Labo.net SUPINFO

http://www.labo-dotnet.com

WebServices

SUPINFO DOT NET TRAINING COURSE

Auteur : Matthieu Nicolescu

Version 1.1 – 3 novembre 2004
Nombre de pages : 38

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Présentation du Cours :

Ce document a pour but de vous présenter les Services Web dit « WebServices ». Nous allons donc dans un premier temps vous présenter les principes d'un WebService et ce qu'il peut apporter de nouveau au monde de l'entreprise. Nous vous présenterons ensuite le fonctionnement d'un WebService, ses principes et ses protocoles, et vous apprendrez enfin à créer votre premier WebService et à l'utiliser.

Pour suivre ce cours, vous devez avoir quelques notions du langage c# qui sera utilisé dans nos exemples. Vous devrez connaître aussi l'architecture .NET, c'est-à-dire les différentes couches du Framework et avoir programmé quelques applications Web et Windows Form.

Nous finirons ce cours par des notions avancées sur les WebServices qui ne sont pas utiles à la compréhension des WebServices mais seront utiles si vous voudrez utiliser la technologie des WebServices dans vos applications .NET ou autres. En effet, comme vous le verrez durant notre cours, les WebServices ne sont pas propriétaires et n'appartiennent donc pas à Microsoft.

2. Introduction aux WebServices :

Dans ce chapitre, nous allons vous présenter les WebServices : c'est-à-dire pourquoi les WebServices ont été créés et à quelle demande répond cette nouvelle technologie.
Nous verrons ensuite le fonctionnement d'un WebService puis nous finirons par deux exemples d'utilisation d'un WebService au sein d'une entreprise.

2.1.Présentation :

Auparavant pour mettre en place des applications distribuées, il fallait utiliser des technologies assez complexes telles que COM. Certes ces technologies étaient abordables pour un développeur, mais il fallait que le développeur passe du temps à établir un protocole de transmission.

Les WebServices sont alors apparus pour faciliter tout d'abord la tâche des développeurs. Avant toute chose, Microsoft, contrairement aux idées reçues, n'a pas créé les WebServices mais Microsoft a participé avec de grandes entreprises telles que IBM, SUN … à la standardisation des WebServices.

Ceci montre bien que la technologie des WebServices est une technologie très jeune, ce qui bien sûr peut être un inconvénient pour son intégration au sein des entreprises. Mais les plus grands spécialistes prévoient une « explosion » de l'utilisation des WebServices d'ici 2004 toutes technologies confondues (.NET, Java …).

2.2.Fonctionnement des WebServices :

L'un des plus gros avantages des WebServices est qu'ils reposent sur des protocoles standardisés. Cela permet que cette technologie soit exploitable par de nombreux langages.

En effet, les WebServices se reposent sur des protocoles tels que XML et http, donc SOAP. Pour vulgariser ce dernier protocole, SOAP permet de faire circuler du XML via du HTTP. Donc lorsqu'on interroge un WebService, les données sont transmises en XML via le port 80 (HTTP). Rien de plus simple ensuite pour le développeur de traiter l'information reçue.

A l'heure actuelle, la quasi totalité des langages informatiques supporte ces protocoles : ils disposent en effet de fonctions pour lire un fichier XML (Parseur XML). Donc un WebService développé sous une plateforme .NET peut être utilisé via le langage Perl, PHP, Python, Delphi, Cobol …

2.3.Exemples d'utilisations :

Nous allons voir à présent deux cas d'utilisation de WebServices sans rentrer dans les détails.

Pour notre premier exemple, prenons le cas simple d'une entreprise qui veut mettre en place un système de messagerie au sein de son Intranet, voir Internet par la suite. Elle pourra donc opter pour la technologie des WebServices en développant le Serveur de Messagerie en WebService .NET. L'avantage de cette solution et que l'application cliente pourra être développée sous n'importe quel

langage. L'entreprise pourra même développer plusieurs clients certains marchant sous Unix, d'autres sous Windows mais utilisant le même serveur de messagerie, c'est-à-dire le WebService qui se charge de la réception et de la transmission des messages. Dans notre cas, nous pourrons donc très bien avoir un client Windows qui communique avec un client Unix voir même un client Pocket PC !

Prenons à présent l'exemple d'une station de Météo qui veut revendre ses services. Elle veut par exemple partager une méthode avec une entreprise tierce : cette méthode permettrai de retrouver la température d'une région en entrant le Code Postal de celle-ci.
Rien de plus simple avec les WebServices ! La Station de Météo devra créer un WebService avec une WebMethod (nous verrons la signification de ce mot dans le chapitre suivant) qui retournera un entier (ou un float) suivant le Code Postal (donc un entier) qui sera transmis à la WebMethod.
Ensuite, si la station de Météo veut faire payer son Service, elle devra bien sûr protéger l'accès de la méthode par des moyens que nous aborderons par la suite.

3. Le protocole SOAP :

Nous allons décrire dans ce chapitre le protocole SOAP et ses concurrents (COM, CORBA …). Nous détaillerons ensuite la structure d'un message SOAP pour bien comprendre comment les informations sont émises et reçues à partir d'un WebService. En effet, comme nous l'avons vu dans le chapitre précédent, la technologie des WebServices repose en autres sur le protocole SOAP.

3.1. Présentation :

SOAP est un protocole adopté par le Consortium W3C. Le Consortium W3C crée des standards pour le Web : son but est donc de créer des standards pour favoriser l'échange d'information. Un standard veut tout simplement dire qu'il peut être accessible à tout le monde, et donc qu'il n'est pas propriétaire. Ce qui a pour conséquence qu'un protocole standard contrairement à un protocole propriétaire pourra être utilisé sous n'importe quelle plateforme.
Les spécifications du protocole SOAP sont disponibles à l'adresse suivante :
http://www.w3.org/TR/SOAP/

SOAP veut dire : Simple Object Access Protocol. Si l'on voulait traduire cette définition en français cela donnerait Protocole Simple d'Accès aux Objets. En effet, le protocole SOAP consiste à faire circuler du XML via du http sur le port 80. Cela facilite grandement les communications, car le XML est un langage standard et le port utilisé est le port 80, qui ne pose donc pas de problèmes pour les firewalls de l'entreprise, contrairement à d'autres protocoles.

Tout comme la technologie des WebServices, le protocole SOAP est très jeune. Le protocole SOAP a été crée en septembre 98, avec la version 0.9, par trois grandes entreprises : Microsoft, UserLand et DevelopMentor. IBM n'a participé au protocole SOAP qu'à partir de la version 1.1 en avril 2000. C'est cette même année que SOAP a été soumis au W3C.
Depuis septembre 2000, SOAP 1.1 est en refonte complète pour donner jour à la version 1.2 avec un groupe de travail de plus de 40 entreprises ! Parmi ces 40 entreprises, on retrouve bien sûr Microsoft, IBM mais aussi HP, Sun, Intel …)

3.2. Les autres protocoles… :

Jusqu'à la création du protocole SOAP, trois grands protocoles étaient utilisés.

3.2.1. COM et DCOM :

Les protocoles COM (Component Object Model) et DCOM (Distributed Component Object Model) ont été écrits par Microsoft et permettaient de faciliter la communication entre les composants Windows. Il y a eu un portage de COM sous Unix, mais ce protocole n'a été utilisé par que par des plateformes Windows et pour l'Intranet.
Les protocoles COM et DCOM n'étaient utilisés la plupart du temps que pour l'Intranet, car le port d'écoute des communications était statique : c'est-à-dire qu'on ne pouvait pas changer ce port et cela posait de gros problèmes de sécurité pour les entreprises qui voulaient utiliser ce protocole pour communiquer entre elles.

3.2.2. CORBA :

CORBA (Common Object Request Broker Architecture) a été créé par l'OMG (Object Management Group) pour faciliter la communication sous n'importe quelle plateforme. Ceci a été réalisé via un langage neutre de définition d'interface appelé IDL (Interface Definition Language) et un protocole commun de transport des données.
Malheureusement, les spécifications de ce protocole sont très denses et l'architecture est donc au final très lourde à déployer.

3.2.3. RMI :

RMI (Remote Method Invocation) était un protocole très simple à utiliser et très efficace mais le problème est que ce protocole ne fonctionnait que sous environnement Java.

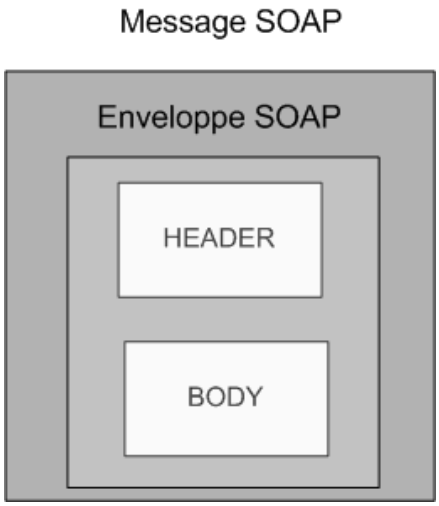
3.3.Messages SOAP :

3.3.1. Structure d'un message SOAP :

La syntaxe d'un message SOAP est décrite sur le site officiel W3C traitant du protocole SOAP : http://www.w3.org/TR/SOAP/

Tout d'abord un message SOAP est un document XML qui doit avoir la forme suivante :

- Une déclaration XML qui n'est pas obligatoire
- Une enveloppe SOAP qui est composée de :
  - Un en-tête SOAP (HEADER)
  - Un corps SOAP (BODY)



3.3.2. Messages SOAP dans l'exemple :

Voyons à présent un exemple concret d'un message SOAP de requête et le message de réponse.
Dans notre exemple, nous allons prendre une méthode qui retourne le nombre entré en paramètre. Ce type de méthode n'a bien sûr aucun intérêt dans la pratique, mais plus la méthode est simple, mieux vous comprendrez la structure d'un message SOAP.

Voici la signature de notre méthode :

```
int GetNombre (int Nombre);
```

La structure du message de requête sera du type :

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
xmlns:xsi=http://www.w3.org/1999/XMLSchema-instance
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
     <ns1:GetNombre
      xmlns:ns1="urn:MySoapServices">
```

```
            <param1 xsi:type="xsd:int">10</param1>
      </ns1:GetNombre>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Comme vous pouvez le voir, la première ligne du message SOAP contient une déclaration XML qui n'est pas obligatoire. L'enveloppe SOAP est ensuite déclarée via la balise <SOAP-ENV:Envelope>. Cette enveloppe est composée d'un corps (<SOAP-ENV:Body>).
Dans le corps de ce message, vous pouvez très bien voir notre méthode « GetNombre » et son paramètre qui est égale à 10 dans notre exemple.

Voyons à présent le message de réponse :

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsi=http://www.w3.org/1999/XMLSchema-instance
xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
     <ns1:GetNombreResponse
     xmlns:ns1="urn:MySoapServices"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <return xsi:type="xsd:int">10</return>
    </ns1:GetNombreResponse>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Le message de réponse a la même structure que celle du message envoyé. Ceci veut dire qu'il contient une déclaration XML, ainsi qu'une enveloppe SOAP composée d'un corps.
Dans le corps du message de réponse, nous pouvons toujours voir notre méthode « GetNombre » : le mot « Response » a été rajouté à la fin du nom de la méthode pour bien préciser qu'il s'agit du retour d'une requête sur la méthode.
La valeur renvoyée par la méthode « GetNombre » est égale à 10.
Dans notre exemple, l'enveloppe SOAP n'était pas composée d'un en-tête : en effet, l'en-tête d'un message SOAP est optionnelle et est utilisée pour transmettre des données d'authentification ou de gestion de sessions. Nous verrons plus loin en détail l'utilité de cet en-tête.

L'exemple que nous venons de voir est un exemple très simple. Pour décrire aux mieux le protocole SOAP, il faudrait voir des exemples avec des types plus complexes. Pour cela nous vous renvoyons sur le site officiel du protocole SOAP :
http://www.w3.org/TR/SOAP/

3.4.Sécurité et SOAP

La sécurité est un des grands défauts du protocole SOAP. En effet, lors de la création du protocole SOAP, des groupes ont voulu faire pression pour insérer dans le protocole SOAP des mécanismes de sécurité. Mais le projet a été abandonné, car le protocole SOAP devait rester facile à mettre en œuvre.
De plus, comme vous avez pu le voir dans notre section précédente, les messages SOAP peuvent être lus sans problèmes et il suffit donc qu'une tierce personne lise les messages SOAP émis et reçus par le WebService pour avoir l'intégralité des informations sans aucune difficulté.

Le cryptage devient donc nécessaire pour des services transmettant des données sensibles. Le protocole HTTPS, qui propose un chiffrement jusqu'à 128 bits, pourra donc être utilisé pour mettre un place un système de communication sécurisé.

Nous verrons dans un chapitre ultérieur les différents moyens qui peuvent être mis en œuvre pour mettre en place un système d'information sécurisé.

4. WSDL et WebMethod :

Avant de passer à la pratique, nous allons voir ensemble quelques termes techniques dont vous devrez connaître la signification avant de commencer à réaliser votre premier WebService.

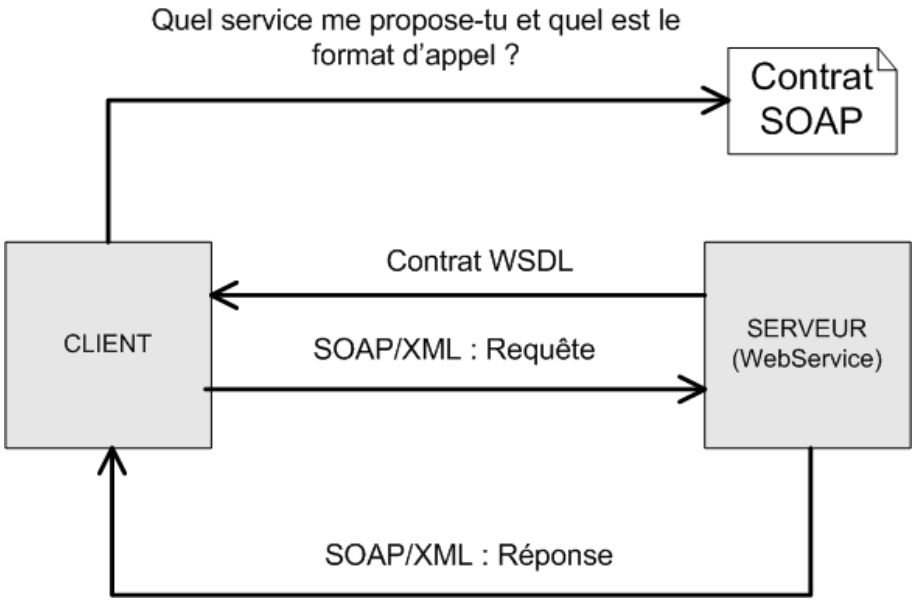
4.1.WSDL :

4.1.1. Présentation :

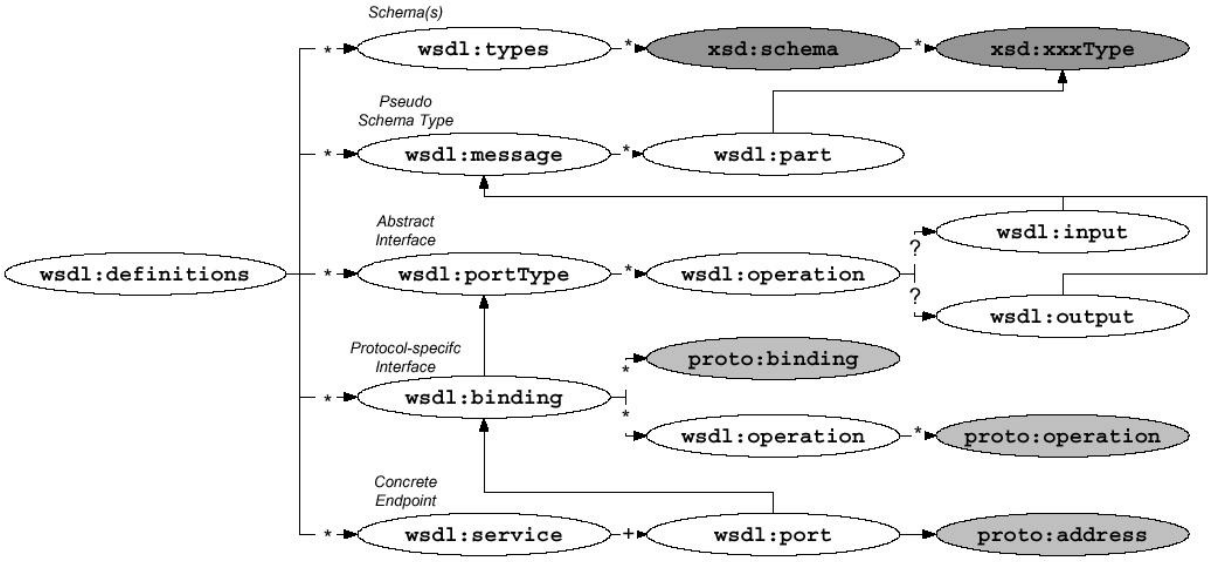
Il faut savoir avant toute chose que chaque WebService possède un fichier de spécification : c'est ce qu'on appelle la WSDL (Web Services Description Language). C'est à partir de ce fichier que le développeur pourra savoir comment interroger le WebService, quels sont ses différentes WebMethods, quels types ces WebMethods retournent t-elles …

La WSDL décrit en fait tout le fonctionnement d'un WebService et il est donc indispensable pour pouvoir utiliser un WebService. Ce fichier est structuré en XML et il est assez complexe quand le développeur ne connaît pas XML. Mais nous verrons par la suite que le Framework génère automatiquement la WSDL d'un WebService ce qui facilite grandement la tâche du développeur.

Enfin, pour que votre application puisse utiliser un WebService, il faut qu'elle dispose d'une classe Proxy appelé Proxy Web qui sera créée à partir du Contrat du WebService, c'est-à-dire le fichier de description WSDL.
Le Proxy va aider le client à savoir où trouver le WebService. Le Proxy contiendra également les détails des communications avec le WebService.
Nous verrons ultérieurement que l'utilitaire « wsdl.exe » fourni avec le Framework SDK .NET permettra de générer notre classe Proxy à partir d'un fichier de description WSDL.



4.1.2. Schéma fichier WSDL :



4.2.WebMethod :

Nous verrons dans le chapitre suivant que lorsque nous devons créer un WebService se nommant par exemple « MyWebService » nous devons créer une classe « MyWebService ». En effet tout est objet en .NET et tout code doit se trouver dans une classe.

Comme vous devez le savoir, une classe dispose de valeurs ainsi que de méthodes. Ceci ne change pas avec les WebServices, mais pour qu'une méthode soit utilisable via votre WebService, il faut donner l'attribut « WebMethod » à cette même méthode.

Examinons à présent le code suivant :

```
[WebMethod]
public string GetName(string Prenom)
{
return "toto";
}
private void Nothing()
{
}
```

Nous avons ici deux méthodes (Nous n'avons pas représenter ici la classe). La première méthode « GetName » possède l'attribut « WebMethod ». La méthode GetName est donc une WebMethod qui retourne un type « string » et qui prend type « string » comme argument.

Attention : Toutes WebMethods doivent être déclarées en public et ne peuvent pas être « private » ou « protected », sinon la WebMethod ne pourra pas être atteinte depuis l'extérieur.
La deuxième méthode n'est pas une WebMethod car elle ne possède pas l'attribut « WebMethod » et ne pourra donc pas être atteinte via le WebService.

5. WebService avec le Framework SDK :

Dans ce chapitre, nous allons créer notre premier WebService avec le Framework SDK, sans utiliser Visual Studio .NET.

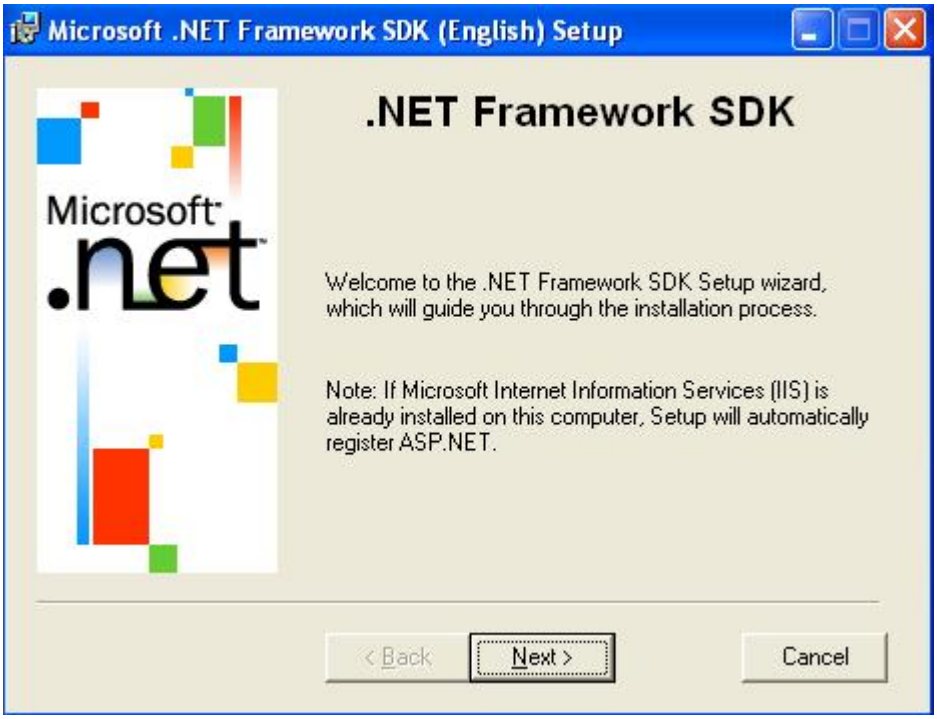
5.1.Présentation du Framework SDK :

Nous avons vu lors des chapitres précédents que les WebServices ne sont pas une technologie propriétaire. Il n'est donc pas nécessaire d'être sous une plateforme .NET pour pouvoir utiliser cette technologie.

Nous allons apprendre à créer dans cet essentiel votre premier WebService en .NET. Vous avez deux possibilités : utiliser Visual Studio .NET ou le Framework .NET SDK, à savoir bien sûr que Visual .NET comprend le Framework SDK. L'avantage du Framework SDK .NET est qu'il est totalement gratuit contrairement à Visual Studio .NET. Le Framework SDK comprend les compilateurs nécessaires pour compiler vos programmes VB.NET ou C#, les classes de bases ainsi bien sûr que tous les utilitaires tels que « wsdl.exe », « wincv.exe », « gacutil.exe »…

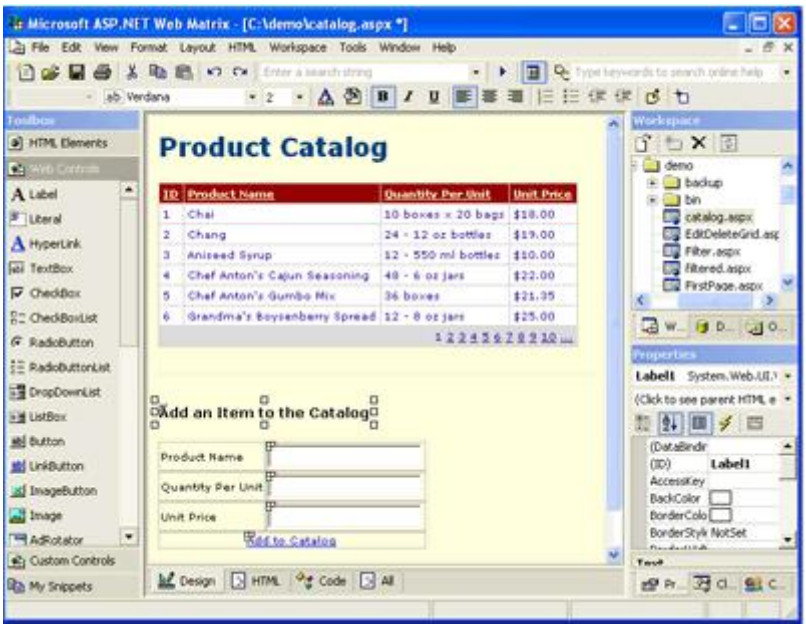
Vous pouvez télécharger le Framework SDK .NET sur le site de Microsoft à partir de l'adresse suivante :

http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml&frame=true



Le Framework SDK est totalement gratuit, mais il n'y a pas d'IDE (interface graphique) qui vous permettra d'éditer vos WebServices. Vous devrez donc tout faire avec votre éditeur de texte. Par contre, vous pouvez utiliser le logiciel « WebMatrix », logiciel totalement gratuit développé par Microsoft et qui permet d'éditer des Web Applications et des WebServices. Vous pouvez télécharger « WebMatrix » à l'adresse suivante :

http://www.asp.net/webmatrix/



5.2. Création d'un WebService :

5.2.1. Enoncé :

Pour notre premier WebService, nous allons commencer par un exercice simple. Nous allons faire un « WebService Calculette » que nous allons nommer « CalcService ». Nous allons implémenter deux méthodes au WebService « CalcService » :

- « Ajouter »
- « Soustraire »

La première méthode « Ajouter » permet de retourner la somme des deux entiers rentrés en paramètre :

```
public int Ajouter(int valeur1, int valeur2) ;
```

La deuxième méthode « Soustraire » permet de retourner la soustraction des deux entiers rentrés en paramètre :

```
public int Soustraire(int valeur1, int valeur2) ;
```

Nous n'implémenterons pas les autres opérations traditionnelles d'une calculatrice telles que « Diviser », « Multiplier »… car le principe reste le même.

5.2.2. Passons à la pratique… :

Il faut tout d'abord savoir qu'en .NET, l'extension des WebServices est « .asmx » : c'est-à-dire que tous les fichiers représentant un WebService ont l'extension « .asmx ». La première étape est donc d'éditer votre fichier en l'appelant « CalcService.asmx » et y insérer le code suivant :

```
<%@ WebService Language="c#" class="CalcService" %>
using System.Web.Services;

public class CalcService : System.Web.Services.WebService
      {
            [WebMethod]
            public int Ajouter(int valeur1,int valeur2)
            {
                  return valeur1 + valeur2;
            }

            [WebMethod]
            public int Soustraire(int valeur1,int valeur2)
            {
                   return valeur1 - valeur2;
            }
      }
```

La première ligne permet d'indiquer à l'environnement .NET, c'est-à-dire la CLR, qu'il s'agit d'un WebService qui est réalisé en C#.
Nous importons ensuite via le mot-clé « using » le namespace « System.Web.Services » pour éviter de retaper le chemin à chaque fois tout au long du programme.

Nous déclarons ensuite la classe « CalcService » qui représente notre WebService et qui dérive de la classe « WebService » du namespace « System.Web.WebService ». La classe « CalcService » pourra donc accéder à toutes les méthodes de la classe « WebService » ainsi qu'a ses attributs.
Dans la classe « CaclService », nous retrouvons nos deux méthodes : « Ajouter » et « Soustraire ». Comme indiqué dans le chapitre précédent, vous pouvez voir que nous assignons à ces deux méthodes l'attribut « WebMethod » qui permet d'indiquer que ces méthodes seront accessibles via notre WebService.

Vous avez à présent créé votre premier WebService !

5.3.Tester un WebService:

Vous pouvez à présent tester votre WebService en entrant l'adresse Web de celui-ci : en effet, le Framework génère automatiquement une interface Web de test pour votre WebService.
Nous traiterons de cette interface Web dans un chapitre ultérieur qui est très utile pour la phase de test avant d'implémenter votre WebService dans une application.

6. WebService avec Visual .NET :

6.1. Avantages de Visual .NET :

Utiliser Visual Studio .NET facilite le travail du développeur et rend plus productif le projet. En effet, lorsque vous commencez un nouveau WebService, Visual Studio .NET vous crée tous les fichiers nécessaires au bon fonctionnement de votre WebService et fournit quelques exemples de code.

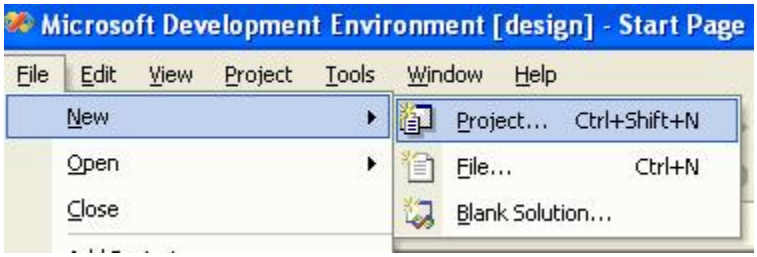
Il est vrai que Visual .NET n'est pas donné du point de vue financier, mais il y a une autre alternative qui est elle entièrement gratuite. Cette solution s'appelle « WebMatrix » et c'est l'éditeur d'applications Web et WebService de Microsoft. « WebMatrix » est bien sûr beaucoup plus léger que Visual .NET et n'apporte donc pas toutes ses fonctionnalisés.

6.2. Création d'un WebService :

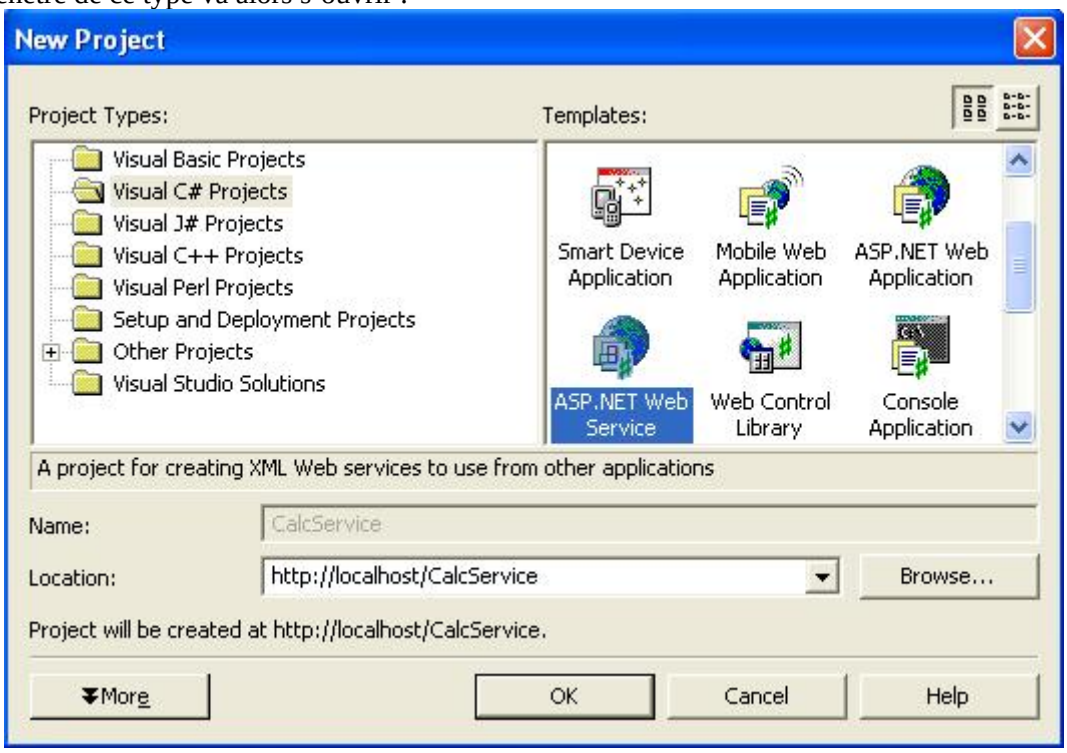
Nous allons donc créer le même WebService que dans le chapitre précédent, c'est-à-dire le WebService « CalcService », mais cette fois-ci avec Visual Studio .NET.

6.2.1. Utilisation de l'interface de VS .NET :

Lancez Visual Studio .NET. Une fois Visual lancé, Cliquez sur :
File -> New -> Project.

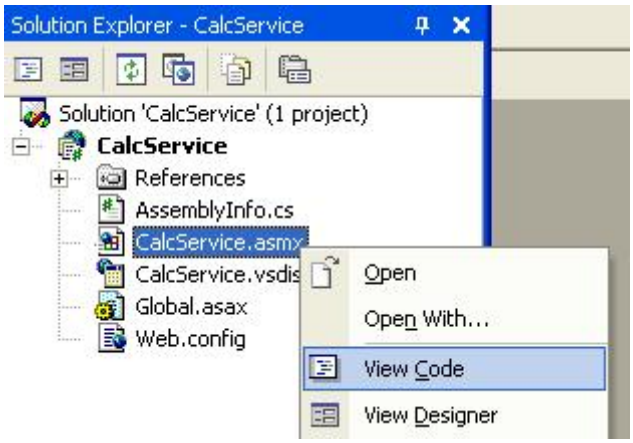


Une fenêtre de ce type va alors s'ouvrir :



Sélectionnez tout d'abord sur le Menu de gauche votre type de projet. Dans notre exemple, nous allons faire notre WebService en C#. Vous devez donc sélectionner « Visual C# Projects » puis dans le menu de droite « ASP.NET WebService ».
Nous allons ensuite donner un nom à notre WebService « CalcService ». Une fois la fenêtre validée, Visual .NET va créer tous les fichiers nécessaires à votre WebService.

Dans le « solution explorer » de votre projet, vous pouvez voir que Visual vous a créé plusieurs fichiers. Le plus important est le fichier «*.asmx » : c'est votre WebService. Sélectionnez alors ce fichier et cliquez sur « View Code » comme vous le montre l'image ci-dessous :



Vous pouvez alors éditer votre WebService…. Les autres fichiers vous sont connus, comme vous avez déjà sûrement programmé des applications Web ou Windows avec .NET, mais en voici une rapide description :

- « CalcService.asmx » : C'est le fichier qui représente votre WebService.
- « Web.config » : C'est le fichier de configuration de votre WebService.
- « AssemblyInfo.cs » : Ce fichier permet d'indiquer des informations sur votre assemblés et de la crypter.
- « Global.asax » : permet de mettre du code dans certains événements de votre application. Par exemple vous avez l'événement « Application_Start » qui est appelé lorsque l'application démarre, l'événement « Application_Error » qui est appelé lorsqu'il y a une erreur…

6.2.2. Accès au code d'un WebService :

Vous êtes à présent dans l'éditeur de code de votre fichier « CalcService.asmx », c'est-à-dire le WebService. Il vous suffit alors d'insérer la même classe que vous avez réalisée dans le chapitre précédent, comme vous le montre la figure de la page suivante.

Comme vous pouvez le voir sur la figure de la page suivante, le code est à quelques différences près le même. Nous avons rajouter dans cet exemple un namespace et importé quelques namespaces supplémentaires. A noter que ces namespaces ont été automatiquement inclus par Visual Studio .NET.

Il y a aussi un constructeur qui est dans notre classe qui n'était pas présent dans l'exemple du chapitre précédent. Ce constructeur permet d'appeler la méthode « InitializeComponent », méthode qui initialisera tous les composant invisibles que vous aurez inséré dans le designer de votre WebService. Vous pouvez en effet insérer par exemple un composant « Timer » à votre WebService, mais en aucun cas des composants visibles tels que « Label », «TextBox »…

Vous avez à présent créé votre premier WebService avec le Framework .NET SDK et avec Visual Studio .NET. Nous allons à présent voir comment nous pouvons tester notre WebService, atteindre la WSDL, et enfin créer une application utilisant notre WebService.

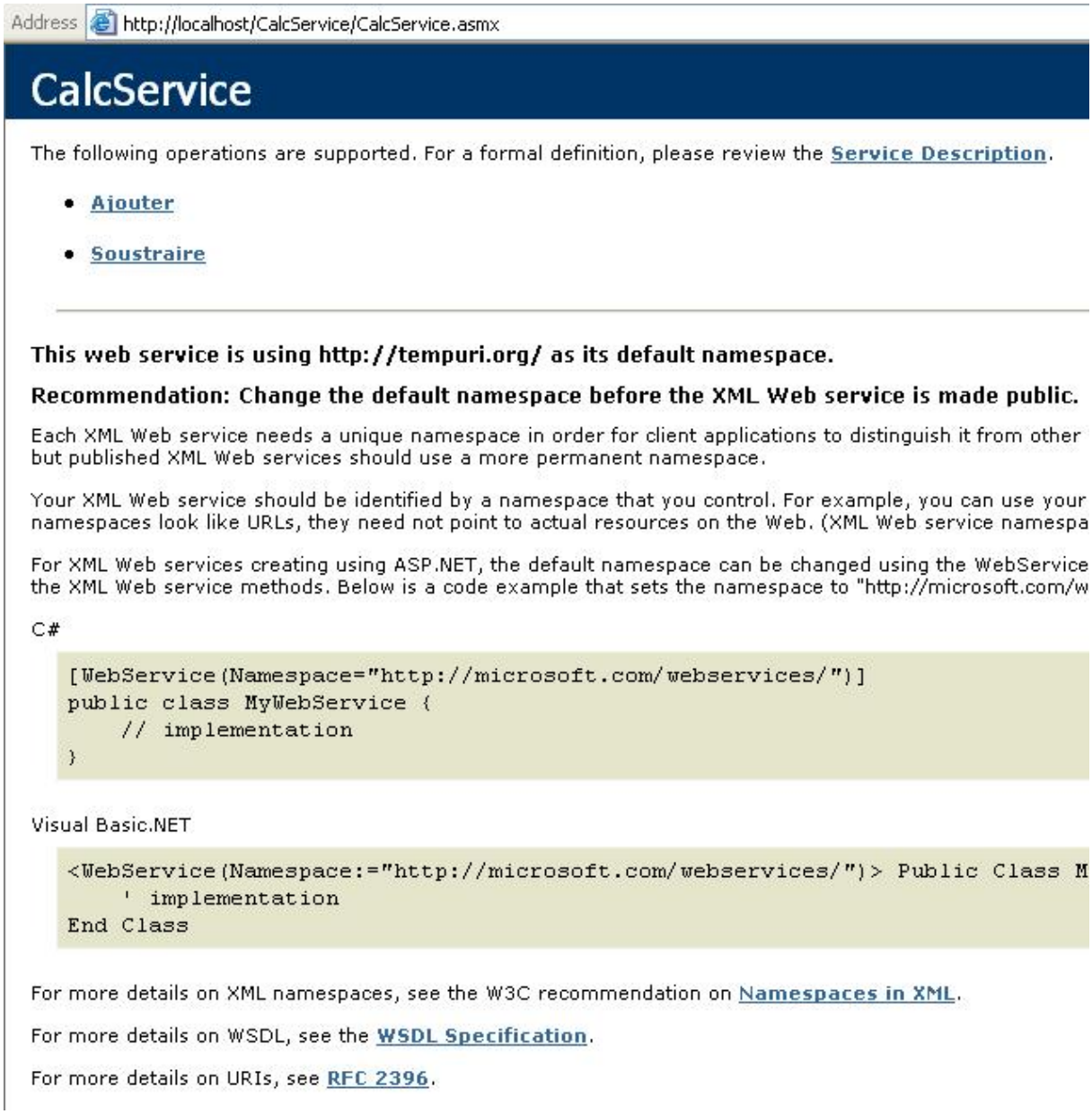
```
1  using System;
2  using System.Collections;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Diagnostics;
6  using System.Web;
7  using System.Web.Services;
8
9  namespace CalcService
10 {
11     /// <summary>
12     /// Summary description for Service1.
13     /// </summary>
14     public class CalcService : System.Web.Services.WebService
15     {
16         public CalcService()
17         {
18             //CODEGEN: This call is required by the ASP.NET Web Services Designer
19             InitializeComponent();
20         }
21
22         [WebMethod]
23         public int Ajouter(int valeur1,int valeur2)
24         {
25             return valeur1 + valeur2;
26         }
27
28         [WebMethod]
29         public int Soustraire(int valeur1,int valeur2)
30         {
31             return valeur1 - valeur2;
32         }
33
34
35         Component Designer generated code
61
62
63     }
64 }
65
```

7. Tester votre WebService :

7.1.Présentation Page de test :

Une fois notre WebService fini, nous pouvons le tester immédiatement, car le Framework génère une page qui permet de tester toutes ses méthodes. Pour accéder à cette page, il vous suffit d'entrer le chemin Web de votre fichier .asmx dans Internet Explorer ou autres… Si vous utilisez Visual Studio .NET, vous pouvez directement avoir accès à votre WebService en appuyant sur la touche « F5 ». Visual va alors compiler votre application et lancer Internet Explorer.

Vous tomberez alors sur une page de ce type :



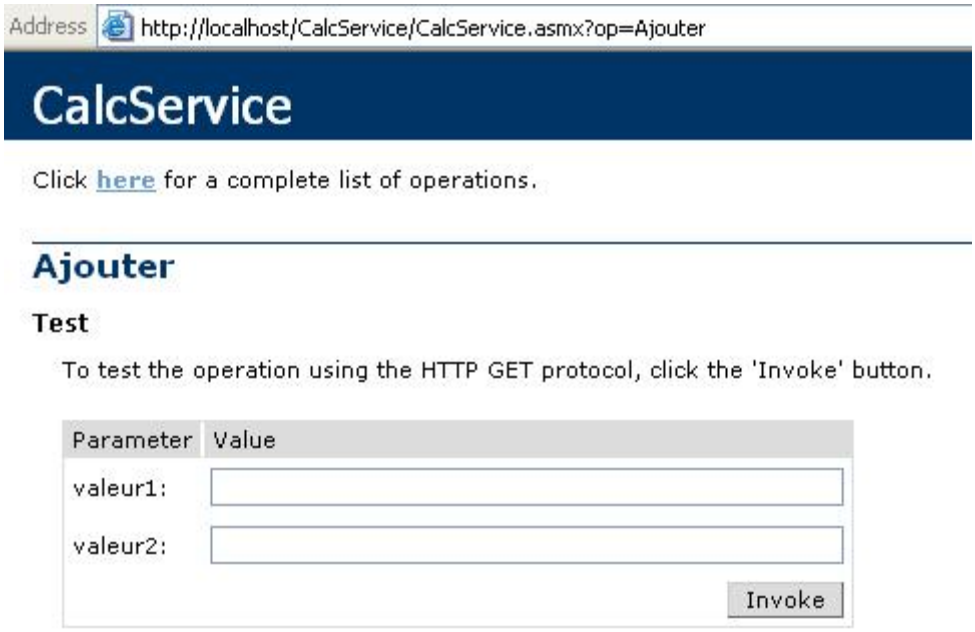
Notez tout d'abord que la page vous indique que votre WebService a un namesapce par défaut « http://tempuri.org ». Nous vous montrerons dans une section ultérieure comment changer le namespace de votre WebService.

Les deux méthodes de notre WebService, « Ajouter » et « Soustraire », sont présentes sur la page en tant qu'hyperlien. Vous allez pouvoir, en cliquant par exemple sur « Ajouter », tester la méthode et voir quelle méthode elle retourne.

7.2. Analyse de la méthode « Ajouter » :

7.2.1. Test de la méthode :

Je vous propose donc de cliquer sur « Ajouter » pour tester et analyser la méthode « Ajouter ». Vous devez alors avoir la page suivante qui apparaît :



Note : Nous n'avons pas affiché pour l'instant les informations qui se trouvent en bas, car nous parlerons de ces informations juste après le test de la méthode « Ajouter ».

Nous retrouvons sur la page d'information de la méthode « Ajouter » ses deux paramètres :

- valeur1
- valeur2

Il y a aussi un bouton « Invoke » qui va vous permettre d'appeler la méthode avec la valeur des paramètres que vous aurez insérés des les textbox appropriées.
Entrons par exemple la valeur « 10 » pour « valeur1 » et « 5 » pour « valeur2 ». Nous ne le savons pas encore mais théoriquement, si notre méthode fonctionne normalement, la méthode « Ajouter » devrait retourner la valeur « 15 ».

Pour savoir si votre méthode « Ajouter » retourne un résultat cohérent, cliquez sur le bouton « Invoke ».

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/">15</int>
```

La valeur retournée est bien sûr envoyée sous format XML et nous pouvons voir que cette valeur est bien égale à « 15 ». Notre méthode « Ajouter » a donc bien fonctionné.

7.2.2. Différents protocole d'accès :

Retournons à présent sur la page d'information de la méthode « Ajouter ». Vous pouvez voir en dessous du bouton « Invoke » plusieurs informations qui sont réparties en trois chapitres qui représentent en fait les différents protocoles d'accès au WebService. Nous n'avons vu jusqu'à présent qu'un seul protocole d'accès à un WebService, le protocole SOAP. Il existe en fait trois protocoles d'accès :

- SOAP
- HTTP GET
- HTTP POST

7.2.3. SOAP :

Le premier protocole traité par la page d'information de la méthode « Ajouter » est le protocole SOAP que nous avons détaillé dans un chapitre précédent.

Comme le montre la figure de la page suivante, vous pouvez voir deux messages. Le premier est le message de requête et le deuxième est le message de réponse.
Les deux messages sont bien constitués d'une enveloppe SOAP et d'un corps appelé « BODY ».

Le corps du message de requête contient bien :

- La méthode « Ajouter » : <Ajouter xmlns="http://tempuri.org/">
- Le premier paramètre : <valeur1>**int**</valeur1>
- Le deuxième paramètre : <valeur2>**int**</valeur2>

Le corps du message de réponse contient :

- La valeur de retour : <AjouterResult>**int**</AjouterResult>

SOAP

The following is a sample SOAP request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /CalcService/CalcService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/Ajouter"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/
  <soap:Body>
    <Ajouter xmlns="http://tempuri.org/">
      <valeur1>int</valeur1>
      <valeur2>int</valeur2>
    </Ajouter>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/
  <soap:Body>
    <AjouterResponse xmlns="http://tempuri.org/">
      <AjouterResult>int</AjouterResult>
    </AjouterResponse>
  </soap:Body>
</soap:Envelope>
```

7.2.4. HTTP GET :

Nous pouvons en effet interroger notre WebService directement à partir de son adresse HTTP et en ajoutant les paramètres nécessaires à la méthode :

HTTP GET

The following is a sample HTTP GET request and response. The **placeholders** shown need to be replaced with actual values.

```
GET /CalcService/CalcService.asmx/Ajouter?valeur1=string&valeur2=string HTTP/1.1
Host: localhost
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<int xmlns="http://tempuri.org/">int</int>
```

Comme indiqué dans la figure précédente, il nous suffit d'entrez l'adresse « http://localhost/CalcService/CalcService.asmx/Ajouter?valeur1=10&valeur2=5 » pour pouvoir interroger notre méthode :

7.2.5. HTTP POST :

Vous pouvez accéder au WebService via une simple requête POST à partir d'un formulaire :

HTTP POST

The following is a sample HTTP POST request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /CalcService/CalcService.asmx/Ajouter HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length

valeur1=string&valeur2=string
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<int xmlns="http://tempuri.org/">int</int>
```

7.3.WSDL :

Vous pouvez accéder au fichier de description de votre WebService en ajoutant « ?WSDL » à l'adresse initiale de votre WebService :

Address http://localhost/CalcService/CalcService.asmx?WSDL

```
<?xml version="1.0" encoding="utf-8" ?>
- <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:soap="http://schemas
    xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="http://tempuri.org/" xmlns:soape
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/" xmlns:mime="http://schemas.x
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      - <s:element name="Ajouter">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="1" maxOccurs="1" name="valeur1" type="s:int" />
              <s:element minOccurs="1" maxOccurs="1" name="valeur2" type="s:int" />
            </s:sequence>
          </s:complexType>
        </s:element>
      - <s:element name="AjouterResponse">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="1" maxOccurs="1" name="AjouterResult" type="s:int" />
            </s:sequence>
          </s:complexType>
        </s:element>
```

Nous n'avons bien sûr pas affiché tout le fichier, car un fichier WSDL est très long même pour un WebService très simple.
Mais la complexité d'un fichier WSDL ne gêne en aucun cas le développeur, car ce fichier de description est automatiquement généré par le Framework.
La WSDL permettra donc aux développeurs de savoir comment utiliser le WebService, c'est-à-dire quelles sont ses méthodes, comment les interroger …

7.4.Personnalisation du WebService :

7.4.1. Changement du namespace :

Si vous n'avez pas attribué de namespace à votre WebService, un namesapce par défaut lui sera attribué et la page d'accueil de test vous conseillera alors de donner un nom à votre WebService :

This web service is using http://tempuri.org/ as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

Vous devez alors attribuer un namepsace à votre WebService. Pour notre exemple, nous le nommerons « http://www.labo-dotnet.com ».
Pour arriver à cela, il vous suffit d'ajouter l'attribut suivant à la classe de votre WebService :

```
[WebService(Namespace="http://www.labo-dotnet.com")
```

7.4.2. Description :

Vous pouvez ajouter une description à vos méthodes et à votre WebService. Ces descriptions seront alors incluses dans le fichier de description WSDL et affichées sur la page de test du WebService.

Pour ajouter une description de la méthode « Ajouter », il vous suffit de détailler l'attribut « WebMethod » comme ceci :

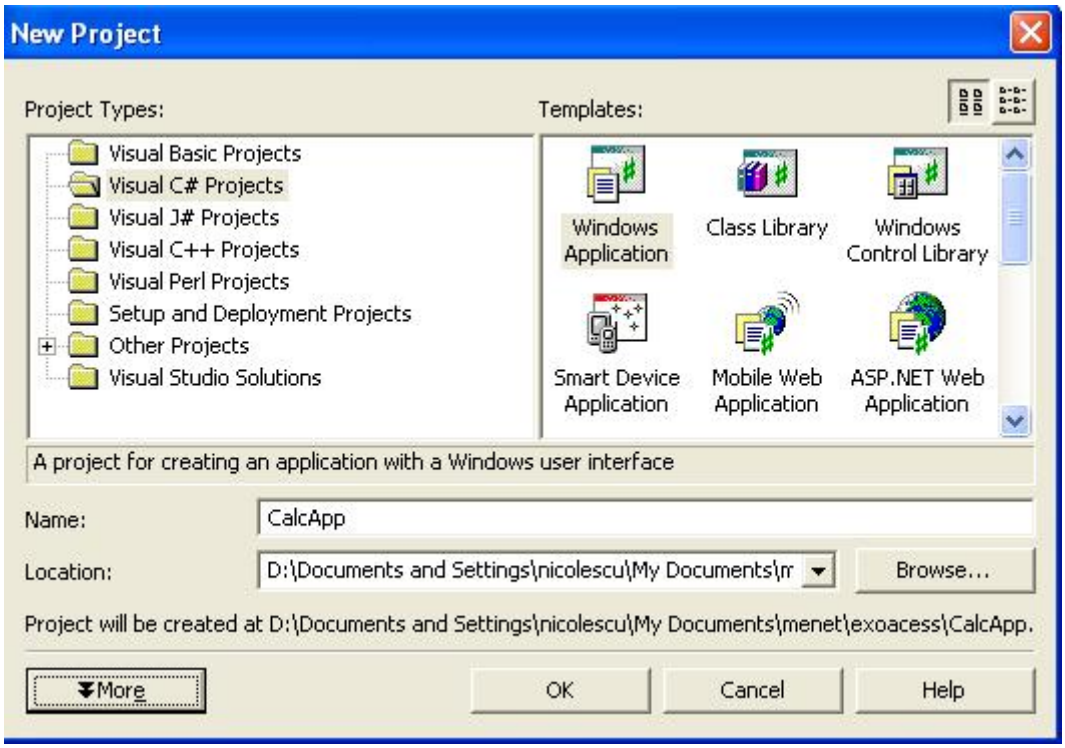
```
[WebMethod (Description="Description de la méthode …")]
public int Ajouter(int valeur1,int valeur2)
...
```

8. Créer une application cliente :

Nous allons dans une chapitre créer une application Windows utilisant notre WebService « CalcService ». Il est donc indispensable que vous ayez un minimum de connaissances en programmation Windows .NET. Dans le cas contraire, je vous renvois à l'essentiel « Applications Windows .NET ».

8.1. Création de l'application :

Nous allons créer une application Windows « CalcApp » qui utilisera la méthode « Ajouter » de notre WebService.

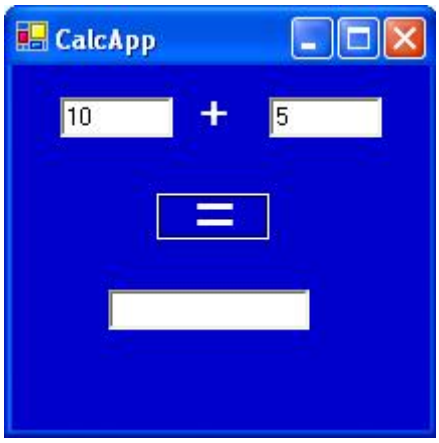


8.2. Interface :

Pour notre exemple, nous allons réaliser une simple Form contenant :

- « txtValeur1 » : Textbox contenant la valeur du premier paramètre de la méthode « Ajouter ».
- « txtValeur2 » : Textbox contenant la valeur du deuxième paramètre de la méthode « Ajouter ».
- « txtResultat » : Textbox contenant le résultat qui sera envoyé par la méthode « Ajouter ».
- « btnAjouter » : Bouton permettant d'appeler la méthode « Ajouter ».

L'interface reste simple comme vous pouvez le voir :



8.3.Classe Proxy :

8.3.1. Présentation :

Vous voulez à présent utiliser le WebService « CalcService » dans votre application Windows. Il faut vous créer une classe Proxy qui va vous permettre de faire la relation entre votre application et le WebService.
Pour générer votre classe Proxy, vous avez deux solutions :

- WSDL.EXE
- Solution intégrée de Visual .NET « Add Web Reference ».

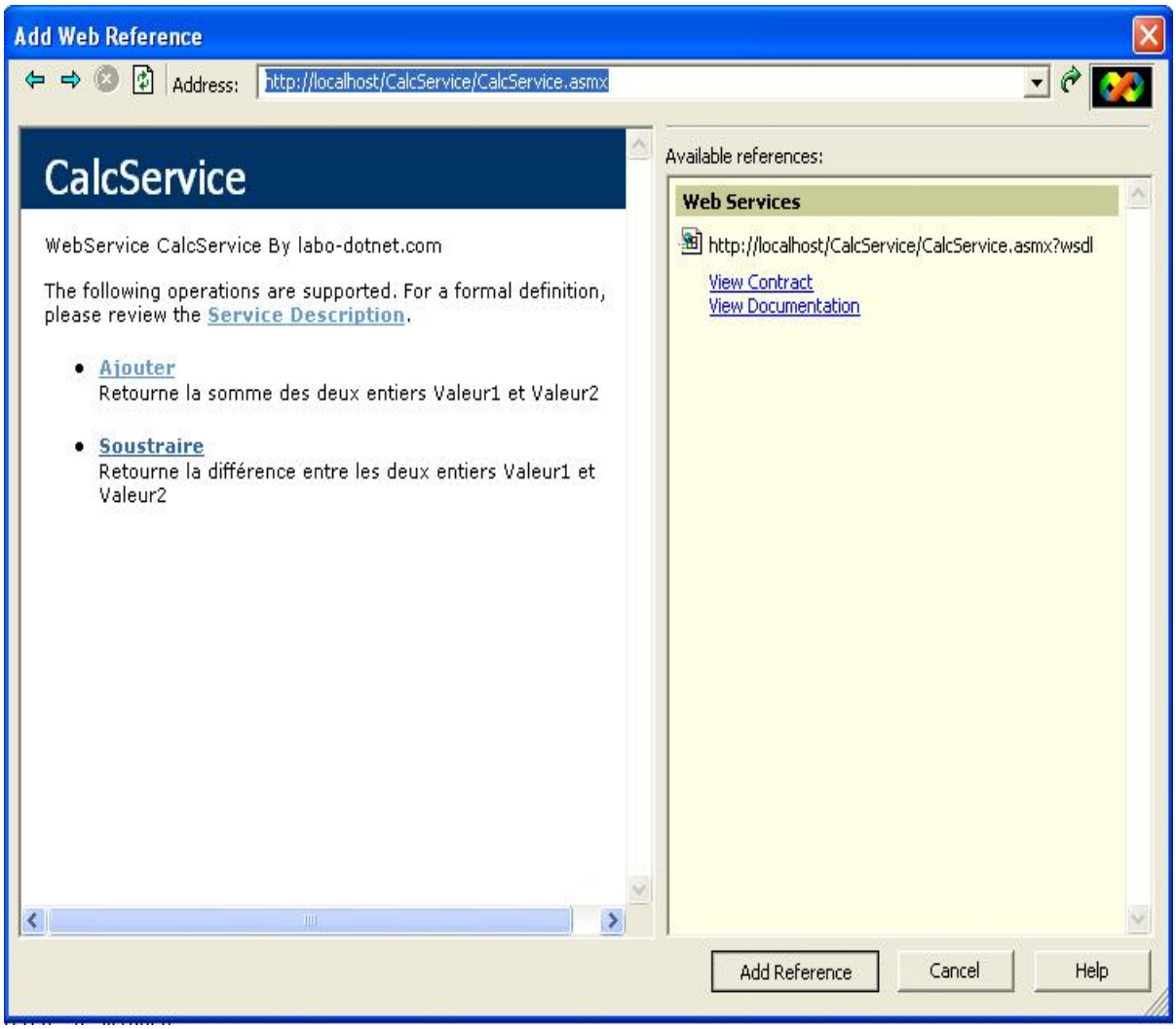
L'utilitaire « wsdl.exe », qui est fournit avec le Framework SDK, va vous permettre de générer votre classe Proxy à partir d'un fichier de description WSDL. La solution intégrée de Visual .NET utlise elle aussi « wsdl.exe », mais vous évite de passer en ligne de commande pour générer votre classe Proxy. En effet, Visual .NET va générer automatiquement tous les fichiers nécessaires pour que votre application puisse communiquer avec le WebService.

Nous verrons dans le chapitre suivant comment utiliser l'utilitaire « wsdl.exe » en détail…

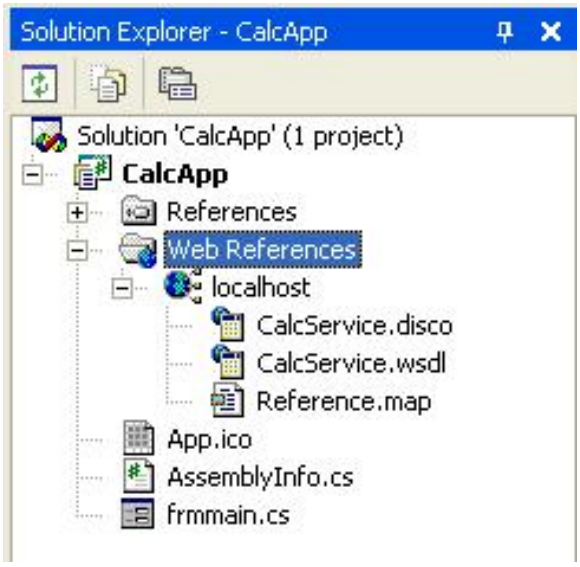
8.3.2. Génération de classe Proxy :

Nous allons donc utiliser Visual Studio .NET pour générer la classe Proxy du WebService « CalcService ».
Pour cela, allez dans le « Solution Explorer » et ajouter une référence Web :

Un Explorateur va alors s'afficher. Entrez l'adresse du WebService et cliquez sur « Add Reference » :



Visual .NET va alors créer plusieurs fichiers dans un dossier « Web References » :



Notre WebService sera accessible dans le namepsace « localhost ». Dans le dossier « localhost », nous trouvons trois fichiers :

- « CalcService.wsdl » : Fichier de description du WebService « CalcService ».
- « Reference.cs » : Fichier représentant la classe Proxy.
- « Reference.map »

8.4.Appel au WebService :

Nous voulons à présent appeler la méthode « Ajouter » lorsqu'on clique sur le bouton « = » de l'interface.
Voici les différentes étapes :

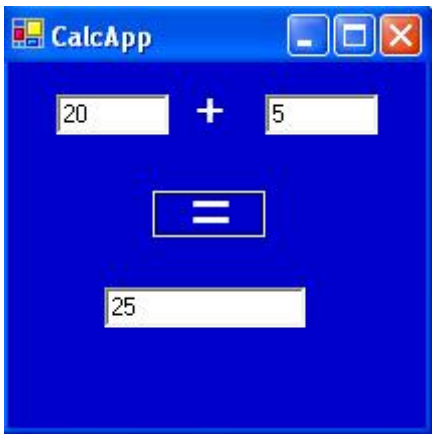
- Créer une nouvelle méthode et associer cette méthode à l'événement « Click » du bouton.
- Créer une instance de la classe « CalcService » qui représente notre WebService. Attention ! Comme indiqué ci-dessus, par défaut le WebService se trouvera dans le namesapce « localhost »
- Appeler la méthode « Ajouter » à partir de l'instance que vous venez de créer et insérer le résultat dans une TextBox.

Et voici le code correspondant à l'événement « Click » du bouton :

```
private void btnAjouter_Click(object sender, System.EventArgs e)
{
   //On crée une instance de la classe CalcService
     localhost.CalcService myWebService = new localhost.CalcService();

     //On appelle la méthode "Ajouter"
     //et on insère le résultat dans txtResultat
     txtResultat.Text =
myWebService.Ajouter(int.Parse(txtValeur1.Text),int.Parse(txtValeur2.Text))
.ToString();
}
```

L'application est à présent terminée et nous pouvons la tester :



« 20 » plus « 5 » font bien « 25 » !

Note : Il est bien sûr inutile de passer par un WebService pour effectuer ce type d'opération.

9. Application Cliente avec le Framework SDK :

Nous allons voir dans ce chapitre quels sont les différents utilitaires pour créer une application cliente utilisant un WebService sans utiliser Visual Studio .NET. Dans un premier temps, nous verrons comment générer une classe Proxy via l'utilitaire « wsdl.exe », puis comment compiler une application client via le compilateur C# « csc ».

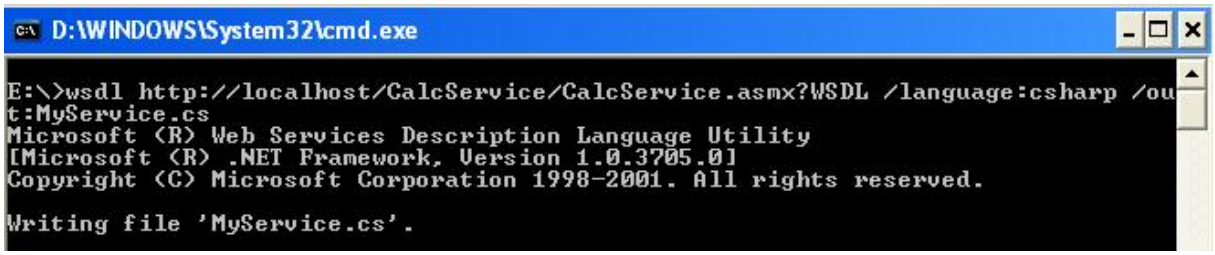
9.1. « WSDL.EXE » :

L'utilitaire « wsdl.exe » est un programme qui s'utilise en ligne de commande et qui permet de générer une classe Proxy à partir d'un fichier de description WSDL.
Il suffit en effet d'entrer en paramètre l'adresse du fichier de description du WebService et « wsdl.exe » va automatiquement générer la classe Proxy du WebService.
« wsdl.exe » peut être utilisé avec plusieurs options dont les principales sont :

- « **/language:<language>**» : Permet d'indiquer en quel langage sera généré la classe Proxy. Par défaut, le langage généré est le C#. Si vous voulez générer votre classe Proxy en VB .NET, il faudra indiquer « /language:VB ».
- « **/out :<filename>**» : Permet d'indiquer le nom de fichier de votre classe Proxy. Par défaut, le nom du fichier sera celui du nom du WebService. Par exemple, si vous avez un WebService « CalcService », le fichier généré par défaut sera alors « CalcService.cs ».
- « **/namespace:<namespace>** » : Permet d'indiquer dans quel namespace se trouvera votre Classe Proxy.
- « **/protocol:<protocol>** » : Permet d'indiquer le protocole qui sera utilisé. Les différents protocoles d'accès à un WebService sont « SOAP », « HtppGet », « HttpPost ». Par défaut, c'est le protocole SOAP.
- « **/nologo** » : Permet d'enlever le banner.

Pour générer la classe Proxy du WebService « CalcService » de notre chapitre précédent, il faudra donc entrer :

```
wsdl http://localhost/CalcService/CalcService.asmx?WSDL /language:csharp
/out:MyService.cs
```



9.2.Compilation de la classe Proxy :

Une fois la classe Proxy générée, il vous faut à présent compiler la Classe Proxy pour pouvoir l'intégrer ensuite à un projet.
Pour cela, nous allons utiliser le compilateur C# « csc » :

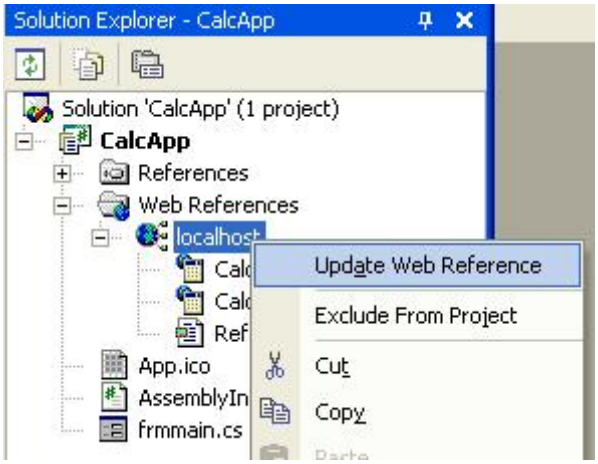
```
csc MyService.cs
/target:library
/out:MyService.dll
/reference:System.dll,System.Web.Services.dll,System.Xml.dll
```

10. Optimiser un WebService :

10.1. Appel Asynchrone :

Lorsque vous appelez une méthode d'un WebService à partir d'une application, l'application se figera tant que la méthode du WebService ne retournera pas de réponse. Cela peut poser beaucoup de problèmes. En effet, imaginez que la méthode de votre WebService effectue un calcul complexe qui prend du temps. Lorsque vous appellerez cette méthode via votre application, l'application se figera tant que la « WebMethod » ne sera pas terminée.

Vous pouvez faire un test en ajoutant une boucle infinie dans la méthode « Ajouter » du WebService « CalcService » des chapitres précédents. Une fois votre WebService modifié, il vous faut mettre à jour la classe Proxy de votre application Windows :



Lorsque vous lancerez votre application et appelez la méthode « Ajouter » du WebService « CaclService », l'application se figera !

Pour résoudre ce problème il suffit d'appeler la méthode « Ajouter » en mode asynchrone. Plutôt que d'appeler directement la méthode, on va passer par une autre méthode qui a été généré automatiquement dans la classe Proxy. Ces méthodes commencent toujours par le préfixe « Begin » suivi du nom de la méthode. Donc dans notre cas, nous allons appeler la méthode :

- **BeginAjouter**

La méthode « BeginAjouter » va tout d'abord prendre comme premiers arguments les deux valeurs « valeur1 » et « valeur2 ». Pour l'instant la signature de la méthode « BeginAjouter » ne change pas de la méthode « Ajouter ».
Mais la méthode « BeginAjouter » va prendre deux autres arguments. Le premier sera l'adresse d'une méthode de callback et le second sera l'instance du WebService « CalcService ».
La méthode « BeginAjouter » va en fait créer un nouveau Thread puis va rendre la main pour que l'application ne se fige pas. C'est dans ce Thread que l'on appellera la méthode « EndAjouter » pour pouvoir interroger la méthode « Ajouter ».

Voici à quoi va ressembler à présent le code de l'événement « Click » du bouton :

```
private void btnAjouter_Click(object sender, System.EventArgs e)
{
```

```
     //On crée une instance de la classe CalcService
     localhost.CalcService myWebService = new localhost.CalcService();

     //On appelle la méthode BeginAjouter
     myWebService.BeginAjouter(int.Parse(txtValeur1.Text),
                                                int.Parse(txtValeur2.Text),
                                                new
AsyncCallback(AjouterThread),
                                                myWebService);
}
```

Il faut donc à présent créer la méthode « AjouterThread » qui sera appelée par la méthode « BeginAjouter » :

```
private void AjouterThread(IAsyncResult state)
{
         //On crée une instance de la classe CalcService
         localhost.CalcService myWebService = (localhost.CalcService)
         state.AsyncState;
         //On appelle la méthode EndAjouter
         txtResultat.Text = myWebService.EndAjouter(state).ToString();
}
```

Vous pouvez tester à présent votre Application et vous verrez alors que votre application ne se figera plus.

10.2. Mise en cache :

Nous pouvons mettre une méthode d'un WebService en cache. Mais la question est « Pourquoi » ? Lorsque vous avez une méthode dont le résultat ne changera pas souvent et que cette même méthode demande beaucoup de ressource, il sera judicieux de mettre cette méthode en cache. C'est-à-dire que nous allons mettre le résultat de la méthode en cache pendant un temps x et la méthode qui demandait beaucoup de ressource ne sera plus appelée pendant ce temps.

Pour mettre une méthode en cache, il suffit de détailler l'attribut « WebMethod » :

```
[WebMethod(CacheDuration=30)]
```

Dans cet exemple, la méthode sera mise en cache pendant 30 secondes.

11. Sécuriser un WebService :

Vous savez à présent créer et utiliser un WebService, mais un autre problème se pose à présent. Ce problème c'est la sécurité de votre WebService. En effet, dans certains cas, vous ne voudrez pas que votre WebService soit accessible par tout le monde.
Nous allons donc voir comment sécuriser un WebService de différentes manières.

11.1. Authentification Windows :

Nous allons voir dans un premier temps comment sécuriser un WebService à l'aide de l'authentification Windows. Nous allons tout d'abord créer un nouvel utilisateur « UserTest » avec comme mot de passe « UserPassword ».

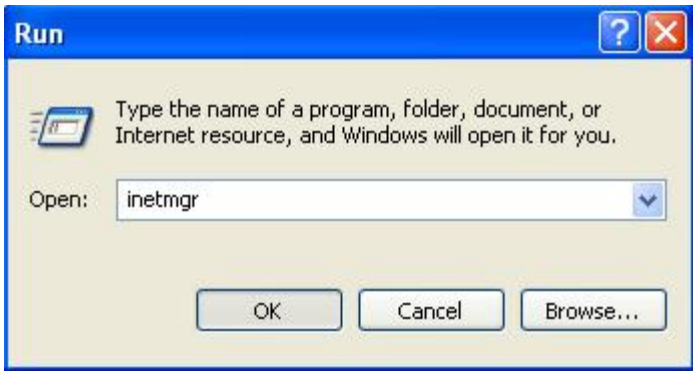
Utiliser l'authentification Windows intégrée pour les WebServices se fait de la même manière que pour les applications Web ASP.NET. Il faut donc indiquer au fichier de configuration de votre WebService que vous utilisez l'authentification Windows :

```
<authentication mode="Windows" />
```

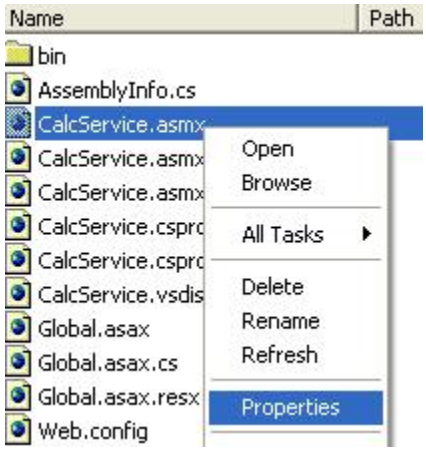
Il faut ensuite indiquer que seul l'utilisateur « UserTest » pourra accéder au WebService :

```
<authorization>
<allow users="UserTest" />
</authorization>
```

Vous devez indiquer ensuite à votre Serveur Web IIS que seul l'authentification Windows est autorisée. Pour cela lancez la console IIS « inetmgr » :

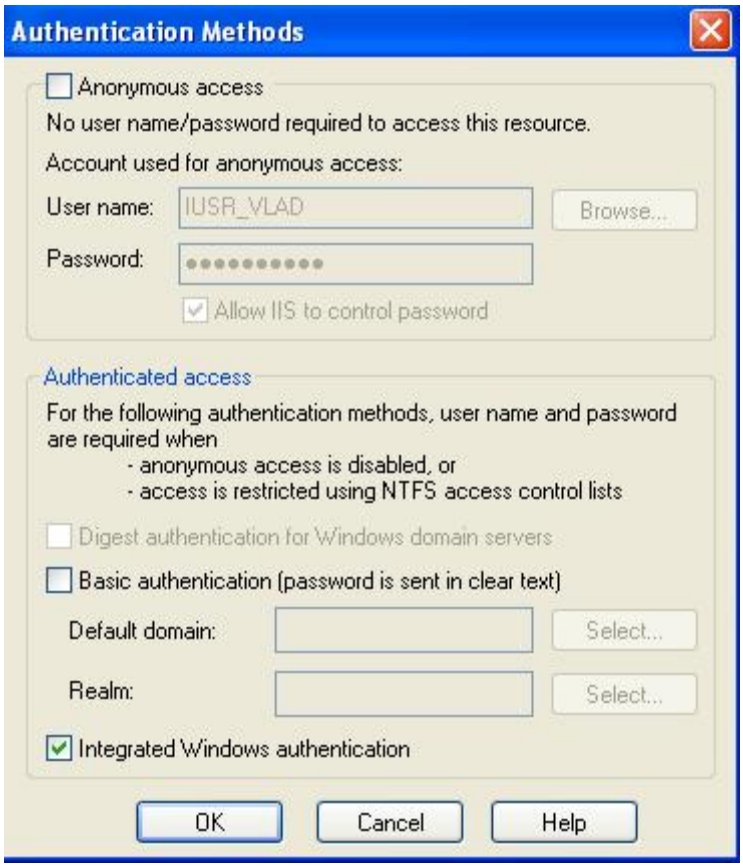


Allez ensuite dans les propriétés représentant votre WebService :



Dans l'onglet Sécurité, éditez les propriétés d'accès à votre WebService :

Une fenêtre va alors s'ouvrir : activez l'authentification Windows intégrée et désactivez « Connexion anonyme » :



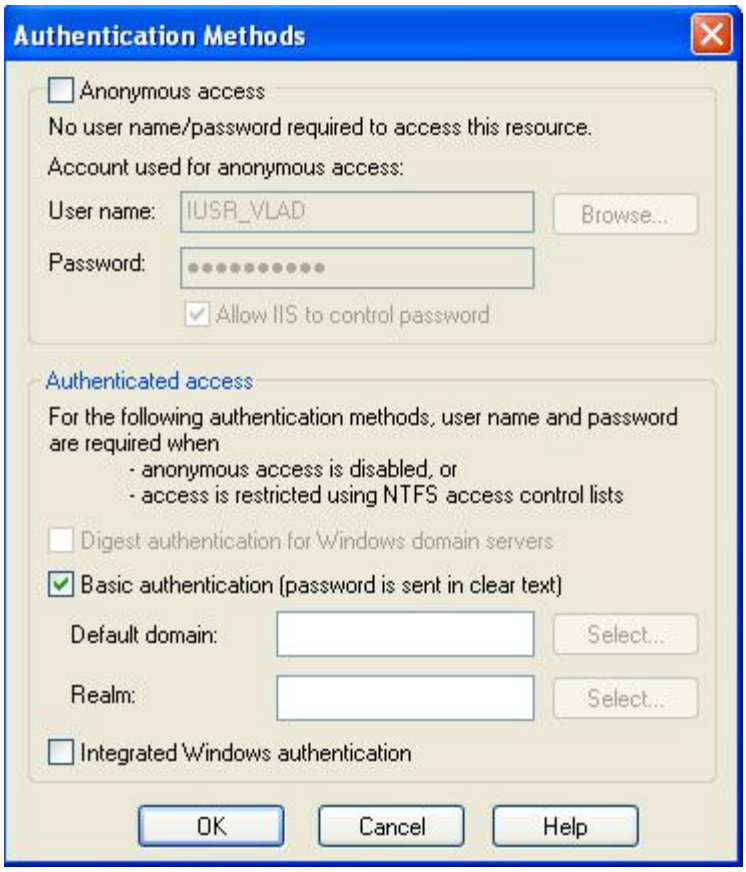
Le WebService est à présent sécurisé et seul l'utilisateur « UserTest » pourra accéder au WebService.

11.2. Authentification de base :

L'authentification Windows fonctionne correctement si on connaît l'utilisateur qui lance l'application utilisant le WebService.

Mais par exemple dans le cas d'une application Web, l'utilisateur exécutant l'application Web est la plupart du temps l'utilisateur « aspnet ». Il faut donc trouver un moyen de préciser l'utilisateur et son mot de passe à partir du code. Nous allons utiliser l'authentification de Base : « Basic Authentification ».

Pour cela, il vous faut retourner dans la fenêtre Sécurité de votre WebService et activer l'authentification de base en désactivant l'authentification Windows et anonyme.



Il faut à présent indiquer dans votre code lorsque vous créez une instance du WebService quel utilisateur appellera le WebService. Pour cela, nous allons utiliser la propriété « Credentials » de la classe Proxy de la manière suivante :

```
myWebService.Credentials = new
System.Net.NetworkCredential("UserTest","UserPassword");
```

Comme vous l'avez compris, le premier argument de « NetworkCredential » est le nom de l'utilisateur et le deuxième est son mot de passe.
Une fois cette ligne entrée, vous pouvez utiliser l'instance de votre WebService de la même manière que si votre WebService n'était pas sécurisé.

11.3. Authentification personnalisée :

11.3.1. Présentation :

L'authentication Windows et l'authentification basique sont parfaites pour des solutions Intranet : en effet, tous les utilisateurs Intranet auront la plupart du temps un compte sur le contrôleur de domaine de l'entreprise et il suffira de vérifier l'existence du compte et la validité du mot de passe via l'authentification Windows.

Mais pour des solutions Internet, nous n'allons pas créer un compte machine pour chaque utilisateur Internet ! La plupart de temps, les informations sur les utlisateurs Internet se trouvent dans une base de données. Dans ce cas, l'authentification personnalisée est la plus approprié.
L'authentification personnalisée va nous permettre d'ajouter des informations supplémentaires dans les messages SOAP qui seront envoyés par le WebService. Dans le chapitre consacré au protocole SOAP, nous avons vu qu'une enveloppe SOAP était constituée d'un en-tête et d'un corps. Or nous n'avons jamais mis d'en-tête dans nos messages SOAP jusque là, car nous n'en avions pas eu besoin.

11.3.2. En-tête SOAP :

Pour l'authentification personnalisée, nous allons utiliser les en-têtes d'un message SOAP pour pouvoir stocker le nom d'utilisateur et son mot de passe.
Pour créer un en-tête SOAP, nous allons devoir créer une classe dans le WebService qui dérive de « SoapHeader » avec deux champs public :

- UserName
- Password

Puis nous allons créer une instance de cette classe dans la classe représentant votre WebService. Avant toute choses, il faut tout d'abord déclarer dans vos fichiers le namespace « `System.Web.Services.Protocols` » :

```
using System.Web.Services.Protocols;
```

Voici à présent le code de la classe « AuthHeader » qui va représenter l'en-tête de nos messages SOAP :

```
public class AuthHeader : SoapHeader
{
      public string Username;
      public string Password;
}
```

Nous allons ensuite créer une instance de la classe « AuthHeader » dans la classe représentant notre WebService :

```
public AuthHeader sHeader;
```

Pour chaque WebMethod du WebService, il vous faut indiquer si le message correspondant à la méthode contient un en-tête. Cela va être fait à partir de l'attribut « SoapHeader ».

```
[SoapHeader("sHeader")]
```

Pour vérifier si l'en-tête de la méthode contient bien les deux champs « UserName » et « Password », vous pouvez aller sur la page de test du WebService et cliquer sur la méthode. Vous pourrez voir alors les détails du message SOAP de requête et le message SOAP de réponse.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSch
  <soap:Header>
    <AuthHeader xmlns="http://www.labo-dotnet.com">
      <Username>string</Username>
      <Password>string</Password>
    </AuthHeader>
  </soap:Header>
  <soap:Body>
    <Ajouter xmlns="http://www.labo-dotnet.com">
      <valeur1>int</valeur1>
      <valeur2>int</valeur2>
    </Ajouter>
  </soap:Body>
</soap:Envelope>
```

Dans l'exemple ci-dessus, vous pouvez voir que l'en-tête contient bien les champs « Username » et « Password ».

11.3.3. Application Cliente :

Il faut à présent indiquer à l'application cliente de créer un en-tête SOAP sinon l'appel de la méthode causera une erreur suivante :

Pour ajouter un en-tête, nous allons créer une instance de la classe « SoapHeader » qui est déclarée dans la classe Proxy du WebService. Il faut ensuite entrer les valeurs de « UserName » et « Password ».

```
//Création d'une instance de la classe CalcService
localhost.CalcService myWebService = new localhost.CalcService();

//Création de l'en-tête
localhost.AuthHeader myHeader = new localhost.AuthHeader();
myHeader.Username = "UserTest";
myHeader.Password = "UserPassword";

//On remplit la propriété AuthHeaderValue
//par le header(en-tête) que nous venons de créer
myWebService.AuthHeaderValue = myHeader;

//Appel des méthodes du WebService
...
```

Le code ci-dessus va permettre de créer un en-tête en attribuant la valeur « UserTest » à « Username » et « UserPassword » à « Password ».

11.3.4. Méthode « Ajouter » :

Voici à présent le code modifié de la WebMethod « Ajouter » du WebService « CalcService », WebService qui a été réalisé dans les chapitres précédents :

```
public class CalcService : System.Web.Services.WebService
{
     public AuthHeader sHeader;

     [WebMethod]
     [SoapHeader("sHeader")]
     public int Ajouter(int valeur1,int valeur2)
     {
          if(sHeader.Username == "UserTest" && sHeader.Password ==
"UserPassword")
```

```
            {
                  return valeur1 + valeur2;
            }
            else
            {
                  return 0;
            }
          }
}
```

Dans cet exemple, la méthode « Ajouter » va retourner la somme des deux arguments si le champ de l'en-tête « UserName » est égale à « UserTest » et si le champ « Password » est égale à « UserPassword ». Dans le cas contraire, la méthode « Ajouter » retourne la valeur « 0 ».
Dans ce cas, le nom de l'utilisateur et son mot de passe est stocké en dur dans le code mais dans un cas réel, il y aurait une méthode qui permettrait de vérifier les informations de l'en-tête à partir d'une base de données.

Attention ! Lorsque l'on crée une authentification personnalisée avec des en-têtes, les protocoles d'accès HttpGet et HttpPost ne sont plus disponibles et seul le protocole SOAP permettra d'accéder au WebService.

11.3.5. HTTPS :

Dans le cas d'une authentification personnalisée, le mot de passe de l'utilisateur circulera en clair dans les en-têtes des messages SOAP.
Il faut donc sécuriser la communication entre le WebService et l'application cliente. Pour cela, il vous faut utiliser le protocole HTTPS (« S » pour « Secure»).

12. UDDI :

Une fois le WebService créé, il faut le rendre disponible à la communauté de développeurs pour qu'ils puissent utiliser le WebService. UDDI est en fait un annuaire de WebServices et va donc permettre aux développeurs de trouver le WebService dont ils ont besoin.

12.1. Présentation :

UDDI veut dire : Universal Description, Discovery and Integration. UDDI est donc une spécification qui va permettre de publier, découvrir et accéder à des informations sur un WebService.

UDDI est donc un annuaire distribué de WebServices. Le protocole de transport d'information pour UDDI est le même que pour les WebServices. Vous l'avez deviné, c'est le protocole SOAP. Il existe trois requêtes pour communiquer avec un annuaire UDDI :

- « Publish » : Permet d'enregistrer un WebService.
- « Find » : Permet de trouver un WebService.
- « Bind ».

La structure des ces trois requêtes se trouve sur le site officiel de UDDI :
http://uddi.org

12.2. Discovery :

Lorsqu'on crée un WebService avec Visual Studio .NET, un fichier *.disco est automatiquement créé. Ce fichier, qui est sous format XML, contient des liens vers le contrat WSDL du WebService et aussi des liens vers d'autres fichiers disco.
Ce fichier va donc permettre de « découvrir » le WebService, c'est-à-dire avoir des informations sur celui-ci.

Si vous n'utilisez pas Visual Studio .NET, mais le Framework SDK, vous pouvez utiliser l'utilitaire « disco.exe » pour générer votre fichier disco.

12.3. Accéder à l'annuaire :

Vous pouvez accéder à l'annuaire UDDI de Microsoft directement à partir de Visual Studio .NET comme cela vous est montré dans la figure de la page suivante. Il vous suffit d'ajouter une référence Web à votre projet et vous pourrez accéder à l'annuaire UDDI. A partir de cet annuaire, vous pourrez chercher un WebService et l'ajouter à votre projet.