



# COURS .NET AVANCÉ

Lemettre Arnaud

[Arnaud.lemettre@gmail.com](mailto:Arnaud.lemettre@gmail.com)

# SOMMAIRE



- ➡ Introduction
- ➡ Unity
- ➡ MEF
- ➡ AutoMapper
- ➡ OData

# INTRODUCTION



- ➡ Maintenant, que nous avons vu les bases de la programmation .NET nous allons pouvoir aborder des concepts plus élaborés permettant de simplifier certaines parties de nos programmes.
- ➡ Attention cependant ceci restera un simple aperçu, ces frameworks offrent beaucoup plus de possibilités.



# UNITY



Microsoft  
**patterns & practices**  
proven practices for predictable results

# UNITY



➡ C'est quoi ?

- ▶ Unity Application Block est un Lightweight qui permet de définir un container d'injection de dépendance avec le support de l'injection sur :
  - ▶ Constructeur
  - ▶ Propriétés
  - ▶ Appel de méthodes

<http://unity.codeplex.com/>

# UNITY



➡ Quels sont les avantages ?

- ▶ L'injection de dépendance permet d'avoir un couplage souple avec les classes
- ▶ Facilite l'utilisation de tests unitaires (Mock)
- ▶ Permet d'intercepter les méthodes

➡ Quels sont les inconvénients ?

- ▶ Programmation haut niveau
- ▶ Pas facile à utiliser
- ▶ Coûteux en terme d'exécution

# UNITY



➡ Cela répond à quels problèmes ?

- ▶ Permet de changer facilement d'implémentation
  - ▶ Test unitaire => simule l'exécution d'une fonction
  - ▶ Permet de changer une classe mock en une classe réelle - travail en équipe.
- ▶ Permet de rajouter du code sans interagir avec le code métier (log, watcher, ...)

# UNITY



➔ Quel en est le fonctionnement ?

- ▶ Mode IOC (Mock, ...)

1. Il faut initialiser le container

- ▶ Soit par fichier de configuration (changement à la volée)

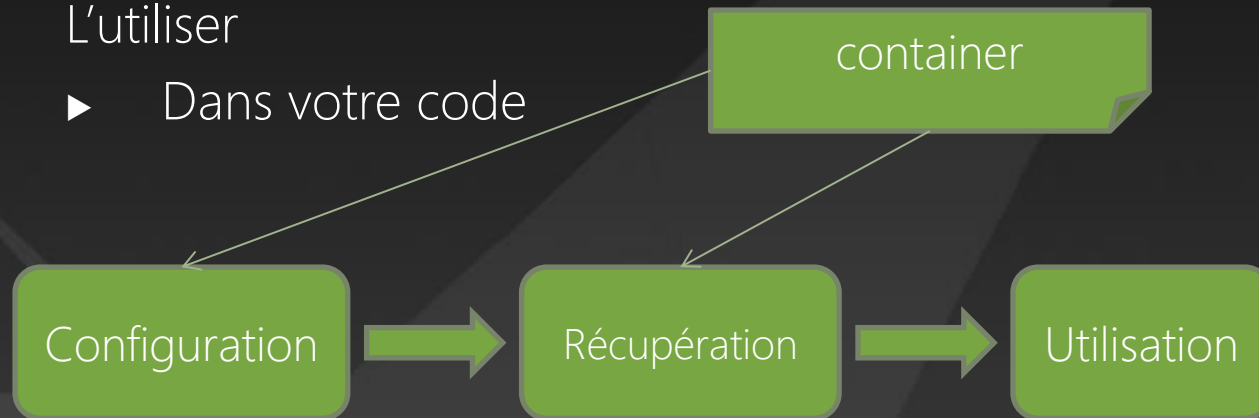
- ▶ Soit par code

2. Récupérer le type

- ▶ A l'aide du resolver

3. L'utiliser

- ▶ Dans votre code





# UNITY



➔ Quel en est le fonctionnement ?

- ▶ Mode IOC (Mock, ...)
- ▶ Il faut initialiser le container

```
public class Container : UnityContainer
{
    public void Configure()
    {
        /* Register types into the container */
        this.RegisterType<Recorder.IRecord, ConsoleUnity.Recorder.Recorder>();
    }
}
```

```
public interface IRecord
{
    string Something { get; set; }

    bool Play();
    string Task();
}
```

Le contrat à implémenter

```
public class Recorder : IRecord
{
    private string _something;
    public string Something
    {
        get { return _something; }
        set { _something = value; }
    }
    public Recorder() { _something = "toto"; }
    public bool Play()
    {
        _something = "john";
        return true;
    }
    public string Task() { return "indochine"; }
}
```

Le contrat  
implémenté





# UNITY

➡ Quel en est le fonctionnement ?

- ▶ Mode IOC (Mock, ...)
  - ▶ Récupérer le type
    - ▶ A l'aide du resolver

```
IRecord record3 = contain.Resolve<IRecord>();
```

- ▶ L'utiliser
  - ▶ Dans votre code

```
record3.Play();
```

# UNITY



## ➔ Pourquoi Mock ?

- ▶ Pour changer d'implémentation, il n'y a qu'à modifier le configure
  - ▶ Par exemple

```
this.RegisterType<Recorder.IRecord, ConsoleUnity.Recorder.Recorder2>();
```

Recorder2 correspond à une autre implémentation, mais avec le même contrat. De ce fait, la modification dans le configure impactera l'ensemble de l'application

Ceci implique que la décision d'utiliser unity est connue depuis le début.

Modifier par le code implique un certain nombre de contraintes surtout pour certains changements.

Par exemple : implémentation d'une connexion avec et sans proxy

# UNITY



➡ Modification au travers du fichier de configuration ?

► Le fichier app.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection, Microsoft.Practices.
      Unity.Configuration"/>
  </configSections>
  <unity>
    <typeAliases>
      <typeAlias alias="IRecord" type="ConsoleUnity.Recorder.IRecord, ConsoleUnity" />
    </typeAliases>
    <containers>
      <container>
        <types>
          <type type="IRecord" mapTo="ConsoleUnity.Recorder.Recorder,ConsoleUnity">
          </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

# UNITY



- ➡ Modification au travers du fichier de configuration ?
  - ▶ Il faut également modifier la configuration du container :

```
public class Container : UnityContainer
{
    public void Configure()
    {
        var section = (UnityConfigurationSection)ConfigurationManager.
GetSection("unity");
        section.Configure(this);
    }
}
```

# UNITY



## ➡ Injection de dépendances

- ▶ Un exemple d'injection avec le constructeur  
+ Singleton

```
class LogService : IMyService
{
    public void WriteToLog(string message)
    {
        Debug.WriteLine("Second debug => " + message);
    }
}
```

```
public interface IMyService
{
    void WriteToLog(string message);
}
```

```
public class LoggingService : IMyService
{
    public void WriteToLog(string message)
    {
        Debug.WriteLine(message);
    }
}
```

```
public class ContainerInjection : UnityContainer
{
    public void Configure()
    {
        // 1er service de log
        LoggingService firstLogService = new LoggingService();
        // 2e service de log
        LogService secondLogService = new LogService();
        //on enregistre la 1er instance
        this.RegisterInstance<IMyService>(firstLogService);
    }
}
```

# UNITY



➡ A la sortie ...

```
public class CustomerService
{
    private IMyService _instance;
    public CustomerService(IMyService myServiceInstance) { _instance = myServiceInstance; }

    public void Trace(string msg)
    {
        _instance.WriteToLog(msg);
    }
}
```

```
private static void testDepend()
{
    SimpleConstrut.ContainerInjection container = new SimpleConstrut.ContainerInjection();

    container.Configure();
    CustomerService myInstance = container.Resolve<CustomerService>();
    myInstance.Trace("msg");
}
```



Sortie console

```
msg
Second debug => msg
```

# UNITY



➡ Un peu plus compliqué ...

- ▶ Interception de méthodes
- ▶ Le but étant de construire un proxy qui s'occupera des appels

```
class Intercept : IInterceptionBehavior
{
    public IEnumerable<Type> GetRequiredInterfaces() { return Type.EmptyTypes; }

    public IMethodReturn Invoke(IMethodInvocation input, GetNextInterceptionBehaviorDelegate getNext)
    {
        Debug.WriteLine("before call");
        var methodReturn = getNext().Invoke(input, getNext);
        Debug.WriteLine("after call");
        Debug.WriteLine("result = " + methodReturn.ReturnValue);
        return methodReturn;
    }

    public bool WillExecute { get { return true; } }
}
```

Il faut implémenter l'interface

Méthode qui s'occupera de l'appel proxy



# UNITY



➡ Un peu plus compliqué ...

► Pour l'appel :

```
private static void TestInterception()
{
    ConfInterception.ContainerInterception contain = new ConfInterception.ContainerInterception();
    contain.Configure();
    IRecord record = contain.Resolve<IRecord>();

    contain.RegisterInstance<IRecord>(record);

    Console.WriteLine("First call");
    Console.WriteLine(record.Something);
    record.Play();

    Console.WriteLine(record.Something);

    Console.WriteLine("Second call");
    IRecord record2 = contain.Resolve<IRecord>();
    Console.WriteLine(record2.Something);
    Console.ReadLine();
}
```

Sortie console

```
First call
toto
john
Second Call
john
```

# UNITY



➡ Pour faciliter le code et la lisibilité nous pouvons également coder des attributs

```
public class LogAttribute : HandlerAttribute
{
    private string _param;

    public LogAttribute() { }

    public LogAttribute(string param) { _param = param; }

    public override ICallHandler CreateHandler(Microsoft.Practices.Unity.IUnityContainer container)
    {
        return new LogCallHandler();
    }
}
```

# UNITY



➡ Classe du proxy permettant d'appeler la méthode :

```
public class LogCallHandler : ICallHandler
{
    public IMethodReturn Invoke(IMethodInvocation input, GetNextHandlerDelegate getNext)
    {
        string className = input.MethodBase.DeclaringType.Name;
        string methodName = input.MethodBase.Name;

        string preMethodMessage = string.Format("{0}.{1}()", className, methodName);
        System.Diagnostics.Debug.WriteLine(preMethodMessage);

        // Call the method that was intercepted
        IMethodReturn msg = getNext()(input, getNext);

        string postMethodMessage = string.Format("{0}.{1}() -> {2}", className, methodName, msg.ReturnValue);
        System.Diagnostics.Debug.WriteLine(postMethodMessage);

        return msg;
    }
}
```

# UNITY



- ➡ Pour utiliser notre attribut nous pouvons :
  - ▶ Soit le placer sur une classe
  - ▶ Soit sur une méthode
- ➡ Il faut placer cet attribut sur l'interface

```
[Log.Log("Recorder")]  
public interface IRecord  
{  
    string Something { get; set; }  
  
    bool Play();  
    string Task();  
}
```

OU

```
public interface IRecord  
{  
    string Something { get; set; }  
    [Log.Log("Recorder")]  
    bool Play();  
    string Task();  
}
```

# UNITY



➡ Pour le container, il faut le configurer de cette manière :

```
public class Container : UnityContainer
{
    public void Configure()
    {
        //enregistrement du type
        this.RegisterType<Recorder.IRecord, ConsoleUnity.Recorder.Recorder>();
        //on déclare une extension d'interception
        this.AddNewExtension<Interception>();
        //on configure l'interception pour le type
        this.Configure<Interception>().SetInterceptorFor(typeof(Recorder.IRecord),
                                                         new TransparentProxyInterceptor());
    }
}
```

Ensuite nous pouvons appeler comme précédemment les fonctions

# UNITY



- ➡ Une autre façon de placer les intercepteurs et d'utiliser les politiques. De cette manière vous n'êtes plus obligé de placer des attributs sur les interfaces.
- ➡ Pour cela il faut modifier le Configure :

```
public void ConfigurePolicy()
{
    this.RegisterType<Recorder.IRecord, ConsoleUnity.Recorder.Recorder>();
    // enregistre la règle
    this.RegisterType<IMatchingRule, AnyMatchingRule>("AnyMatchingRule");
    this.RegisterType<ICallHandler, LogCallHandler>("LogCallHandler");
    //on crée l'association avec la règle
    this.AddNewExtension<Interception>();
    this.Configure<Interception>().AddPolicy("LogPolicy")
        .AddMatchingRule("AnyMatchingRule")
        .AddCallHandler("LogCallHandler");
    //on configure l'interception pour le type
    this.Configure<Interception>().SetInterceptorFor(typeof(Recorder.IRecord),
        new TransparentProxyInterceptor());
}
```

# UNITY



➡ Les règles s'implémentent de la façon suivante :

```
public class AnyMatchingRule : IMatchingRule
{
    public bool Matches(MethodBase member)
    {
        if (member.Name == "Play")
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Implémentation de l'interface  
*IMatchingRule*

Règle de gestion permettant d'  
appeler ou non l'interception de la  
méthode



MEF



# MEF



➡ Qu'est ce que MEF ?



- ▶ Manage Extensibility Framework
- ▶ Site des sources : <http://mef.codeplex.com>
- ▶ Simplifie la création d'extensions d'applications avec des capacités de découvertes et d'extensions.

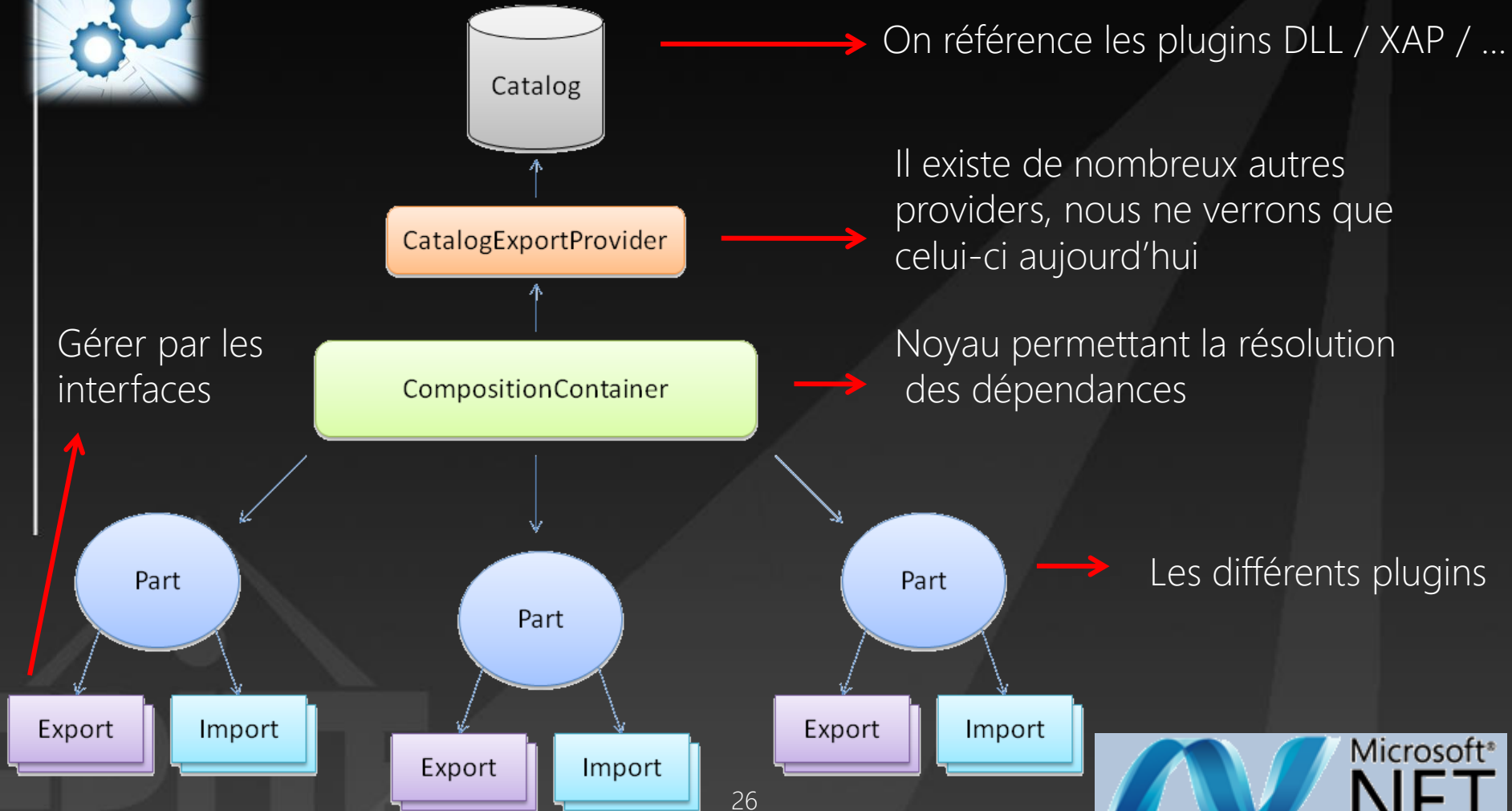
➡ Pour résoudre quels problèmes ?

- ▶ Fournir une implémentation standard de gestion de plugin
- ▶ Permet de localiser et charger à la volée les extensions
- ▶ Permet de rajouter des métadonnées pour faciliter la recherche d'extension

# MEF



➡ Comment ça marche ?

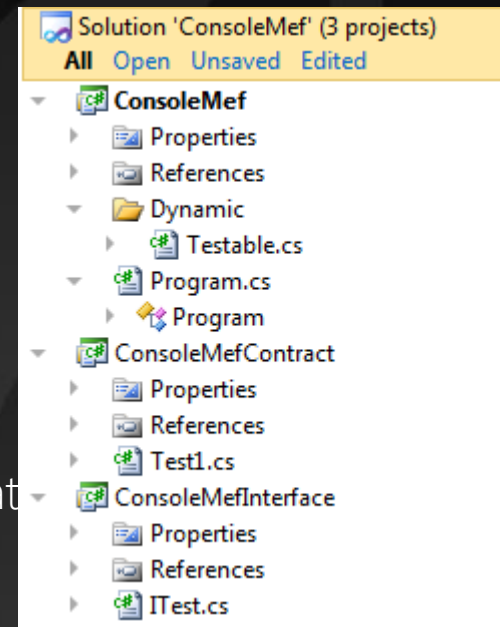
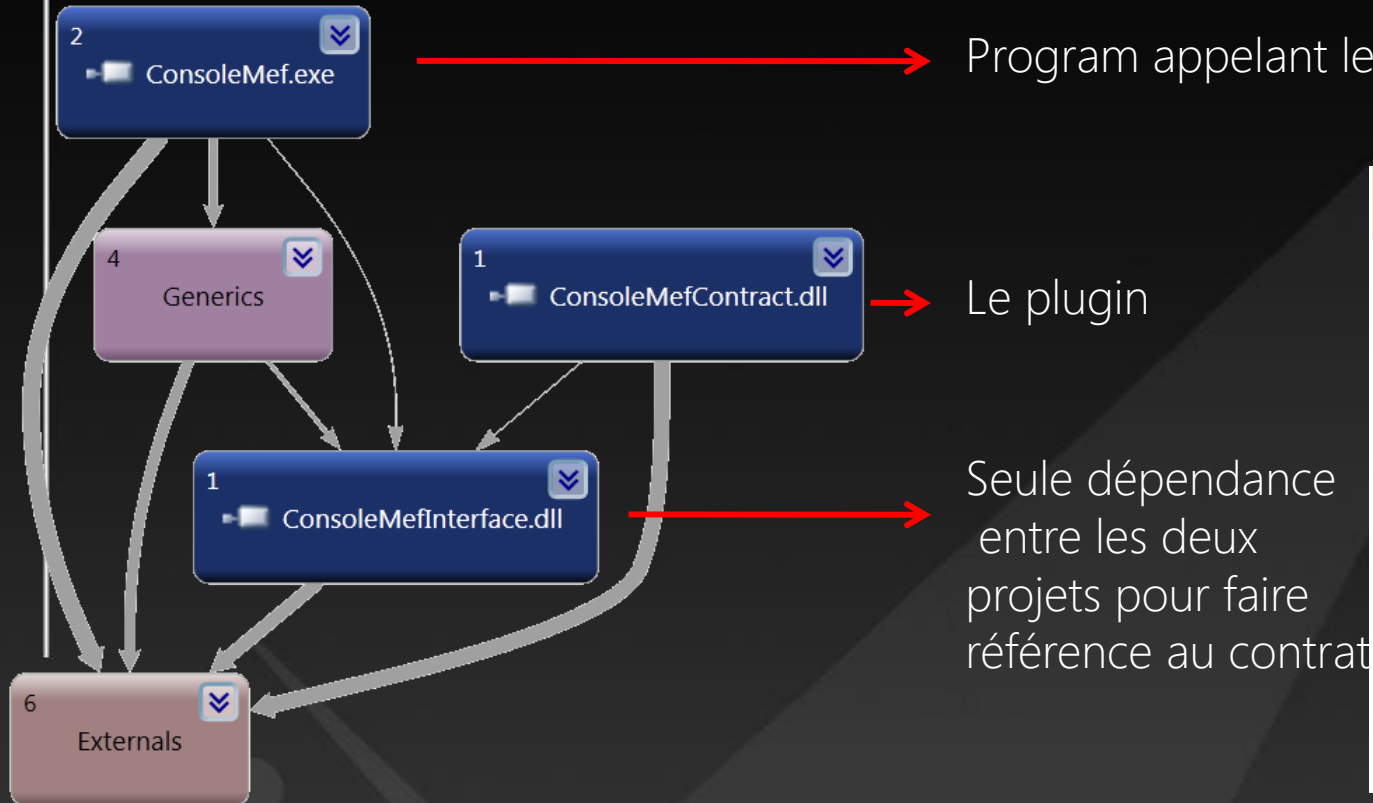


# MEF



## ➡ Cas concret :

- ▶ Dans cet exemple, nous allons créer un plugin qui viendra calculer une moyenne



*Sous Visual Studio*

# MEF



- ➔ Dans un 1er temps, il faut déterminer l'interface qui va nous permettre de communiquer entre le « Core » et le plugin

```
public interface ITest
{
    bool LaunchTest(string student, XDocument trace);
}
```

# MEF



- ➔ Comment réaliser un plugin ?
  - ▶ Il faut rajouter les interfaces suivantes :

• `ConsoleMefInterface`  
• `System.ComponentModel.Composition`

Mot clé permettant de faire connaître la classe comme plugin. Prend en paramètre le contrat du plugin

Contient les classes pour MEF

MetaData qui seront exposés pour le catalogue de plugins

```
[Export(typeof(ITest))]  
[ExportMetadata("TP", 1)]  
public class Test1 : ITest  
{  
    public bool LaunchTest(string student, System.Xml.Linq.XDocument trace)  
    {  
        trace.Root.Add(new XElement("mark"));  
        return true;  
    }  
}
```

# MEF



- ➡ Dans cet exemple, nous allons voir comment explorer un répertoire de plugin (dans un vrai développement il y aurait une façon plus élégante de le faire)
- ➡ Déléguer cette exploration à une classe :

Indique que potentiellement nous allons avoir plus d'un plugin à découvrir

```
public class Testable
{
    private DirectoryCatalog _directoryCatalog;
    [ImportMany]
    public Lazy< ITest, IDictionary<string, object>>[] LoadedTests { get; set; }
    public bool Run()
    {
        Compose();
        bool res = false;

        return res;
    }

    private void Compose()
    {
        string currentDirectory = new FileInfo(Assembly.GetCallingAssembly().Location).Directory.ToString();
        _directoryCatalog = new DirectoryCatalog(currentDirectory);
        AggregateCatalog catalog = new AggregateCatalog(
            new AssemblyCatalog(Assembly.GetExecutingAssembly()), _directoryCatalog );
        var container = new CompositionContainer(catalog);
        container.ComposeParts(this);
    }
}
```

La collection Lazy n'est instanciée que lorsque value est appelée

On récupère les plugins dans le répertoire

On demandera la composition au container pour la résolution



# MEF



➡ Pour faire appel aux différents plugins chargés :

```
public bool Run()
{
    Compose();
    bool res = false;
    XDocument trace = XDocument.Parse("<root/>");
    foreach (var item in LoadedTests)
    {
        if (Convert.ToInt32(item.Metadata["TP"]) == 2)
        {
            res = item.Value.LaunchTest("lemett_a", trace);
        }
    }
    return res;
}
```

Metadata permettant de regarder les informations du plugin

Value crée l'instance qui permettra d'exécuter la fonction



AUTOMAPPER

autoxmapper



# AUTOMAPPER



- ➡ Dans le cadre de vos développements, vous avez pu voir que l'on devait transcrire des objets en d'autres types d'objets :
  - ▶ Table en DBO
  - ▶ Ou tout simplement les mêmes objets mais juste des namespaces différents.
  
- ➡ L'une des solutions est d'écrire des méthodes qui font les conversions :  
généralement cela prend du temps et il faut tester

Site : <https://github.com/AutoMapper/AutoMapper>

# AUTOMAPPER



➡ Les deux classes :

```
namespace ConsoleAutoMapper.Dbo
{
    public class Car
    {
        public string Name { get; set; }
        public int Speed { get; set; }
    }
}
```

```
namespace ConsoleAutoMapper.Service
{
    public class Car
    {
        public string Name { get; set; }
        public int Speed { get; set; }
        public string Color { get; set; }
    }
}
```

# AUTOMAPPER



- ➡ Pour faire les conversions :
  - ▶ Il faut paramétrer la classe :

```
private static void Conf()
{
    // réinitialise tous les mappages
    Mapper.Reset();
    // effectue le mappage
    Mapper.CreateMap<Dbc.Car, Service.Car>();
}
```

- ➡ Pour appeler la conversion :

```
Dbc.Car car = new Dbc.Car() { Name = "saxo", Speed = 90 };
//conversion simple
Service.Car carConv = Mapper.Map<Dbc.Car, Service.Car>(car);
```

```
//conversion null
Service.Car carTmp = Mapper.Map<Dbc.Car, Service.Car>(null);
```

Si on appelle avec une valeur NULL  
la valeur retournée sera NULL

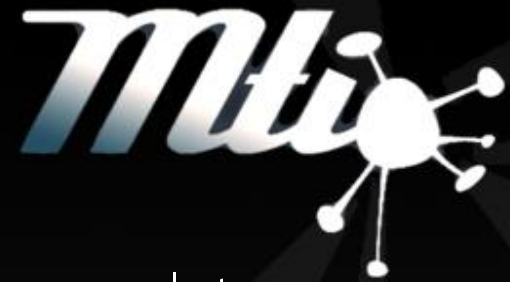
# AUTOMAPPER



- ➡ Automapper permet de faire du mappage « intelligent ».
- ➡ Par exemple avec le précédent mapping on peut également convertir une liste directement :

```
//conversion liste  
List<Service.Car> res = AutoMapper.Mapper.Map<List<Db.Car>, List<Service.Car>>(list);
```

# AUTOMAPPER



- ➡ Dans l'exemple donné le mapping n'est pas complet :
  - ▶ Le propriété Color du second objet n'est pas mappé
- ➡ Automapper fournit des fonctions permettant de traiter le problème avant ou après coup (à définir au mappage) :

```
Mapper.CreateMap<Db.Car, Service.Car>().BeforeMap((x, y) => y.Color = "red");
```

# AUTOMAPPER



- ➡ La configuration peut se faire également à base de profile

```
public class ProfileConfiguration : AutoMapper.Profile
{
    public override string ProfileName
    {
        get
        {
            return "myProfile";
        }
    }
    protected override void Configure()
    {
        CreateMap<Dbó.Car, Service.Car>().BeforeMap((x, y) => y.Color = "red");
    }
}
```

- ➡ On charge la configuration de cette manière :

```
private static void ConfWithProfile()
{
    Mapper.Reset();
    Mapper.AddProfile<Profile.ProfileConfiguration>();
}
```



ODATA



Open Data Protocol


# ODATA



➡ Qu'est ce que OData ?

- ▶ Open data

➡ Pourquoi Odata ?

- ▶ venue  du constat que beaucoup de données sont stockées mais jamais exploitées ou partagées

## Point Clefs

- Facilement utilisable par tout le monde
- Modèle unifié quelque soit le schéma
- Couche d'abstraction permettant l'interopérabilité

➡ Pourquoi ne pas utiliser les web services ?  
- Pas si interopérable que cela ... .Net – Java / Soap v1.1 – Soap v2.2 / Transfert binaire

Adresse : <http://www.odata.org/>



# ODATA



- ➡ Les spécificités techniques d'ODATA :
- ➡ Le format de sortie doit être de l'ATOM ou JSON
  - ▶ ATOM : format de balise basé sur du XML utilisé notamment pour les flux RSS
  - ▶ JSON : (JavaScript Object Notation) est un format de données textuel, générique. Il permet de représenter de l'information structurée sous forme de paire clef / valeur.
- ➡ Le système de requête au travers de conventions d'url permettant le post, get, put, delete, ainsi que des systèmes de navigation et de filtre.
- ➡ La navigation se décompose comme suit :
  - ▶ Un sommaire regroupant l'ensemble des services
  - ▶ Les services
    - ▶ Workspace (si on simplifie, cela correspond aux tables de la base de données)
    - ▶ Collections de données (les listes d'enregistrements selon les tables)

# ODATA



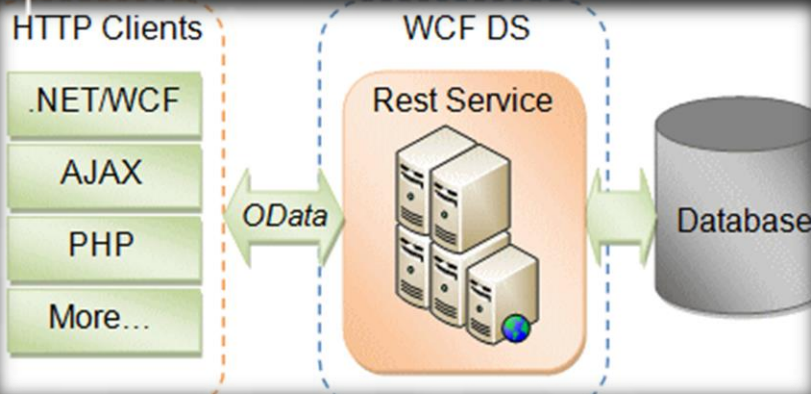
## ➡ Que vient faire Microsoft ?

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
- <edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/edmx/2007/03" />
- <edmx:DataServices xmlns:m="http://schemas.microsoft.com/odata/2007/08" />
- <Schema Namespace="NetflixCatalog.Model" xmlns:d="http://schemas.microsoft.com/odata/2007/08" />
- <EntityType Name="Genre">
- <Key>
- <PropertyRef Name="Name" />
- </Key>
- <Property Name="Name" Type="Edm.String" Nullable="false" />
- <NavigationProperty Name="Titles" Relationship="NetflixCatalog.Model.Titles" />
- </EntityType>
- <EntityType Name="Language">
- <Key>
- <PropertyRef Name="Name" />
- </Key>
- <Property Name="Name" Type="Edm.String" Nullable="false" />
- <NavigationProperty Name="Titles" Relationship="NetflixCatalog.Model.Titles" />
- </EntityType>
```

A cette description, Microsoft a répondu en implémentant WCF Data Services. WCF Data services, qui fut à l'origine du projet, permet d'exposer ces données selon l'OData d'une manière simple.

Au travers d'un template de projet sous Visual Studio, il devient possible d'exposer ces données selon le modèle unifié d'OData. La compilation de ce dernier permet donc de les exposer à partir d'une adresse de type :

[http://localhost/service\\_odata.svc/](http://localhost/service_odata.svc/)

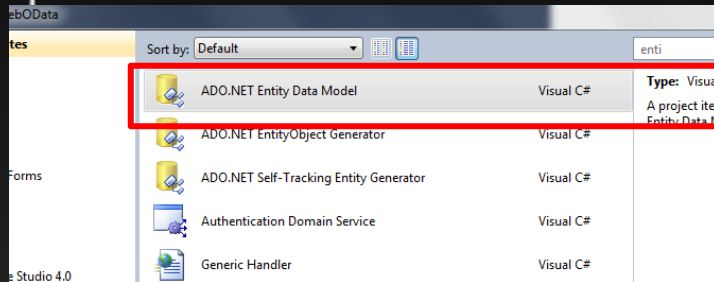


# ODATA



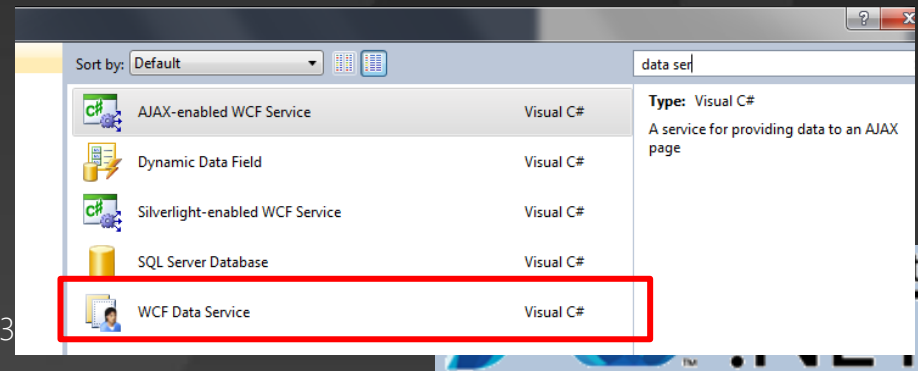
## ➔ Création sous visual studio

- ▶ Il faut d'abord créer un projet web (WebApplication par exemple)
- ▶ Puis insérer un nouvel item (Entity Framework):



Puis faire le mappage avec la base de données.

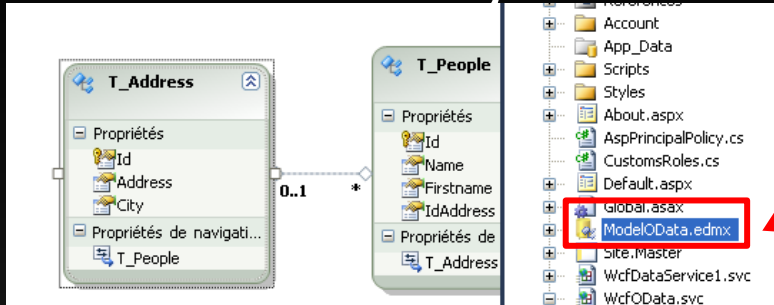
- ▶ Ensuite nous pouvons insérer l'item data service :



# ODATA



## ➔ Paramétrage du service



Liaison vers votre source de données, ici notre entity framework

```
public class WcfDataService1 : DataService< /* TODO: source de données */ >
{
    // Cette méthode n'est appelée qu'une seule fois pour initialiser les stratégies au niveau
    // des services.
    public static void InitializeService(DataServiceConfiguration config)
    {
        // TODO: définissez des règles de sécurité
        config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
    }
}
```

```
public class WcfOData : DataService< ODataEntities >
```

# ODATA



## ➡ Paramétrage du service

### ► En ce qui concerne la sécurité :

```
public static void InitializeService(DataServiceConfiguration config)
{
    //configuration de la pagination
    config.SetEntitySetPageSize("*", 2);

    // soit on veut faire pour une table en particulier
    //config.SetEntitySetAccessRule("T_People", EntitySetRights.AllRead);
    //config.SetServiceOperationAccessRule("MyServiceOperation", ServiceOperationRights.All);

    //soit pour toutes les tables
    config.SetEntitySetAccessRule("*", EntitySetRights.AllRead);
    config.SetServiceOperationAccessRule("*", ServiceOperationRights.All);

    config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
}
```

Indique que pour toutes les tables on souhaite n'avoir que 2 résultats à la fois par requête

Soit on indique par table et opération les actions possibles

Sinon pour l'ensemble des tables

# ODATA



## ➡ Requêtes simples

- ▶ Ces requêtes peuvent être exécutées dans un

[http://localhost:1395/WcfOData.svc/T\\_People](http://localhost:1395/WcfOData.svc/T_People)

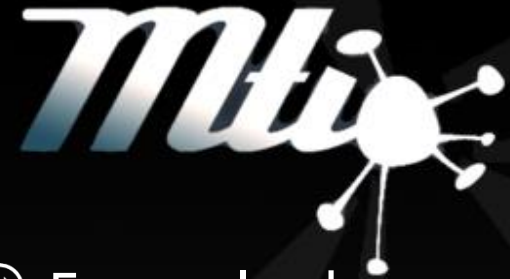
[http://localhost:1395/WcfOData.svc/T\\_People\(2L\)](http://localhost:1395/WcfOData.svc/T_People(2L))

[http://localhost:1395/WcfOData.svc/T\\_People?\\$filter=Name eq 'siemens'](http://localhost:1395/WcfOData.svc/T_People?$filter=Name eq 'siemens')

Résultat :

```
<title type="text">T_People</title>
<id>http://localhost:1395/WcfOData.svc/T_People</id>
<updated>2011-04-20T17:26:54Z</updated>
<link rel="self" title="T_People" href="T_People" />
<entry>
  <id>http://localhost:1395/WcfOData.svc/T_People(2L)</id>
  <title type="text"></title>
  <updated>2011-04-20T17:26:54Z</updated>
  <author>
    <name />
  </author>
  <link rel="edit" title="T_People" href="T_People(2L)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/T_Address" type="application/atom+xml;type=entry" title="T_Address" href="T_People(2L)/T_Address" />
  <category term="ODataModel.T_People" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:Id m:type="Edm.Int64">2</d:Id>
      <d:Name>lemettre</d:Name>
      <d:FirstName>arnaud</d:FirstName>
      <d:IdAddress m:type="Edm.Int64">1</d:IdAddress>
    </m:properties>
  </content>
</entry>
<entry>
  <id>http://localhost:1395/WcfOData.svc/T_People(3L)</id>
  <title type="text"></title>
  <updated>2011-04-20T17:26:54Z</updated>
  <author>
    <name />
  </author>
  <link rel="edit" title="T_People" href="T_People(3L)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/T_Address" type="application/atom+xml;type=entry" title="T_Address" href="T_People(3L)/T_Address" />
  <category term="ODataModel.T_People" scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:Id m:type="Edm.Int64">3</d:Id>
      <d:Name>siemens</d:Name>
      <d:FirstName>herve</d:FirstName>
      <d:IdAddress m:type="Edm.Int64">2</d:IdAddress>
    </m:properties>
  </content>
</entry>
<link rel="next" href="http://localhost:1395/WcfOData.svc/T_People?$skiptoken=3L" />
</feed>
```

# ODATA



Operator	Description	Example
<b>Logical Operators</b>		
Eq	Equal	/Suppliers?\$filter=Address/City eq 'Redmond'
Ne	Not equal	/Suppliers?\$filter=Address/City ne 'London'
Gt	Greater than	/Products?\$filter=Price gt 20
Ge	Greater than or equal	/Products?\$filter=Price ge 10
Lt	Less than	/Products?\$filter=Price lt 20
Le	Less than or equal	/Products?\$filter=Price le 100
And	Logical and	/Products?\$filter=Price le 200 and Price gt 3.5
Or	Logical or	/Products?\$filter=Price le 3.5 or Price gt 200
Not	Logical negation	/Products?\$filter=not endswith(Description,'milk')
<b>Arithmetic Operators</b>		
Add	Addition	/Products?\$filter=Price add 5 gt 10
Sub	Subtraction	/Products?\$filter=Price sub 5 gt 10
Mul	Multiplication	/Products?\$filter=Price mul 2 gt 2000
Div	Division	/Products?\$filter=Price div 2 gt 4
Mod	Modulo	/Products?\$filter=Price mod 2 eq 0
<b>Grouping Operators</b>		
( )	Precedence grouping	/Products?\$filter=(Price sub 5) gt 10

➞ Exemple de requêtes.

➞ Pour plus de détails

➞ <http://www.odata.org/documentation/uri-conventions>



# ODATA



## ➡ Requêtes customs

- ▶ Data Service permet également de définir ses propres requêtes.

Attribut de méthode permettant d'exposer la méthode

```
[WebGet]  
public IQueryable<T_People> GetPeople()  
{  
    return CurrentDataSource.T_People;  
}
```

Appel sur le navigateur :



# ODATA



## ➡ Requête customs

- ▶ On peut également indiquer que l'on limite le nombre de résultats retournés, et passer des paramètres

Indique le fait que l'on retourne un seul résultat

```
[SingleResult]
[WebGet]
public IQueryable<T_People> GetAuthorByFirstName(string name)
{
    IQueryable<T_People> query = from au in CurrentDataSource.T_People
                                where au.Name == name
                                select au;

    return query.Take(1);
}
```



## ➔ Les interceptors

- ▶ Permettent de rajouter de la logique métier sur les opérations.
- ▶ Mais également de la sécurité

Mot clé permettant de définir un interceptor



Nom de la table sur laquelle va s'exercer l'interceptor

```
[QueryInterceptor("T_People")]  
public Expression<Func<T_People, bool>> OnQueryInterceptorT_People()  
{  
    return c => c.Firstname != "";  
}
```

L'interceptor se déclenchera avant l'exécution de la requête sur la table

# ODATA

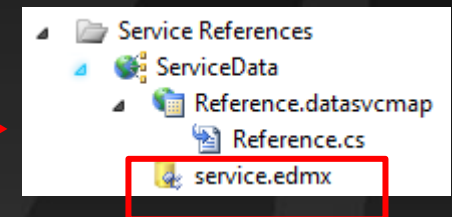
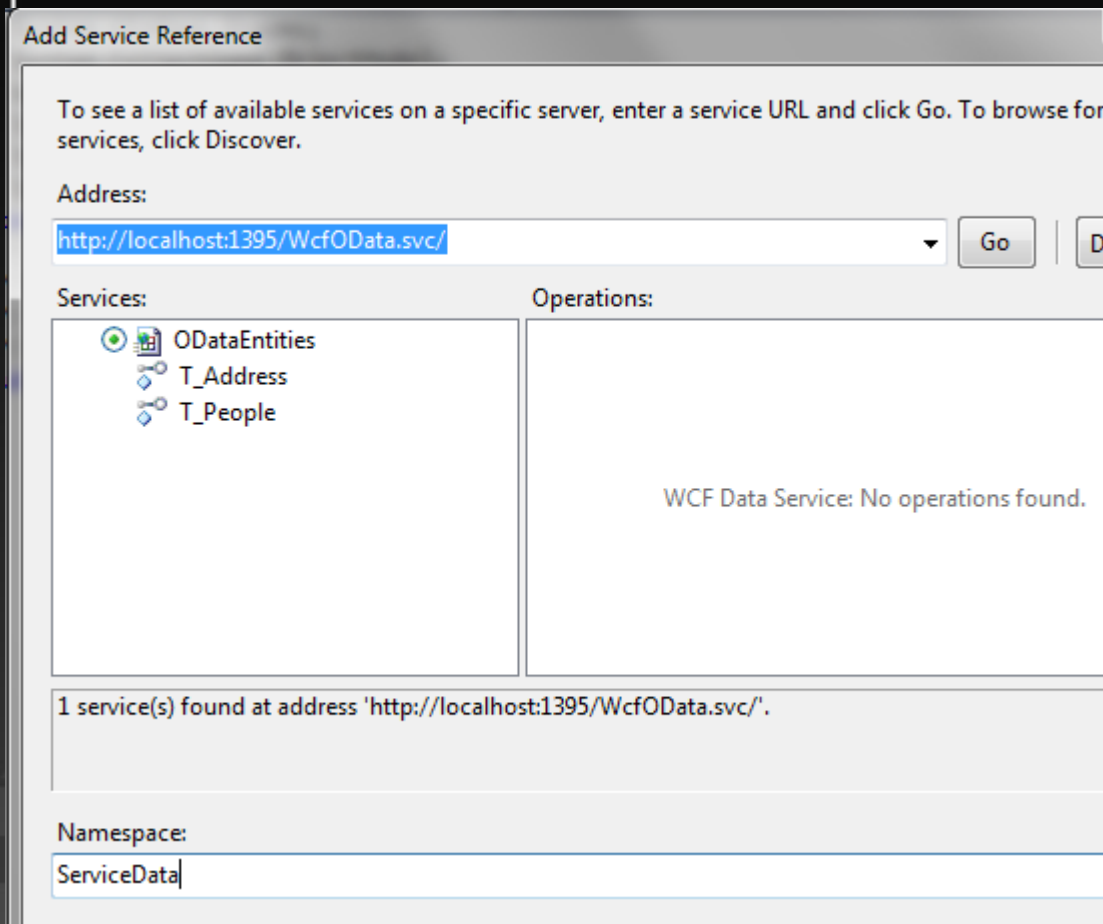


- ➡ Maintenant que nous avons un serveur basé sur le protocole OData, nous pouvons également créer un client pour l'interroger.
- ➡ Dans cet exemple, nous allons créer une console avec une simple requête.

# ODATA



➡ Il faut d'abord rajouter le service à l'application



On peut voir ici que c'est un entity framework

# ODATA



➡ Il nous faut une variable pour stocker les résultats :

```
public ObservableCollection<ServiceData.T_People> People { get; private set; }  
public DataServiceCollection<ServiceData.T_People> People2 { get; private set; }
```

La première variable permet de pouvoir observer lorsqu'un élément est ajouté ou supprimé

La seconde variable permet d'exploiter les résultats d'Odata beaucoup plus facilement comme par exemple la gestion de la pagination

Pour utiliser le service :

```
private ServiceData.ODataEntities _serviceOData;
```

# ODATA



## ➔ Comment récupérer une liste :

```
//Permet d'utiliser la pagination avec les dataServicesCollection
People2 = new DataServiceCollection<T_People>(_serviceOData.T_People);

if (People2.Continuation != null)
{
    //chargement de la requêtes pour la page suivante de façon asynchrone
    _serviceOData.BeginExecute<T_People>(People2.Continuation.NextLinkUri, (ar) =>
    {
        var peopleNextPage = _serviceOData.EndExecute<T_People>(ar);
        People2.Load(peopleNextPage);
        Debug.WriteLine("Nouveau People");
    },
    null);
}

//utilisation d'une requête simple
this.People = new ObservableCollection<T_People>(_serviceOData.T_People);
```

# ODATA



- ➡ Le fait d'utiliser data service évite d'écrire les adresses (elles sont constituées automatiquement par le moteur) Ainsi pour l'enregistrement d'une donnée :

```
private bool SavePeople()
{
    T_People people = new T_People();
    people.Firstname = "firstname";
    _serviceOData.AddObject("T_People", people);
    var result = _serviceOData.SaveChanges();

    // renvoie une liste de code si jamais on lui en envoie beaucoup
    return result.First().StatusCode == (int)HttpStatusCode.Created;
}
```

# ODATA



➔ Pour manipuler les données, il existe déjà des clients comme :

- ▶ Odata explorer
- ▶ LinqPad
- ▶ ...

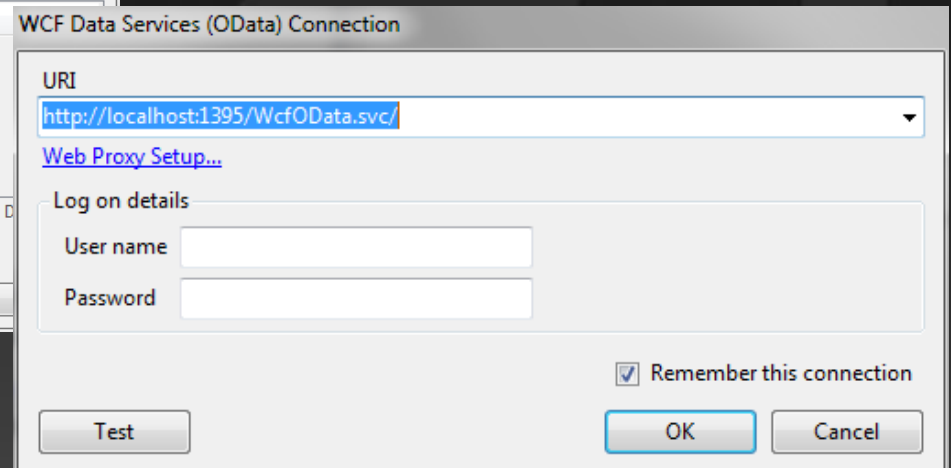
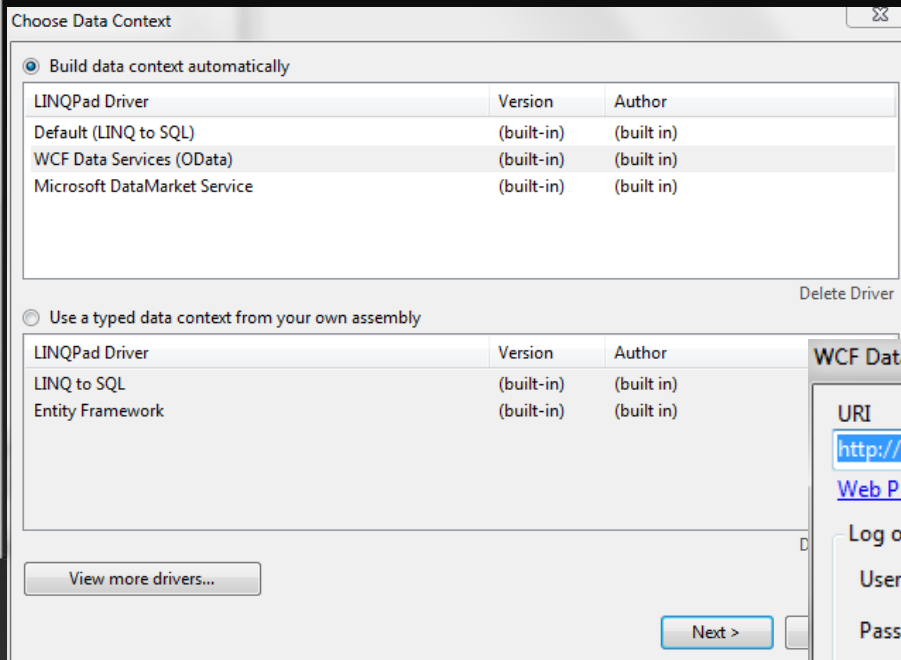


# ODATA



## ➔ Utilisation de linqPad

- Pour requêter une adresse :



# ODATA



## ➡ Utilisation de linqPad

- Pour requêter une adresse :

The screenshot shows the LINQPad 4 application. On the left, a tree view shows a connection to 'http://localhost:1395/WcfOData.svc/' with two entities: 'T\_Address' and 'T\_People'. 'T\_People' has fields: 'Id (Int64)', 'Name (String)', 'Firstname (String)', and 'IdAddress (Int64)'. The main editor shows a query in C# Expression language: 

```
from tmp in T_People
where tmp.Id > 2
select tmp
```

 The 'Results' tab is selected, showing a table with 2 items. The table has columns: 'Id', 'Name', 'Firstname', 'IdAddress', and 'T\_Address'. The data rows are: 

Id	Name	Firstname	IdAddress	T_Address
3	siemens	hervé	2	null
4	dan	simth	2	null

On peut voir ce que donne la requête selon la requête LinQ

Results λ SQL IL

http://localhost:1395/WcfOData.svc/T\_People()?\$filter=Id gt 2L

# ODATA



- ➡ Pour requêter sur les opérations de services, il faut passer par cette méthode :

```
this.Execute<T_People> (new Uri ("GetAuthorByFirstName?name='siemens'", UriKind.Relative))
```



Nom de la table



Nom de la fonction



# QUESTIONS ?