Description de la ressource

Propriétés	Description
Intitulé long	Développer un web service avec WCF et un service REST avec OData
Formation concernée	BTS SIO
Matière	SLAM4 et PPE
Présentation	Cette ressource présente la technologie WCF de Microsoft. En prenant appui sur le contexte GSB ; le support propose deux implémentations d'un web service, WCF d'une part et OData d'autre part
Notions	Le Framework EF de persistance, web service, REST, le protocole OData
Outils	Visual Studio2010
Mots-clés	Laboratoire GSB, BTS SIO, WCF, OData, Entity Framework
Auteur(es)	Patrice Grand, avec la relecture attentive d'Annie Baraban
Version	v 1.0
Date de publication	Septembre 2012
Fichiers associés	Les scripts de création de la table (dossier scriptsBD) ainsi que les deux versions de l'application (la version 2 correspondant à l'option alternative décrite en annexe)

SOMMAIRE

Description de la ressource 1
SOMMAIRE....1
Présentation....2
1.Création de la couche de persistance 2
2.Création d'un service 8
3.Un service REST 17
Annexe : gestion de plusieurs contextes 26
4.Déploiement de l'application REST 28

Présentation

Ce document présente, à partir du contexte GSB, les technologies Microsoft pour la création et le déploiement d'un service Web.
Les aspects abordés sont :

- Création d'une couche de persistance avec Entity Framework (EF) avec accès à une base MySQL
- Création d'un service dans deux versions, service Web utilisant WCF et service REST en utilisant Data Services
- Déploiement du service sur un serveur IIS (version 7 ici)

L'ensemble a été développé sous VS2010.

1. Création de la couche de persistance

Nous avons choisi d'utiliser une base de données –*bdMedecins*- sous MySql. Cette base ne contient qu'une seule table, la table Médecin :

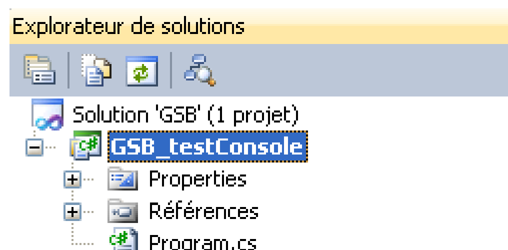
	Champ	Type	Interclassement	Attributs	Null	Défaut	Extra
☐	id	int(11)			Non	*Aucun*	auto_increment
☐	**nom**	varchar(30)	latin1_swedish_ci		Non	*Aucun*	
☐	**prenom**	varchar(30)	latin1_swedish_ci		Non	*Aucun*	
☐	**adresse**	varchar(80)	latin1_swedish_ci		Non	*Aucun*	
☐	**tel**	varchar(15)	latin1_swedish_ci		Oui	*NULL*	
☐	**specialiteComplementaire**	varchar(50)	latin1_swedish_ci		Oui	*NULL*	
☐	**departement**	int(11)			Non	*Aucun*	

Par défaut VS ne gère pas les connexions à MySql. Il faut installer un connecteur :
http://dev.mysql.com/downloads/connector/net/

Attention, il faut la version 6.5.4 au moins !

Ce connecteur peut être utilisé avec ADO ou EF.

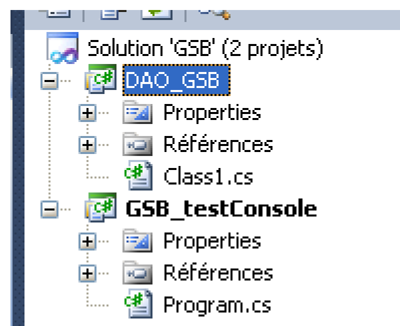
La base de données installée et ses lignes exécutées (fichiers ***createtable.sql*** et ***lignes.sql***), nous allons lancer VS2010 et créer la solution de type console. Cette solution contiendra plusieurs projets correspondant aux couches applicatives. Notre solution s'appelle « GSB ». La solution permet de regrouper et tester les différents projets, mais en fin de compte le déploiement concernera des DLL et des fichiers de configurations.
Nous devons obtenir une structure ressemblant à ceci :



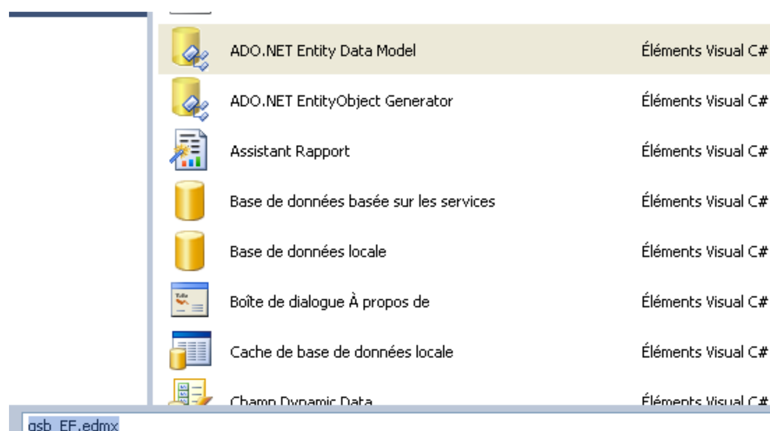
Ce premier projet *GSB_testConsole* (de type console) sera utilisé pour réaliser quelques tests.

1.a Le projet de gestion des données

Ce projet a pour fonction de communiquer avec MySQL et fournir la couche de persistance avec EF. Ajoutons un projet DAO_GSB de type *VisualC#/**Bibliothèque de classes*** à la solution, on obtient :



Ajoutons un *nouvel élément*de type ADO.*NET Entity Model* au projet DAO_GSB, nommé *gsb_EF* :

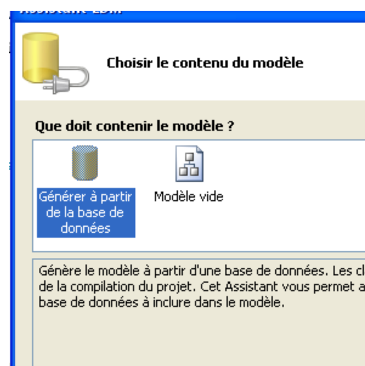


Deux supports du Certa abordent Entity Framework :

- « Utilisation du databinding, d'Entity Framework et de Linq to objects pour développer une application orientée gestion en C# »[1] par Pierre Loisel ;
- « Découverte d'Entity Framework »[2] par Patrice Grand.

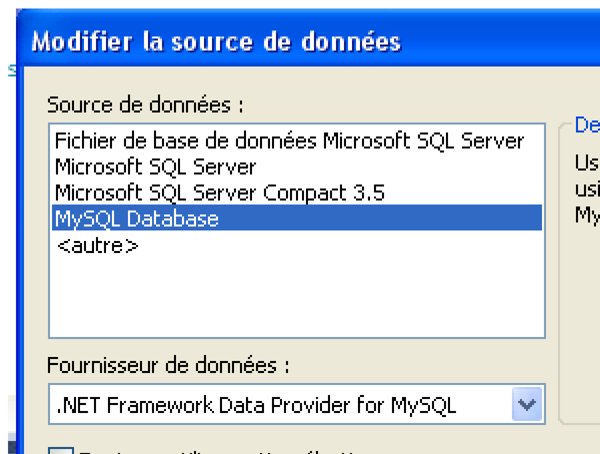
Nous n'y reviendrons pas ici, ce n'est pas l'essentiel.

L'assistant vous propose ensuite de générer le modèle à partir d'une base de données :

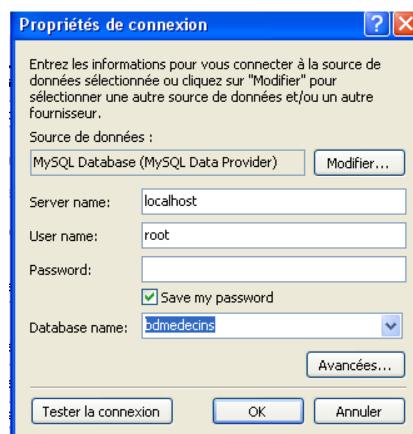


1 http://www.reseaucerta.org/cotelabo/cotelabo.php?num=516
2 http://www.reseaucerta.org/cotecours/cotecours.php?num=508

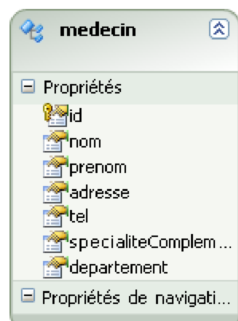
La création de la nouvelle connexion doit faire apparaître le connecteur MySQL pour la nouvelle source de données :



Les propriétés de cette connexion sont celles de votre configuration MySQL :



Tester la connexion. Si tout fonctionne après avoir récupéré les tables, VS2010 vous montre la classe de *mapping* générée :



Pour rappel, une exception de type System.Data.ProviderIncompatibleException est levée si le connecteur MySql utilisé est de version inférieure à la 6.5.4.

Quelques remarques :

- On peut supprimer la classe *class1* générée automatiquement dans le projet.
- On appelle la *classe de contexte*, ou simplement le *contexte*, la classe qui a la responsabilité de gérer les liens entre la base de données et les classes mappées, ici :

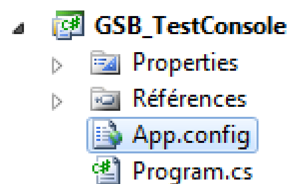
```
publicpartialclassbdMedecinsEntities :ObjectContext
```

- *bdMedecinsEntities* est ce *contexte*.

- La classe *bdMedecinEntities* est annoncée *partial*, comme les classes de mapping ; ce qui signifie que nous pouvons étendre les fonctionnalités de ces classes avec une clause *partial*. Nous le ferons plus loin.
- Une forte dépendance existe entre la classe de contexte et les classes métiers (ici la classe medecin doit demander à son contexte) ; il est possible de réduire fortement cette dépendance en mettant en œuvre une architecture particulière POCO (Plain Old CLR Object) disponible dans VS2010 avec un template de projet POCO.

1.b Test du contexte de données

Nous pouvons tester le contexte de données dans le projet de test GSB_TestConsole, après avoir ajouté une référence au projet DAO_GSB :
Il faut par ailleurs ajouter un fichier de configuration (*ajouter/ nouvel élément/fichier de configuration de l'application*) qui va contenir la chaîne de connexion, copiée à partir du fichier de configuration du projet DAO_GSB.




```
…
usingSystem.Data.EntityClient; // et ajouter la référence System.Data.Entity

classProgram
{
staticvoid Main(string[] args)
{
            bdMedecinsEntitiescontexte = newbdMedecinsEntities();
        varreq = from m incontexte.medecin
        where m.id == 45
        selectm.nom;
            var liste = req.ToList();
            Console.WriteLine(liste.First().ToString());
}

}
```

Quelques remarques :

- L'utilisation de ***var*** (depuis leframework 3.0) permet de s'affranchir de la déclaration du type, l'objet créé restant néanmoins fortement typé.

- Nous avons mis en œuvre notre première requête en langage *Linq*, la syntaxe s'approche de SQL -voir à ce propos la ressource de Pierre Loisel, très complète sur le sujet- et nous bénéficions de l'intelliSense.

1.c Gestion du contexte

Notre couche de mapping est basée sur la création d'un contexte. Il existe plusieurs stratégies concernant la gestion du contexte. On peut envisager de ne créer qu'un seul contexte (une seule instance) pour tous les clients ou bien un contexte par client.
Nous présentons une version simple où un unique contexte sera utilisé. Nous présentons en annexe une version avec plusieurs contextes.
Pour cela nous allons ajouter une classe *Contexte* au projet DAO_GSB qui assurera l'unicité du contexte utilisé ; cette classe met en œuvre un *singleton* (objet singleton *static* + constructeur privé + méthode publique *static* d'accès au singleton) :

```
publicclassContexte
{
privatestaticbdMedecinsEntitiesleContexte = null;

privateContexte(){}

publicstaticbdMedecinsEntitiesgetContexte()
{
if(Contexte.leContexte == null)
Contexte.leContexte = newbdMedecinsEntities();
returnContexte.leContexte;
}
}
```

1.d Evolution des fonctionnalités du contexte de données

Notre couche de *mapping* est générée automatiquement par VS2010 (la classe de contexte et le classe mappée –*medecin*-).
Sur certaines des fonctionnalités proposées par défaut, on peut imaginer des services enrichis justifiés par des contraintes « métiers ». Nous allons ainsi enrichir la classe *medecin* de quelques méthodes que nous utiliserons dans notre service WCF.
Il y a deux techniques pour *étendre* –enrichir- une classe : soit lorsque c'est possible, utiliser le contrat *partial* de la classe à enrichir, soit, depuis le Framework 3.5 utiliser *les méthodes d'extension*. Nous utiliserons ces deux techniques.

1.d.1 Une classe *partial*

VS 2010 a défini la classe *medecin**partial*** ; utilisons cette opportunité pour créer notre propre classe *medecin*, déclarée également *partial*, nécessairement dans le même *namespace.*
Nous allons simplement ajouter quelques méthodes utiles pour notre futur ServiceWeb :

```
publicpartialclassmedecin
{
publicstaticList<int>lesDepartements()
{
bdMedecinsEntitiescontexte =  Contexte.getContexte();
varreq = from m incontexte.medecin
selectm.departement;
varliste = req.Distinct().ToList();
       liste.Sort();
returnliste;
}
publicstaticList<medecin>lesMedecins(intnumDepartement)
{
bdMedecinsEntitiescontexte = Contexte.getContexte();
varreq = from m incontexte.medecin
wherem.departement==numDepartement
select m;
returnreq.ToList();
}
publicstaticList<medecin>lesMedecins(string nom)
{
bdMedecinsEntitiescontexte = Contexte.getContexte();
varreq = from m incontexte.medecin
wherem.nom.ToUpper() == nom.ToUpper()
select m;
returnreq.ToList();
}
publicstaticmedecinCreatemedecin(string nom, stringprenom, stringadresse, stringtel,
stringville, stringspecialite, intdepartement)
```

```
{
medecin m = newmedecin();
m.nom = nom; // pas d'dentifiant puisque les id sont auto-incrémentés dans MySql
m.prenom = prenom;
m.tel = tel;
m.specialiteComplementaire = specialite;
m.departement = departement;
m.adresse = adresse;
return m;
}
}
```

La méthode *lesDepartements* retourne la liste (sans doublon) des départements où sont implantés des médecins. Les méthodes surchargées *lesMedecins* filtrent les médecins suivant le département ou le nom. La dernière méthode permet de créer un nouveau médecin en surchargeant la « fabrique » *static* générée par le framework ; nous créons ce nouveau médecin sans passer le numéro.

Le test se fait plutôt en utilisant le débogage après appel dans le Main :

```
List<int> listeDepartements = medecin.lesDepartements();
List<medecin> listeMedecinDep = medecin.lesMedecins(5);
List<medecin> listeMedecinNom = medecin.lesMedecins("Cherioux");
```

100 %

Espion 1

Nom	Valeur
lesContextes	Count = 1
[0]	{[session1, DAO_GSB.bdMedecinsEntities]}
Key	"session1"
Value	{DAO_GSB.bdMedecinsEntities}
base	{DAO_GSB.bdMedecinsEntities}
_medecin	{System.Data.Objects.ObjectSet<DAO_GSB.medecin>}
medecin	{System.Data.Objects.ObjectSet<DAO_GSB.medecin>}
base	{System.Data.Objects.ObjectSet<DAO_GSB.medecin>}
EntitySet	{medecin}
Membres non publics	
Affichage des résultats	L'agrandissement de l'affichage des résultats permet d'énumérer IEnumerable
[0]	{DAO_GSB.medecin}
[1]	{DAO_GSB.medecin}

1.d.2 Les méthodes d'extension

Depuis le framework 3.5, il est possible d'enrichir les services d'une ***classe existante*** (du framework ou métier). Ainsi, on peut ajouter des méthodes à la classe *String* ou *int* en respectant un mécanisme et une syntaxe appropriée. Les méthodes ainsi « ajoutées » sont les méthodes d'extension. La « durée de vie » de ces méthodes étant liée –bien sûr- à la déclaration de ces extensions (une classe n'est *étendue* que le temps d'utilisation du projet dans lequel cette extension a été définie).
Nous allons illustrer cette technique en ajoutant une méthode d'ajout et de sauvegarde pour la classe du framework*EntityObject* dont héritent toutes les classes de mapping.
Ajoutons une nouvelle classe au projet DAO_GSB :

```
publicstaticclassExtendEntityObject
{
publicstaticvoidajouter(thisEntityObject entity)
{
bdMedecinsEntities contexte = Contexte.getContexte();
contexte.AddObject(entity.GetType().Name, entity);
contexte.SaveChanges();
}
}
```

Remarques générales sur les méthodes d'extension :

- La classe qui fournit une méthode d'extension doit être *static*
- La méthode doit être aussi déclarée *static*
- La méthode d'extension doit avoir comme premier paramètre un paramètre du type de la classe à « étendre » précédé du mot ***this***
- Une même classe peut proposer plusieurs méthodes d'extension d'une même classe ou de plusieurs classes
- L'utilisateur de cette méthode doit –bien sûr- inclure cette classe dans son projet.

Notre méthode d'extension ne fait qu'ajouter et sauvegarder un objet dans le contexte.

Remarques sur le code :

- La première instruction récupère le contexte
- La seconde ajoute l'objet au contexte. Il y a (au moins) deux méthodes pour ajouter un objet au contexte. Soit nous appelons la méthode AddObject à partir de la propriété *medecin* du contexte : `contexte.medecin.AddObject(entity),` soit nous procédons comme montré dans le code en demandant au contexte *d'inférer* sur le type. L'avantage de cette deuxième solution est qu'elle est générique et utilisable pour toute classe (héritant d'*EntityObject*)
- La dernière instruction sauvegarde en base.

Notre partie d'accès aux données est terminée.
Dans le Main du projet de test, nous pouvons écrire :

```
medecin m = medecin.Createmedecin("Janus", "Gilles", "23, rue Petit",
              "0565987898", "Paris 75000", "sports", 75);
m.ajouter();
```

2. Création d'un service

Un service est un programme qui permet de faire communiquer deux applications. Lorsque ce service utilise le protocole http, on parle de service web.
Un service a besoin d'un *fournisseur* de services(application qui héberge ce service) et d'applications clientes qui vont *consommer* ce service.
L'application « Vélib »[3]
présentait la consommation d'un service (REST).

Dans notre cas, nous allons passer de l'autre côté, côté fournisseur.

Nous allons utiliser WCF (Windows Communication Foundation) qui est la technologie qui remplace les échanges d'objet COM, DCOM.

WCF a comme objectif de simplifier la configuration des échanges. De nombreuses ressources sur internet présentent cette technologie ; rappelons seulement l'essentiel qui est basé sur la définition de 3 éléments (A-B-C).

A pour Adresse : à partir de quelle *uri* peut-on accéder à la ou les ressources ?
B pour Binding : quelle communication entre les applications ? (http, tcp ou autre)
C pour contrat : quel sont les noms des services (méthodes) ?

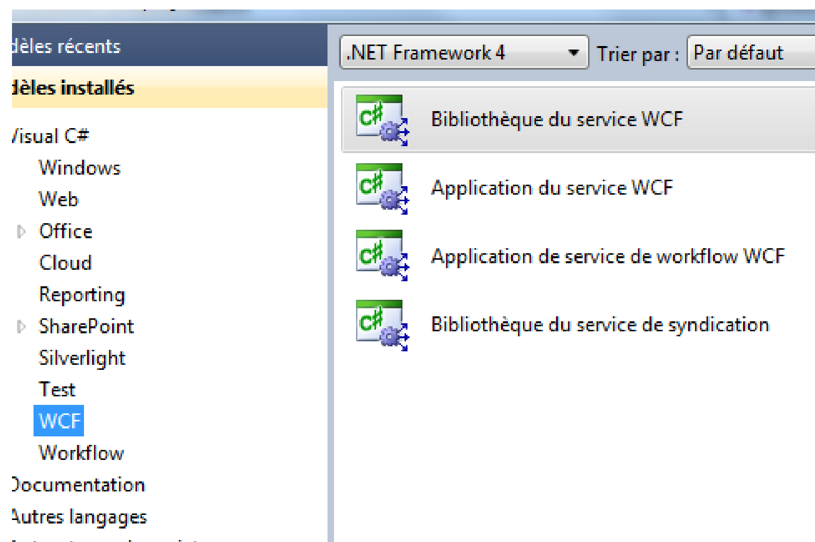
Les 2 premiers éléments doivent être décrits dans un fichier de configuration XML.

Nous présenterons deux types de service, un premier de type WCF (service WEB) et un second de type REST qui utilise une couche ODATA dédiée au service REST.

3 http://www.reseaucerta.org/exonets/exonet.php?num=515

2.a Le projet WCF

Nous allons ajouter à notre solution un nouveau projet :



Ce projet va générer une dll qui sera utilisée par l'application qui hébergera notre service. C'est dans ce projet que nous allons définir le point C (contrat) évoqué, nommons ce service ***WcfService_GSB***.

VS2010 propose un *patron* pour l'application que nous allons en partie utiliser, le projet généré présente une classe interface et son implémentation :

La définition du contrat –point C- d'un service passe obligatoirement par une classe interface.
Si on ouvre l'interface (*IService1.cs*) on trouve du code par défaut :

L'interface est décorée de l'attribut [*ServiceContract*], cet attribut est obligatoire pour définir un service. Chaque méthode est décorée par l'attribut [*OperationContract*] obligatoire également.
Nous allons faire un peu le ménage dans le projet proposé.

- Supprimons le fichier *App.config*
- Ajoutons une référence vers le projet DAO_GSB
- Renommons l'interface IService1 qui devient *IServiceMedecin*
- Supprimons la classe *CompositeType*, qui était décorée *DataContract* car nous pourrons directement utiliser la classe *medecin* de notre projet d'accès aux données (VS2010 pose un attribut DataContract sur cette classe).

Nous obtenons un code très léger pour l'interface :

```
namespaceWcfService_GSB
{
[ServiceContract]
publicinterfaceIServiceMedecin
{
[OperationContract]
stringGetData(int value);
}
}
```

Enfin, supprimons la classe d'implémentation (Service1) pour nous concentrer sur le seul contrat.
Dans cette interface nous devons présenter les signatures des méthodes de notre service, bref que voulons-nous exposer ?
Ajoutons au projet une référence à *System.Data.Entity*.
Proposons les méthodes de notre service :

```
[ServiceContract]
publicinterfaceIServiceMedecin
{
[OperationContract]
List<medecin>getMedecinsParDep(intnumDepartement);
[OperationContract]
List<medecin>getMedecinsParNom(string nom);
[OperationContract]
List<int>lesDepartements();
[OperationContract (IsOneWay=true) ]
voidajouterMedecin(string nom, stringprenom, stringadresse,
stringville, stringspecialite, string telephone, intnumDepartement);
}
```

Remarque : *IsOneWay* indique à l'attribut que rien n'est retourné, il s'agit d'une « procédure ».
Passons maintenant à l'implémentation du contrat. Pour cela ajoutons une classe au projet : *ServiceMedecin*.

```
namespace serviceGSB
{
    class ServiceMedecin :IServiceMedecin
    {
    }
}
```

Si on fait un clic droit sur l'interface, VS nous demande si on veut implémenter l'interface ; répondons bien sûr oui. Toutes les signatures des méthodes de l'interface apparaissent.

Il ne nous reste plus qu'à écrire le code.

```
publicclassServiceMedecin : IServiceMedecin
{
#regionIServiceMedecinMembres

publicList<DAO_GSB.medecin>getMedecinsParDep(intnumDepartement)
{
returnmedecin.lesMedecins(numDepartement);
}
publicList<DAO_GSB.medecin>getMedecinsParNom(string nom)
{
returnmedecin.lesMedecins(nom);
}
publicList<int>lesDepartements()
{
returnmedecin.lesDepartements();
}
publicvoidajouterMedecin(string nom, stringprenom, stringadresse, stringville,
stringspecialite,
string telephone, intnumDepartement)
{
medecin m = medecin.Createmedecin(nom, prenom, adresse, telephone, ville, specialite,
numDepartement);
m.ajouter();
}
#endregion
}
```

Remarques

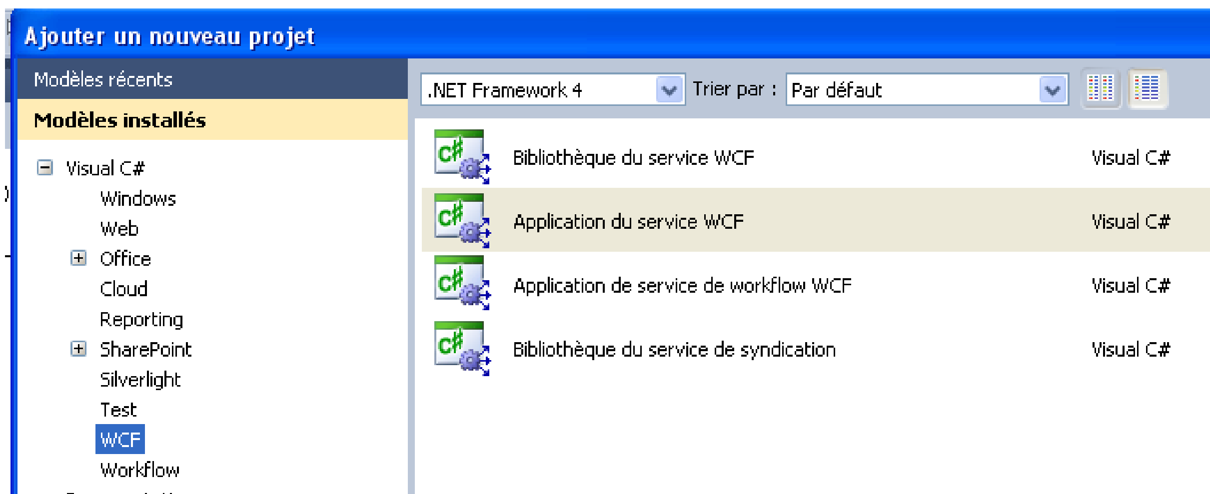
- Nous avons conservé le type *DAO_GSB.medecin*, généré par l'environnement ; il est, bien sûr, possible de ne mettre que *medecin* (à condition d'avoir bien ajouté la clause *using* correspondante).
- Dans la méthode *ajouterMedecin*, nous utilisons la méthode *CreateMedecin* de la classe *partial* et la méthode d'extension *ajouter*.

2.b Hébergement du service

Un service WCF peut être hébergé par différents type de projets –côté serveur- : nous avons généré jusqu'ici deux dll (une pour la gestion des données et l'autre pour la gestion du service). Aussi, tout type d'application (Winform, ASP, console) peut lancer les dll, il suffira de configurer le serveur web (IIS) pour lui indiquer quelle application il doit lancer (et pas seulement ASP). Nous avons choisi d'héberger le service dans une application web –ASP- car le cahier des charges demande une authentification pour accéder au service.

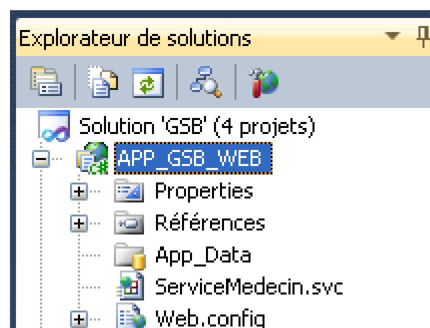
Remarque : en fait la *seule* authentification dans l'en-tête http ne nécessitait pas une application web.

Ajoutons un nouveau projet à notre solution :



Le projet se nommera *APP_GSB_WEB*.
Si nous observons les fichiers générés, nous retrouvons une organisation proche du type de projet précédent (Bibliothèque du service WCF). Nous pouvions aussi n'utiliser qu'un seul projet exposant le service (et donc son interface) et son hébergement. Nous avons choisi de séparer les fonctionnalités dans deux projets distincts : la création du service d'une part (projet WcfService_GSB générant une dll) et l'application d'hébergement de type ASP.

Supprimer les fichiers, de manière à obtenir l'organisation suivante :



Ce projet, dans un premier temps, ne comprend que peu de fichiers, le fichier *ServiceMedecin.svc* (pour lequel nous avons supprimé le fichier de *code behind*, *ServiceMedecin.svc.cs*) et le fichier de configuration *Web.config*.
Le code du fichier ServiceMedecin.svc ne doit contenir ici que le nom du service :

```
<%@ServiceHost Language="C#" Debug="true" Service="WcfService_GSB.ServiceMedecin"%>
```

Le nom du service est de type :

<nom du namespace>.<nom de la classe d'implémentation du service>

Remarque : ce fichier n'est plus nécessaire depuis la version 4.0 du framework ; on peut en effet indiquer le nom du service dans le fichier de configuration.
Ajoutons les références aux deux projets *DAO_GSB* et *WCF_Service_GSB* ainsi qu'à la dll *System.Data.Entity*.

La configuration du service (les points A et B) se fait dans le fichier *Web.config* :

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
  <system.serviceModel>
    <services>
      <service name="WcfService_GSB.ServiceMedecin">
        <endpoint address="" binding= "basicHttpBinding" contract="WcfService_GSB.IServiceMedecin">
          <identity>
            <dns value="localhost" />
          </identity>
        </endpoint>
        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
      </service>
    </services>
    <behaviors>...</behaviors>
    <serviceHostingEnvironment multipleSiteBindingsEnabled="true" />
  </system.serviceModel>
  <system.webServer>...</system.webServer>
  <connectionStrings>
    <add name="bdMedecinsEntities" connectionString="metadata=res://*/gsb_EF.csdl|res://*/gsb_EF.ssdl|res://*/gsb_EF.msl
  </connectionStrings>
</configuration>
```

Le fichier doit contenir la chaîne de connexion (copiée à partir du projet DAO_GSB).
La balise <service> décrit les paramètres du service. Deux *endPoint* – points de connexion- sont indiqués.

2.b.1 Premier point de connexion

Le premier a comme adresse relative (Point A du contrat) la racine et comme liaison (point B du contrat) *basicHttpBinding* . Il existe 4 types de *binding* selon le protocole et la sécurité choisis(voir la documentation MSDN à ce propos) ; celui mis en œuvre ici supporte le protocole SOAP indispensable à un webService.
C'est ce premier point qui exposera les méthodes du service.
La balise <identity> n'est pas nécessaire.

2.b.2 Deuxième point de connexion

Le second point est obligatoire pour un service SOAP : il s'agit ici de présenter au client *la structure* du service afin que celui-ci puisse le consommer.
Pour voir de quoi il s'agit, lançons en mode debug (F5) la solution (après avoir indiqué que le projet ASP est le projet de démarrage). L'application lance le navigateur :

```
  Tuesday, July 17, 2012 09:25 AM        <dir> App_Data
Wednesday, July 18, 2012 11:44 AM        4,663 APP_GSB_WEB.csproj
Wednesday, July 18, 2012 11:44 AM        1,086 APP_GSB_WEB.csproj.user
Wednesday, July 18, 2012 11:44 AM        <dir> bin
  Tuesday, July 17, 2012 09:25 AM        <dir> obj
  Tuesday, July 17, 2012 09:25 AM        <dir> Properties
Wednesday, July 18, 2012 11:10 AM           88 ServiceMedecin.svc
Wednesday, July 18, 2012 11:44 AM        1,863 Web.config
  Tuesday, July 17, 2012 09:25 AM          287 Web.Debug.config
  Tuesday, July 17, 2012 09:25 AM          383 Web.Release.config
```

Si nous ouvrons le lien *ServiceMedecin.svc* nous obtenons la présentation du service.

Service ServiceMedecin

Vous avez créé un service.

Pour tester ce service, vous allez devoir créer un client et l'utiliser pour appeler le service. Pour ce faire, vous

```
svcutil.exe http://localhost:1784/ServiceMedecin.svc?wsdl
```

Cette opération va créer un fichier de configuration et un fichier de code contenant la classe du client. Ajoutez exemple :

C#

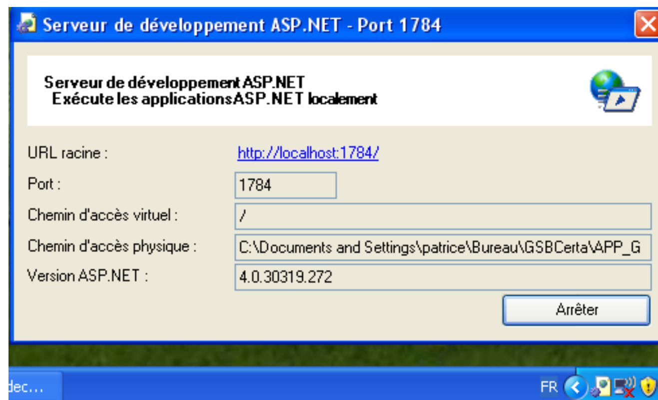
```
class Test
{
    static void Main()
    {
        ServiceMedecinClient client = new ServiceMedecinClient();

        // Utilisez la variable 'client' pour appeler des opérations sur le
```

Si nous cliquons sur le lien proposé, nous obtenons la description (au format XML) des méthodes du service :

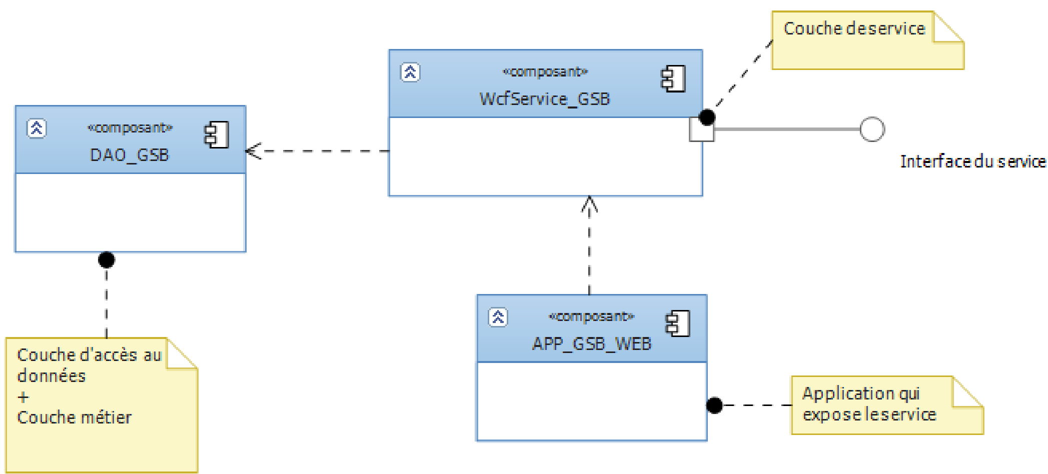
```
  <?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions name="ServiceMedecin" targetNamespace="http://tempuri.org/" xmlns:wsdl=
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:soapenc="http://schemas.
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:xsd="http:/
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:tns="http://tempuri.o
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsap="http://scher
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:msc="http://schema
    xmlns:wsa10="http://www.w3.org/2005/08/addressing" xmlns:wsx="http://schemas.xm
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
  - <wsdl:types>
    + <xsd:schema targetNamespace="http://tempuri.org/Imports">
    </wsdl:types>
  - <wsdl:message name="IServiceMedecin_getMedecinsParDep_InputMessage">
      <wsdl:part name="parameters" element="tns:getMedecinsParDep" />
    </wsdl:message>
  - <wsdl:message name="IServiceMedecin_getMedecinsParDep_OutputMessage">
      <wsdl:part name="parameters" element="tns:getMedecinsParDepResponse" />
    </wsdl:message>
  - <wsdl:message name="IServiceMedecin_getMedecinsParNom_InputMessage">
      <wsdl:part name="parameters" element="tns:getMedecinsParNom" />
    </wsdl:message>
```

C'est ce fichier qui permettra au client de consommer le service en donnant les informations nécessaires à la création d'une classe *proxy* (mandataire du service), le sur-langage utilisé est WSDL.

Remarque : on peut aussi lancer l'application directement par le serveur de développement installé dans VS2010, à partir de l'icône en bas à droite du bureau :



Pour résumer l'architecture applicative, voici le diagramme de composants :

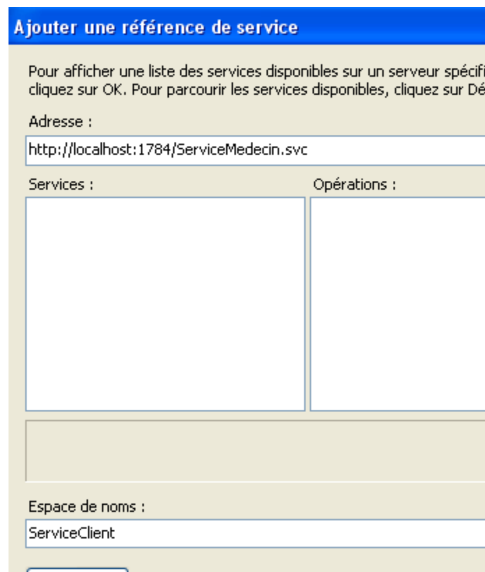


Remarque : la couche métier dans notre cas est réduit à l'enrichissement de la classe medecin (classe partial) et aux méthodes d'extension.

2.c Un client de test du service

Nous allons ajouter un nouveau projet afin de tester le service. Faisons simple, contentons-nous d'une application console :

Ajoutons à ce projet une référence de service (clic droit sur Références/ajouter une référence de service) ; une fenêtre s'ouvre, complétons les champs :

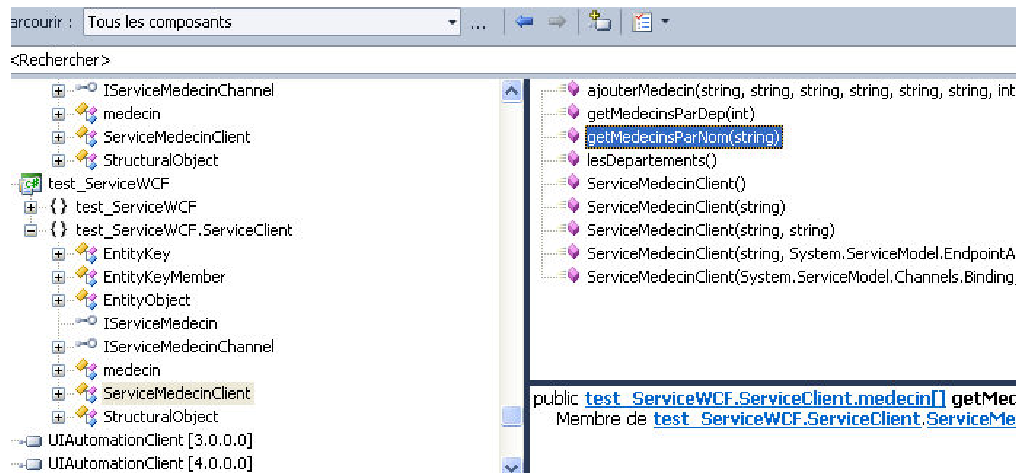


Remarques :

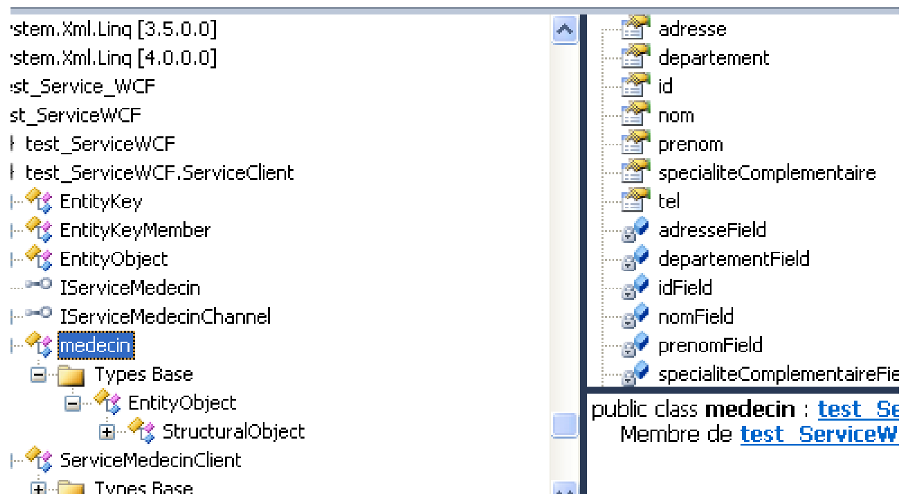
- Le numéro de port est visible à partir du serveur de développement.
- L'espace de nom est le namespace du service côté client.

En cliquant sur OK, on constate que beaucoup de choses ont été générées :

- Si on demande à voir le service dans l'explorateur d'objets (clic droit sur ServiceMedecin/voir dans l'explorateur d'objets), on peut visualiser les services côté client :



- La classe *medecin* est également accessible :



- Les services sont disponibles à partir d'une classe proxy : *ServiceMedecinClient,* générée automatiquement au moment de la création de la référence au service Web.
- Les fonctions qui retournaient des listes dans le service retournent maintenant des tableaux, côté client. C'est bien normal puisque les WebServices doivent être indépendants des technologies et le type List est spécifique à C#, contrairement aux tableaux.

L'autre élément généré est le fichier de configuration *app.config* dans lequel on peut retrouver la configuration (*A, B et C*) du point de connexion :

```
<endpointaddress="http://localhost:1784/ServiceMedecin.svc"
binding="basicHttpBinding"bindingConfiguration="BasicHttpBinding_IServiceMedecin"
contract="ServiceClient.IServiceMedecin"name="BasicHttpBinding_IServiceMedecin" />
```

Testons le service côté client :

```
staticvoid Main(string[] args)
{
ServiceClient.ServiceMedecinClientsrv =
        newServiceClient.ServiceMedecinClient();
        Console.WriteLine(srv.lesDepartements().Count());
List<ServiceClient.medecin>lst =
srv.getMedecinsParNom("brouzais").ToList<ServiceClient.medecin>();
foreach (ServiceClient.medecin m inlst)
Console.WriteLine(m.prenom);
srv.ajouterMedecin("Dupond", "jean", "rue petit", "paris", "sport", "0213123456", 75);
}
```

Les résultats sont conformes.

Par contre, si l'on tente l'appel suivant :

```
        ServiceClient.ServiceMedecinClientsrv =
        newServiceClient.ServiceMedecinClient();
List<ServiceClient.medecin>lst =
        srv.getMedecinsParDep(2).ToList<ServiceClient.medecin>();
foreach (ServiceClient.medecin m inlst)
Console.WriteLine(m.prenom);
```

Nous générons une exception :

```
{
ServiceClient.ServiceMedecinClient srv = new ServiceClient.ServiceMedecinClient();
List<ServiceClient.medecin> lst = srv.getMedecinsParDep(2).ToList<ServiceClient.medecin>();
foreach (ServiceClient.medecin m in lst)
    Console.WriteLine(m.prenom);
```

La taille maximale pour les messages entrants a été dépassée; modifions dans le fichier de configuration les deux attributs *maxBufferSize* et *maxReceivedMessageSize* en augmentant largement leurs capacités (qui doivent rester néanmoins égales).

La configuration du client est très réduite avec VS2010 ; il est néanmoins possible de générer « soi-même » la classe proxy en utilisant un outil du framework*svcutil.exe* sur le fichier wsdl.

3. Un service REST

Le projet GSB doit fournir un service REST, nous allons aborder maintenant cette partie.
Un service REST fournit les mêmes services, mais les échanges entre le serveur et le client sont différents.

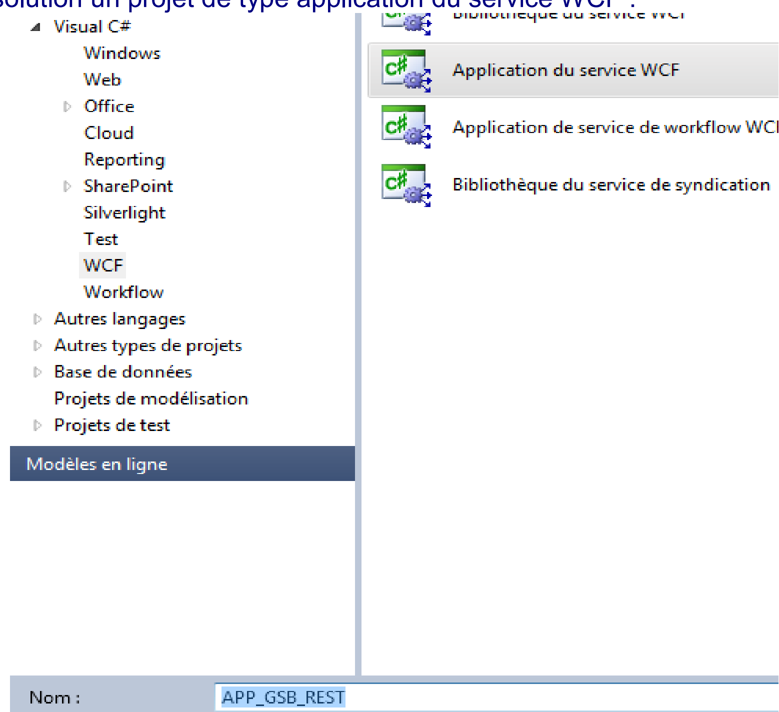
- Le protocole n'est plus SOAP mais simplement http.
- Le service n'est plus consommable par l'intermédiaire de méthodes mais directement par une URI qui fournit la ressource en XML.

Ce type d'échange est très utilisé dans la téléphonie ou pour accéder aux différents services d'Amazon, Google Maps, YouTube, FaceBook ou autre.

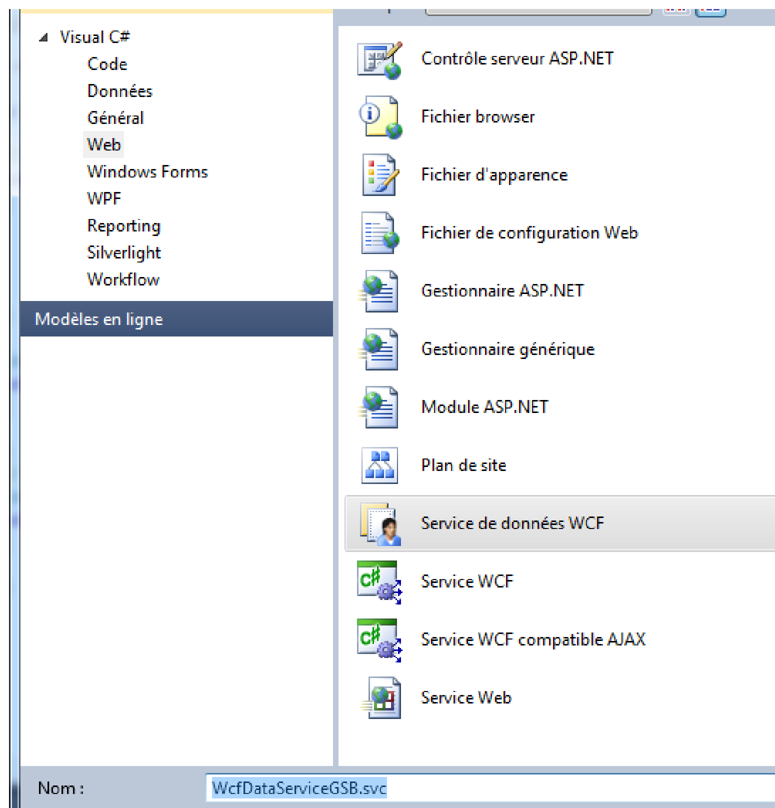
Dans notre projet, notre service REST permettra de consulter les médecins et d'ajouter un nouveau médecin. Nous utiliserons la couche d'accès aux données DAO_GSB.

3.a Création du projet REST

Ajoutons à notre solution un projet de type application du service WCF :



Une fois le projet créé, supprimons tous les fichiers. Ajoutons à ce projet un nouvel élément de type Service de données WCF :



Ce type d'élément est dédié au service REST, il utilise le protocole Microsoft Open Data (OData) qui permet des échanges utilisant les langages Atom ou Json.

Un *service*OData peut bien sûr être consommé par n'importe quel client.
OData permet des échanges CRUD à partir du seul protocole http (spécifique au service REST) et offre une très riche syntaxe de requêtage à partir de l'URL. Un service est ainsi *déclaré*OData s'il autorise le requêtage du protocole. Le site référence du protocole présente toutes les nombreuses possibilités : http://www.odata.org/documentation/uri-conventions
Un service WCF peut être construit pour OData ; dans ce cas, on doit utiliser une couche au-dessus de WCF qui est WCF Data Services. C'est le choix que nous faisons ici.

Le projet doit ressembler à ceci :

Un fichier Web.Config et un service ont été ajoutés.
Ajoutons au projet la référence à notre projet DAO_GSB.
Si nous ouvrons le *code behind* du service, une *classe template* –WcfDataServiceGSB- est proposée :

```
publicclassWcfDataServiceGSB : DataService</* TODO: placez ici le nom de votre classe source de données */>
```

Cette classe demande une classe source de données ; indiquons notre classe construite avec Entity Framework :
…

```
usingSystem.Web;
using DAO_GSB;
```

```
namespace APP_GSB_REST
{
publicclassWcfDataServiceGSB : DataService<bdMedecinsEntities>
{
```

La classe de service est ainsi prête à utiliser notre classe *medecin*.

Ce fichier propose de configurer différents services :

```
// config.SetEntitySetAccessRule("MyEntityset", EntitySetRights.AllRead);
// config.SetServiceOperationAccessRule("MyServiceOperation", ServiceOperationRights.All);
```

La première configuration correspond aux classes de données –Entity- ; indiquons que notre classe *medecin* sera accédée en lecture et écriture :

```
config.SetEntitySetAccessRule("medecin", EntitySetRights.All);
```

Il faudrait ajouter de nouvelles lignes si nous exposions plusieurs classes.

La seconde ligne propose la configuration de méthodes (qu'il faudra définir ensuite) ; ajoutons deux méthodes :

```
config.SetServiceOperationAccessRule("departement", ServiceOperationRights.AllRead);

config.SetServiceOperationAccessRule("lesDepartements",
ServiceOperationRights.AllRead);
```

La première méthode devra s'appeler *departement* et la seconde *lesDepartements*, l'accès sera en lecture.

Les méthodes s'écrivent dans la classe du service :

```
[WebGet]
publicIQueryable<medecin>departement(intnum)
{
returnmedecin.lesMedecins(num).AsQueryable<medecin>();
}

[WebGet]
publicList<int>lesDepartements()
{
returnmedecin.lesDepartements();
}
```

Remarques :

- Chaque méthode accessible par la méthode http « GET » devra utiliser l'attribut *WebGet.*
- Pour pouvoir bénéficier du requêtageOData, une méthode doit retourner un objet de type IQueryable<T> (type de base des requêtes Linq) ; aussi nous castons la liste de *medecin* en IQueryable de *medecin*.

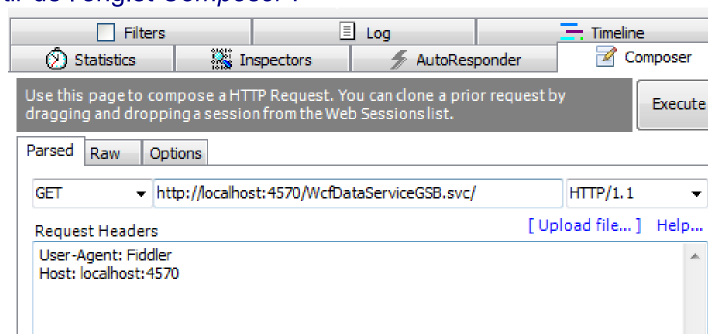
Nous en avons terminé avec la définition du service REST.

3.b Test du service REST, l'outil Fiddler2

Microsoft a développé un outil qui permet de visualiser les échanges http, émis et reçus côté serveur.

Il faut, bien sûr installer le logiciel :
http://www.fiddler2.com/fiddler2/version.asp

Lançons le logiciel. La fenêtre de gauche historise les requêtes, celle de droite permet de lancer sa propre requête à partir de l'onglet *Composer* :



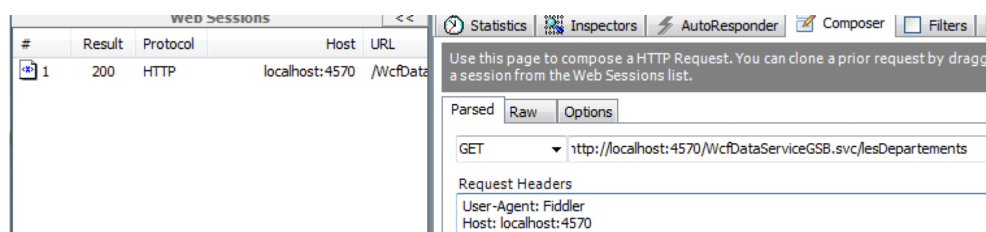
Nous avons lancé une requête http (GET) appelant le service ; le numéro du port du serveur IIS de développement est visible si on passe au-dessus du serveur (en bas à droite). On peut également faire un clic droit sur le service dans l'explorateur de solutions, demander à *afficher dans le navigateur* et copier cette adresse dans Fiddler2. Une autre possibilité est de fixer le port en ouvrant les propriétés du projet, aller sur onglet Web et remplir la valeur du port spécifique.

Si on clique sur le bouton *Execute*, la requête s'exécute côté serveur :

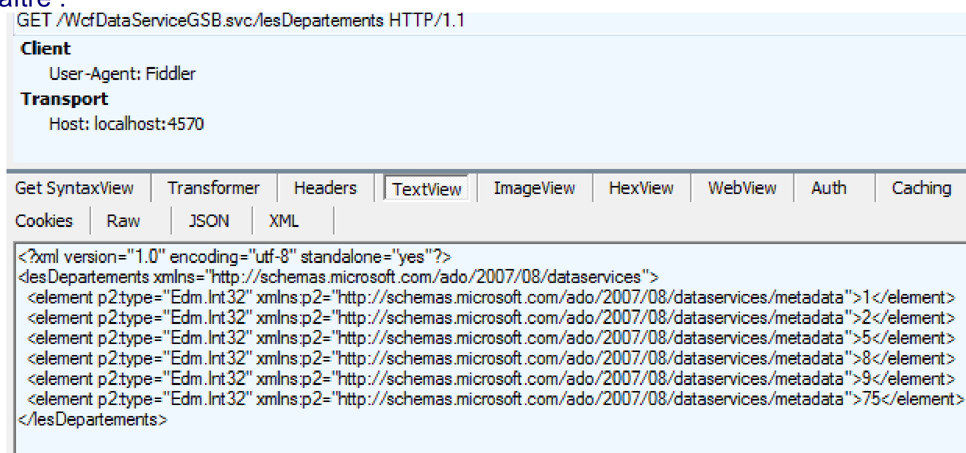
#	Result	Protocol	Host	URL	Body	Caching	Content-Type
1	200	HTTP	localhost:4570	/WcfDataServic...	410	no-cac...	application/xml;charset=utf-8

Le serveur nous retourne une classe 2xx, le type de retour sera du XML.

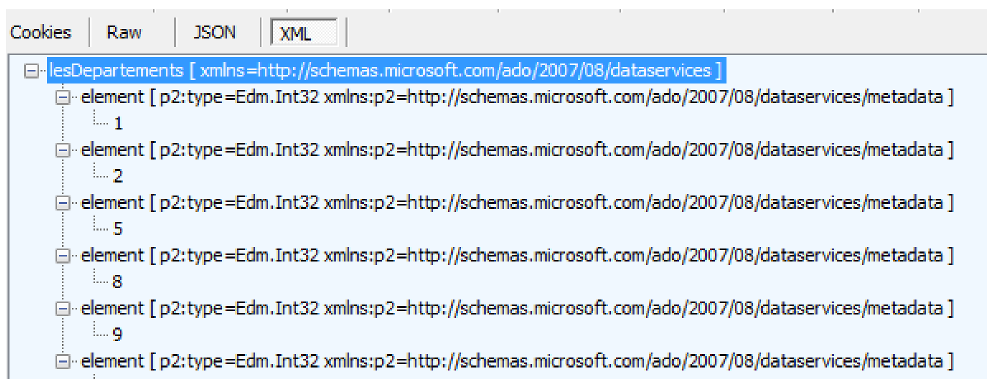
Modifions notre requête afin de demander les départements :



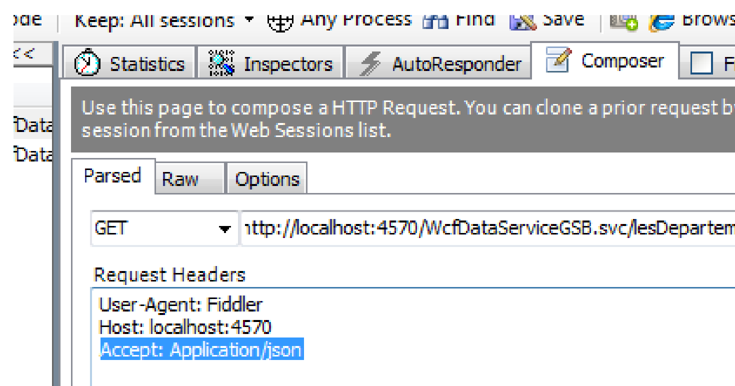
Maintenant regardons ce que retourne le serveur en cliquant sur l'onglet *Inspector*, l'onglet *textView* fait apparaître :



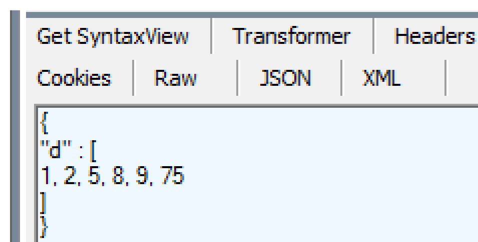
Et l'onglet XML :



Si nous voulons recevoir la réponse au format Json, il faut le demander dans l'en-tête http, et c'est là où l'outil est intéressant car nous pouvons modifier l'en-tête de la requête.
Revenons sur l'onglet Composer et ajoutons une entrée dans l'en-tête :

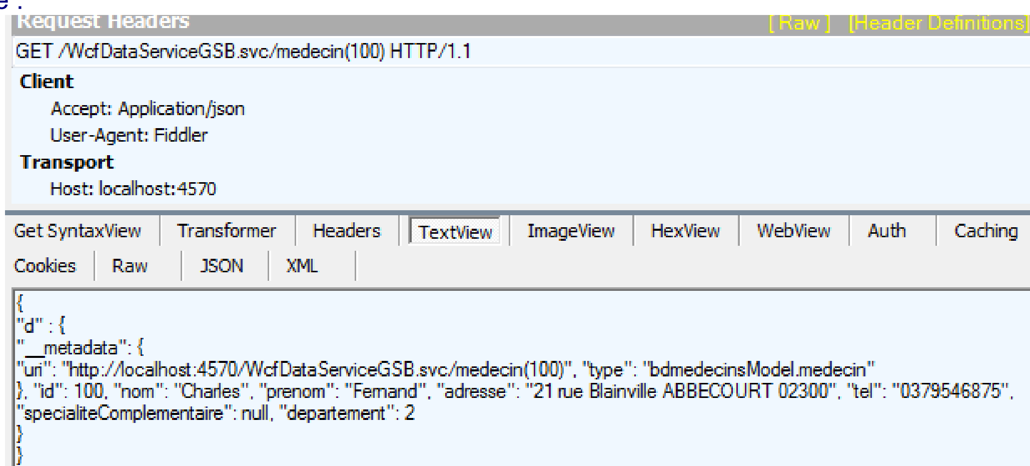


Si nous exécutons et affichons au format textView :



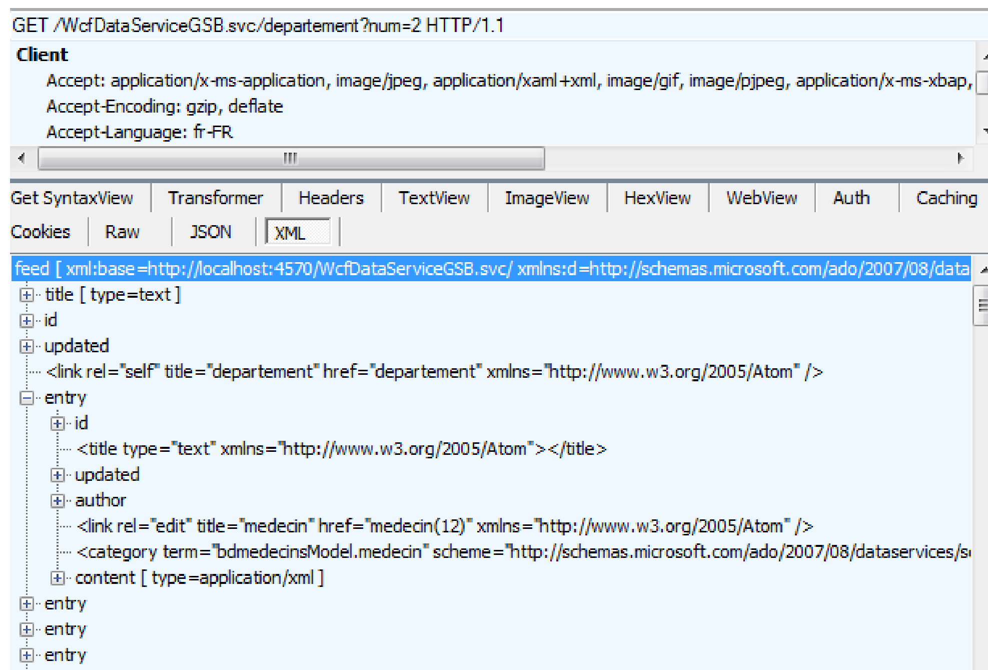
Nous visualisons le format json. Ici le résultat n'est pas totalement significatif car le type de retour n'est pas une donnée structurée mais un tableau d'entiers, nous referons le même exercice pour les médecins plus loin.

Continuons en requêtant les médecins. On peut demander le médecin dont l'identifiant est 100 avec la requête :



La réponse a été demandée au format json.

Continuons en testant la méthode departement :



La réponse –ici au format XML/Atom- retourne bien les médecins du département 2 ; notez num=2 qui correspond au nom du paramètre de la méthode.
On peut envisager de nombreuses requêtes, par exemple :

http://localhost:4570/WcfDataServiceGSB.svc/medecin?$select=nom,prenom
retourne les nom et prénom de tous les médecins

http://localhost:4570/WcfDataServiceGSB.svc/medecin(200)?$select=nom,prenom
retourne le nom et prénom du médecin d'identifiant 200

http://localhost:4570/WcfDataServiceGSB.svc/departement?num=2&$top=20&$orderby=nom

retourne les 20 premiers ($top=20) médecins du département numéro 2, triés sur le nom.

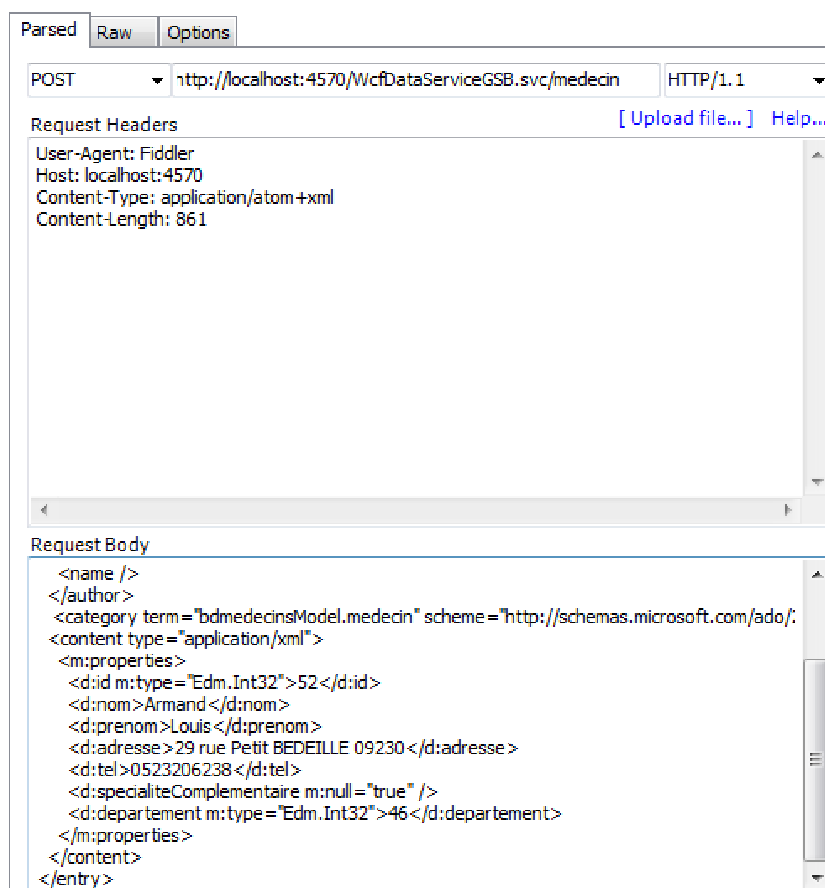
http://localhost:4570/WcfDataServiceGSB.svc/medecin?$filter=departement_eq_2

retourne les médecins du département 2

Remarques :
- OData appelle *option* les mots clés commençant par $
- Nous aurions pu ainsi ne pas écrire notre méthode *departement* ; néanmoins, le requêtageOData peut décourager certains clients.

3.c Ajout d'un médecin

Il nous reste à tester l'ajout d'un nouveau médecin. Nous pouvons simuler dans Fiddler2 le passage des variables avec une méthode POST ; c'est dans le corps de la requête que doivent figurer les valeurs passées.
Le mécanisme est très simple, il suffit de faire un POST sur medecin en envoyant les données au format XML (ou json).



Le détail du corps de la requête est le suivant :


```xml
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
<title type="text"></title>
<updated>2012-07-26T20:08:53Z</updated>
<author>
<name />
</author>
<category term="bdmedecinsModel.medecin"
scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
<content type="application/xml">
<m:properties>
<d:id m:type="Edm.Int32">52</d:id>
<d:nom>Armand</d:nom>
<d:prenom>Louis</d:prenom>
<d:adresse>29 rue Petit BEDEILLE 09230</d:adresse>
<d:tel>0523206238</d:tel>
<d:specialiteComplementaire m:null="true" />
<d:departement m:type="Edm.Int32">46</d:departement>
</m:properties>
</content>
</entry>
```


3.d Authentification et gestion des erreurs

Nous allons mettre en œuvre un système d'authentification simple basé sur le passage dans l'en-tête de la requête du client des valeurs des *username* et *password*.
Pour cela dans le fichier Web.Config, nous ajoutons ces deux nouvelles entrées :

```
<configuration>
<appSettings>
<addkey="username"value="gsb"/>
<addkey="password"value="gsb"/>
</appSettings>
```

Remarque : des valeurs cryptées MD5 pourraient être utilisées.

Ajoutons une classe globale ; dans le fichier *Global.Asax.cs*, nous vérifions pour chaque requête la validité des en-têtes http :

```
protectedvoidApplication_BeginRequest(object sender, EventArgs e)
{
        if (Request.RawUrl.ToLower().Contains("wcfdataservicegsb.svc"))
        {
                stringvalUsername = AppSettingsExpressionBuilder.GetAppSetting("username").ToString();
        stringvalPassword = AppSettingsExpressionBuilder.GetAppSetting("password").ToString();
        stringuserEnTete = Request.Headers["username"];
        stringmdpEnTete = Request.Headers["password"];
        if (!(userEnTete == valUsername&&mdpEnTete == valPassword))
        {
                Server.Transfer("default.aspx");
        }
        }
}
```

Remarque :

- Nous récupérons les entrées du fichier Web.Config grâce à la classe AppSettings.ExpressionBuilder pour laquelle il faut ajouter
- `usingSystem.Web.Compilation;`

Pour gérer les erreurs, nous utilisons la méthode :

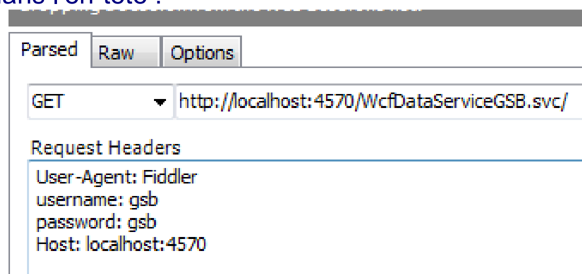
```
protectedvoidApplication_Error(object sender, EventArgs e)
{
Server.Transfer("default.aspx");
}
```

Il ne nous reste plus qu'à créer le fichier default.aspx évoqué :

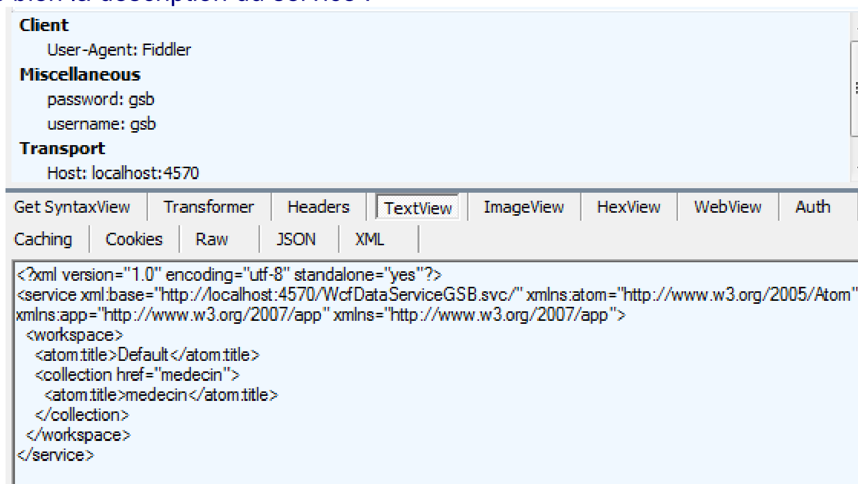
```
protectedvoidPage_Load(object sender, EventArgs e)
{
string s = "Vous n'êtes pas connecté";
if(Server.GetLastError() != null)
s =Server.GetLastError().Message;
lblErreur.Text = s;
}
```

Testons l'authentification avec Fiddler2 :

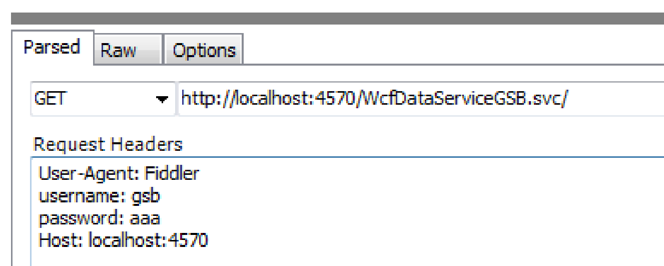
Avec les bonnes valeurs dans l'en-tête :



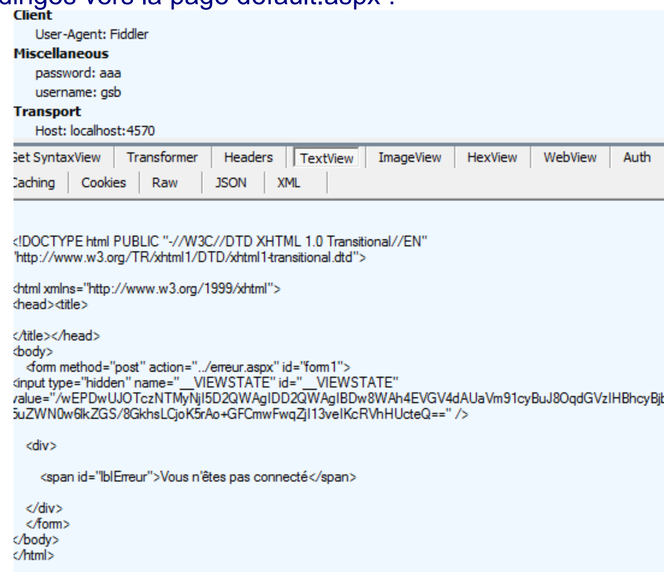
Nous obtenons bien la description du service :



Avec des valeurs erronées :



Nous sommes bien redirigés vers la page default.aspx :



Annexe : gestion de plusieurs contextes

Nous avons décidé ici d'associer un contexte distinct à chaque (future) session d'un client au service web.
Dans un environnement multi-client (c'est le cas d'un service web) il est bon que les ressources allouées à un contexte soient libérées à la fin de son exploitation.
C'est pourquoi nous allons gérer nos contextes dans une classe dédiée nommée *Contextes* qui vient ici remplacer la classe *Contexte* utilisée plus haut dans l'environnement mono-contexte.

```
namespace DAO_GSB
{
    public class Contextes
    {
       private static Dictionary<string, bdMedecinsEntities> lesContextes=null;

       static Contextes()...
       public static bdMedecinsEntities getContexte()...
       public static void supprimer(string idSession)...
    }
}
```

Notre classe contient un dictionnaire <identifiant de la session, contexte> qui est géré par les méthodes classiques (ajout, suppression). Les champs et méthodes sont statiques.

L'utilisation de ces services se fera au début d'une demande de ressource et à la fin de la demande. Ici, puisque nous serons dans une application http, nous utiliserons les événements de la classe « Global » du fichier global.asax ; nous y reviendrons plus tard.

Remarques :
La méthode *getContexte* crée (si besoin, s'il n'en existe pas un pour la session courante) un contexte et le retourne.

```
publicstaticbdMedecinsEntitiesgetContexte()
{
bdMedecinsEntitiesbd = null;
stringidSession = "session1";
if (HttpContext.Current != null)
{
idSession = HttpContext.Current.Session.SessionID;
}
if (!Contextes.lesContextes.ContainsKey(idSession))
{
bd = newbdMedecinsEntities();
Contextes.lesContextes.Add(idSession, bd);
}
else
bd = lesContextes[idSession];
return bd;
}
```

Le premier test est fait pour permettre un accès au service en dehors d'une requête http, ce qui sera le cas de nos projets de tests. La session est récupérée grâce à la classe *HttpContext* (ajouter une référence à S*ystem.Web*).

-
-

- La suppression est simple :

```
publicstaticvoid supprimer(stringidSession)
{
if (Contextes.lesContextes.ContainsKey(idSession))
Contextes.lesContextes.Remove(idSession);
}
```

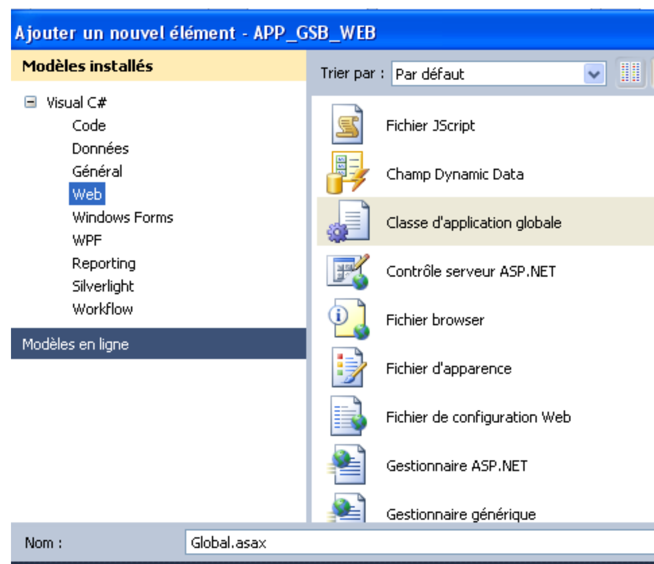
- Par contre la déclaration d'un constructeur statique est moins courante. Ce constructeur est appelé automatiquement et une seule fois dès qu'on demande un service à la classe. Il doit être déclaré *static* sans niveau de visibilité :

```
staticContextes()
{
if(lesContextes==null)
lesContextes = newDictionary<string,bdMedecinsEntities>();
}
```

La gestion du pool de contextes n'est pas obligatoire, elle permet ici de s'assurer de la bonne destruction des objets inutiles.

Revenons à notre projet d'application APP_GSB_WEB, nous allons ajouter la gestion des contextes EF (du projet DAO_GSB). Nous avions prévu de gérer un pool de contextes et de supprimer (en fin de session) le contexte du client web.

C'est l'application APP_GSB_WEB qui aura cette responsabilité. Ajoutons un nouvel élément de type « Classe globale »



Si nous ouvrons le *code behind* de la classe Global.asax, nous voyons les signatures des méthodes associées à divers événements de l'application ou d'une session.

C'est dans la méthode associée à la fin d'une session que nous allons gérer la destruction du contexte de données, après avoir ajouté la clause *using DAO_GSB* :

```
protectedvoidSession_End(object sender, EventArgs e)
{
stringidSession = HttpContext.Current.Session.SessionID;
Contextes.supprimer(idSession);
}
```

Nous avons terminé pour le service WCF et sa gestion de plusieurs contextes.

4. Déploiement de l'application REST

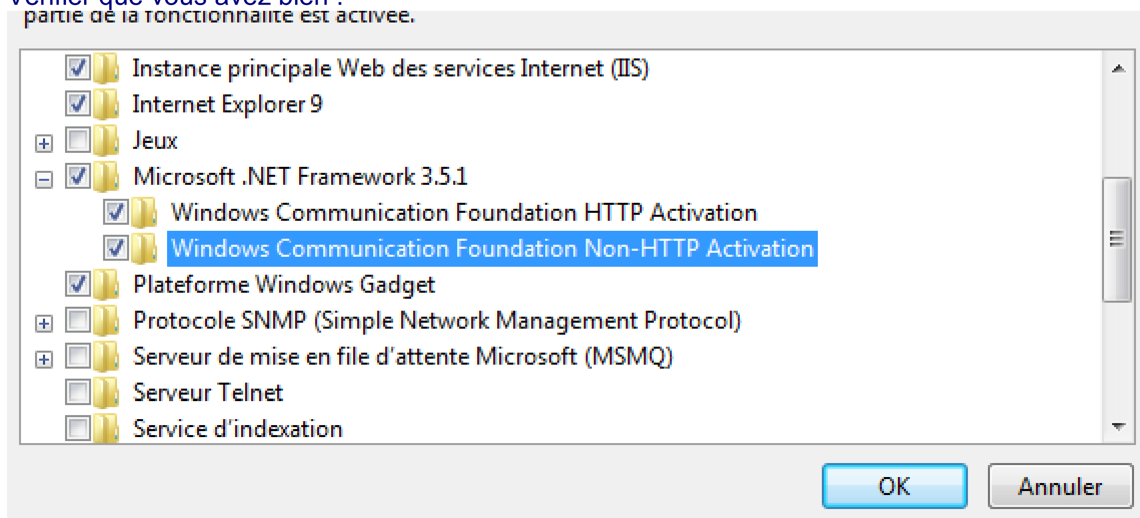
Deux points sont à configurer, le serveur IIS et l'installation des fichiers de l'application.

- Configuration des fonctionnalités Windows

MSDN propose une configuration WCF[4] pour le serveur IIS :

4 http://msdn.microsoft.com/fr-fr/library/bb675150.aspx

Vérifier que vous avez bien :



- Configuration de IIS

Il faut installer la prise en charge de WCF dans IIS en exécutant dans une console, en tant qu'administrateur :
C:\WINDOWS\Microsoft.NET\Framework\v3.0\Windows Communication Foundation\ServiceModelReg.exe –i

- Déploiement de l'application

Les fichiers à installer sont :

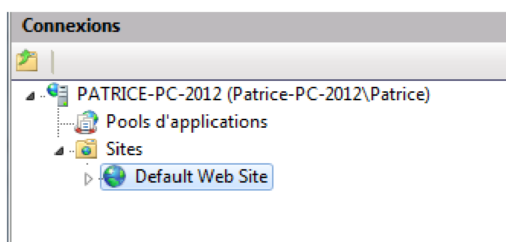
Nom	Modifié le	Type	Taille
bin	30/07/2012 09:48	Dossier de fichiers	
default	29/07/2012 08:44	ASP.NET Server Pa...	1 Ko
Global	28/07/2012 08:54	ASP.NET Server A...	1 Ko
WcfDataServiceGSB	24/07/2012 14:40	WCF Web Service	1 Ko
Web	28/07/2012 09:18	XML Configuratio...	1 Ko

- default.aspx (sans son fichier de code behind -default.aspx.cs- qui est compilé).
- global.aspx (idem)
- WcfDataServiceGSB (idem)
- Web.Config

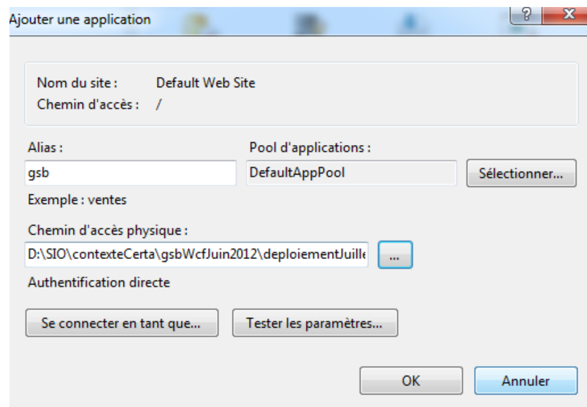
Dans le répertoire bin (obligatoirement *bin*) se trouvent les deux dll, l'une du projet REST et l'autre de la couche de mapping DAO_GSB.dll.

Placer ces fichiers et répertoire dans un répertoire (par exemple ServiceWcf) n'importe où sur le disque (donc pas nécessairement dans le répertoire de publication).

Dans l'outil d'administration de IIS (inetMgr.exe), à partir de la fenêtre de gauche :



Ajouter une application en précisant l'alias et le répertoire physique :



Pour tester, sélectionner l'application à gauche (gsb) et demander à parcourir (à droite).

Erreurs éventuellement rencontrées :

Erreur 1

Impossible de charger le type 'System.ServiceModel.Activation.HttpModule' à partir de l'assembly 'System.ServiceModel, Version 3.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'
Cette erreur se produit si .NET Framework 4 est installé et que la fonctionnalité Activation HTTP .NET Framework 3.5 WCF est activée. Pour résoudre ce problème, exécutez la ligne de commande suivante depuis l'invite de commandes Visual Studio 2010 :
Solution :
aspnet_regiis.exe -i -enable
(à exécuter en tant qu'admin)

Erreur 2

Pb de clé « target Framework » non reconnue
Solution :
modifier dans IIS *defaultAppPool*le framework => 4.0

Erreur 3

HTTP Error 500.19 - Internal Server Error

Cette erreur peut être due à différentes manipulations dans l'explorateur (modification de nom de répertoire, suppression) mal intégrées par IIS.
Solution :
Supprimer les fichiers et le répertoire de l'application et recommencer l'installation.