

Pour Naviguer dans l'arborescence XML il existe plusieurs méthodes :

Element() & Attribute()

La méthode Element permet de sélectionner un seul nœud XML à partir d'un nom. L'élément retourné est le premier à satisfaire à la condition.

```
// supprime le premier livre
books.Element("book").Remove();
```

Elements() & Attributes

Contrairement à Element() qui ne retourne qu'un seul nœud, la méthode Elements() retourne tous les nœuds satisfaisant le nom fourni en argument qui sont des enfants directs.

```
// supprime tous les livres enfants
```

```
books.Elements("book").Remove();
```

Attention : la méthode Elements() ne fonctionne que pour les enfants directs du nœuds.

Descendants() & Ancestors()

La méthode Descendants() fonctionne de la même manière que Elements() mais ne se limite pas seulement aux nœuds enfants directs. La méthode retournera tous les nœuds de l'arborescence XML.

```
books.Descendants("book").Remove();
```

Cas particulier : La méthode Descendants n'inclue pas le nœud en cours dans les résultats. Si c'est le comportement attendu, utiliser DescendantsAndSelf().

La méthode Ancestors est similaire à Descendants mais, si Descendants parcoure l'arbre vers le bas, Ancestors le parcoure vers le haut.

Modification de contenu existant

Pour modifier de contenu de nœuds on peut utiliser la méthode SetElementValue()

Par exemple :

```
listBooks.Element("book").SetElementValue("author", "Unknown");
```

Le code ci-dessus va mettre à jour la balise <Author> du premier nœud trouvé.

Validation avec LinqToXml

La validation d'un fichier XML est également possible et relativement simple à implémenter. C'est encore une fois l'objet XmlSchemaSet qui sert à encapsuler le schéma.
Il faut donc commencer par créer cet objet :

```
XmlSchemaSet sc = new XmlSchemaSet();
sc.Add("", "SampleLibraryXsd.xsd");
```

On utilisera ensuite la méthode Validate qui consommera cet objet et procèdera à la validation du XML.

```
XDocument doc1 = new XDocument( … création de l'arborescence )
bool errors = false;
doc1.Validate(schemas, (o, e) =>
{
    Console.WriteLine("{0}", e.Message);
    errors = true;
});
```

Benchmark

Avertissement :

Les résultats des tests de performances ci-dessous ne doivent pas être exploités de manière absolue. En effet, je ne suis moi-même pas capable de reproduire exactement chacun des résultats ci-dessous. La lecture d'un même fichier par une même méthode 2 fois d'affilée (en ayant fermé l'application après la première lecture) ne donne pas les mêmes résultats. Ceux-ci sont heureusement en général assez proches, mais il convient donc de rajouter un intervalle de tolérance de +/- 15% à chacun des résultats ci-dessous.

Il est plus intéressant de faire une analyse comparative des résultats afin de déterminer la performance des solutions entre elles.

Description du test :

Chaque lancement du test en écriture va déboucher sur la création d'un document XML contenant un nombre de nœuds défini en argument pour une solution donnée. Une fois cette opération effectuée, l'application se ferme. On mesure le temps que la méthode de lecture ou d'écriture prend à l'aide de log4net.

Dans le test d'écriture, le fichier à générer n'est pas le même d'une exécution à une autre. Le nombre de nœud ne change pas mais les données des nœuds sont générées aléatoirement et 2 exécutions successives d'un même test ne donneront pas lieu au même résultat.

En revanche, pour le test en lecture, c'est le même fichier qui est lu dans tous les tests pour une taille de fichier donnée.

Afin de rendre ce test plus intéressant, j'ai converti le nombre de nœuds en taille de fichier.

Résultats

Voici les résultats du benchmark en lecture :

LECTURE (ms)	10ko	100ko	1mo	5mo	10mo	100mo
XmlReader	16	20	45	145	386	3800
XmlDocument	20	24	90	566	1000	10000
Sérialisation	300	300	380	413	600	6000
Dataset	50	110	500	2070	4230	35000
LinqToXml	25	27	69	413	935	7200
XPath	20	24	73	250	535	9000

Et ceux du benchmark en écriture :

Ecriture (ms)	10ko	100ko	1mo	5mo	10mo	100mo
XmlReader	17	30	160	225	1350	5200
XmlDocument	20	40	260	705	2100	12500
Sérialisation	266	266	351	500	1000	8000
Dataset	31	159	288	600	2040	11500
LinqToXml	27	40	330	800	1800	9000

Analyse des résultats

On constate, sans surprise que XmlReader est la solution la plus rapide en lecture et Xmlwriter est la plus rapide en écriture. LinqToXml et la sérialisation utilisent ces objets pour leurs opérations de lecture écriture ; leurs performances ne peuvent pas être inférieures à celles XmlReader/XmlWriter.

Pour les petits documents, que ce soit en lecture ou en écriture, XmlDocument, XPath (lecture seule) et LinqToXml présentent de bonnes performances. En revanche la sérialisation est une opération couteuse pour ce type d'opération. La sérialisation est une solution à éviter pour gérer des fichiers de taille inférieure à 1mo.

A partir de 1 mo et pour les tailles au-delà, les différentes solutions techniques exhibent un comportement relativement consistant : la solution la plus rapide après XmlReader/XmlWriter est la sérialisation. LinqToXml n'est pas loin derrière. Il apparait assez clair que pour traiter de gros fichiers il faille utiliser XmlReader/XmlWriter, LinqToXml ou la sérialisation.

Une exception toutefois : XPath. XPath offre des performances en lecture très respectables jusqu'à 10mo mais semble perdre son avantage sur des fichiers plus volumineux 100Mo.

Conclusion générale

Le code qui a servi à ce projet est accessible sur simple demande à thomas.blotiere@gmail.com. Nous avons détaillé au cours de cet ouvrage les différents moyens de travailler avec des fichiers XML en .NET :

XmlReader/XmlWriter

Ces 2 classes présentent les meilleures performances de toutes les solutions proposées. En revanche, l'implémentation du code nécessaire à la lecture ou à l'écriture est fastidieuse et présente une maintenabilité faible.

XmlTextWriter et XmlTextReader

Ces 2 classes sont obsolètes et leur utilisation doit être proscrite. De nombreux bugs ont été trouvés et si elles sont toujours dans la package c'est pour une question de compatibilité avec les versions précédentes du Framework.

XmlDocument

XmlDocument est une classe qui a été introduite dès les débuts du Framework .NET. Même si elle n'est pas officiellement obsolète, il n'y a pas vraiment de raison de l'utiliser aujourd'hui.

Dataset

Les Dataset sont une des pires solutions en termes de performances pour gérer du XML de manière générale. Le développeur peut souhaiter les utiliser pour garder une certaine homogénéité dans du code les utilisant déjà cependant il vaut mieux se tourner vers d'autres solutions.

Sérialisation

La sérialisation est une alternative aux précédentes fonctionnant assez différemment de ces dernières. L'avantage principal est de ne pas à travailler avec les balises XML directement mais avec le modèle objet que le développeur maitrise bien mieux. La sérialisation présente en outre de très bonnes performances quand vient le besoin de traiter des fichiers supérieurs à 5mo.

LinqToXml

LinqToXml, la dernière-née des technologies .NET permettant de traiter du XML tient ses promesses. LinqToXml permet de lire et d'écrire des fichiers avec une syntaxe proche du SQL. Cette syntaxe est homogène avec les autres technologies Linq (LinqToSql, LinqToObjects).
C'est une technologie très puissante, maintenable et performante. A moins de travailler avec une ancienne version du Framework, c'est indéniablement la solution à considérer pour traiter des fichiers XML.

XPath

XPath est un bon moyen de lire ou parser des fichiers XML si le Framework de l'application est inférieur à 3.5. XPath permet de lire ou de retrouver des données dans un fichier XML de manière très rapide. Cette technologie ne permet pas l'écriture de fichier et il faut se cantonner à de la lecture seule.
La syntaxe des requêtes XPath est assez ésotérique et peut en dérouter plus d'un.
C'est pourquoi, pour des raisons de maintenabilité, je conseille l'utilisation de LinqToXml à XPath lorsque cela est possible.