Initiation au C# / LINQ / Couche d'accès aux données / ORM

Premier cours sur les bases du langages et du Framework ainsi que des outils disponibles pour réaliser des application .NET

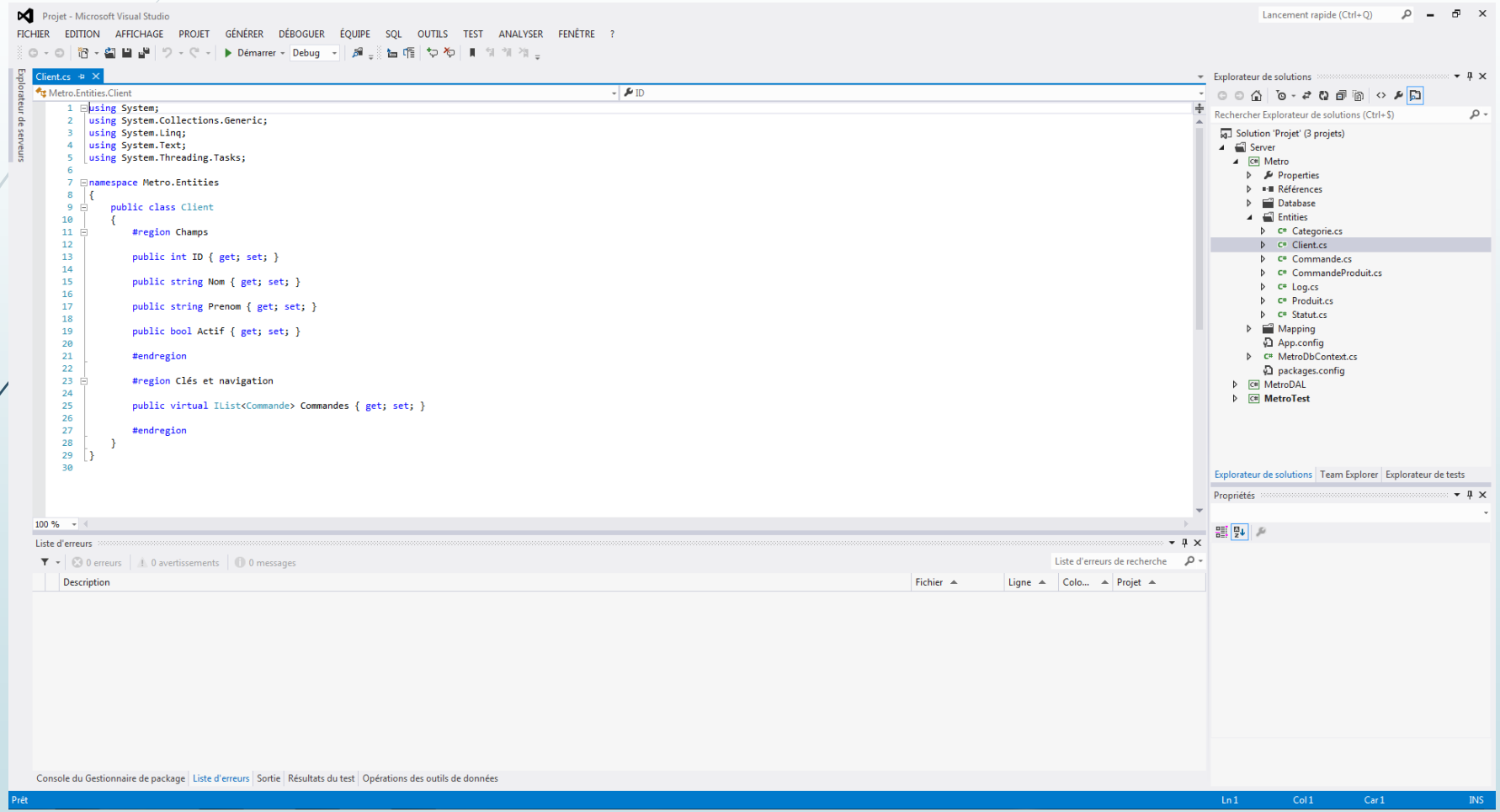
Sommaire

- Introduction sur les rudiments du langage (syntaxe, découverte de l'environnement Visual Studio, bonnes pratiques, etc…).
- Introduction sur l'utilisation des collections, tableaux, énumérations, méthodes d'extensions, exceptions.
- Introduction à LINQ.
- Qu'est ce qu'un ORM? Introduction à Entity Framework.
- Création d'une couche d'accès aux données.
- Tests unitaires.

Syntaxe de base du C#

- Langage orienté objet proche du JAVA.
- Fichiers avec .cs pour extension.
- Le code C# est une suite d'instructions qui se terminent par un point virgule.
- Blocs de code (classes, fonctions, etc…) définies par des accolades.
- Il est possible d'ajouter des commentaires « // », « /* » et « */ ».

Présentation rapide de Visual Studio



Présentation rapide de Visual Studio

De gauche à droite on a :

- Explorateur de serveur : Explorer un serveur SQL, un serveur IIS, Azure, etc.
- Fenêtre dédiée au code (peut être scindée).
- Explorateur de la solution (arborescence).

En bas, on retrouve par défaut les sorties :

- Liste des erreurs, des avertissements.
- Sorties.
- Les recherches.
- Eléments issus du contrôle de code source.

Présentation rapide de Visual Studio

Visual Studio est configurable en fonction du besoin!!

- Régler les tailles des volets.
- Le menu « affichage » permet de disposer de plus d'options.
- Utilisation des points d'arrêt.
- Configuration des points d'arrêts (exemple : s'arrêter au dixième élément dans une boucle).
- Le gestionnaire de package (NuGets).
- Etc.

NB : Dernière version de Visual Studio : 2019

Version minimale à utiliser pour le cours : 2017

Variables

- Les variables contiennent des données et sont utilisées dans des instructions.
- Une variable est typée.
- Exemple de type : Int16, Int32 (ou int), Int64, String, Decimal, Double, Float, etc…

- **NB :** Différence entre String et string? Aucune, string est un alias c# pour le type System.String.

Méthodes

- Permet de factoriser du code afin d'éviter d'avoir à le répéter et pouvoir le réutiliser. (DRY = Don't Repeat Yourself).
- Une méthode peut prendre des paramètres en entrée.
- Une méthode retourne une valeur du type de retour à l'appelant de la méthode. Une méthode qui ne renvoie rien est préfixée du mot-clé **void**.
- Une méthode est accessible en fonction de sa portée :
  - Internal : méthode accessible pour les méthodes internes de l'assembly.
  - Protected
  - Private
  - Public

Bonnes pratiques à respecter

- Le nom des méthodes et des classes doivent commencer par une majuscule.
- Une variable commence par une minuscule et ne contient pas « d'underscore ».
- L'utilisation du mot clé **var** doit être la plus limitée possible (le typage est important en terme de maintenance).
- Une classe par fichier. Eviter de créer plusieurs classes dans le même fichier.
- Eviter les valeurs « codées en dur » (ou « hardcodée »).
- Intercepter les erreurs et gérer les exceptions rigoureusement.

Bonnes pratiques à respecter

- Attention aux erreurs « classiques » :
  - Les classes ajoutées sont par défaut **INTERNAL**. Il faut spécifier la portée **PUBLIC** pour la rendre accessible.
  - Faire attention aux namespaces d'une classe : Si on déplace une classe, le namespace n'est pas mis à jour. Idem si on renomme le dossier qui contient la classe.

Les tableaux

- Permet de stocker plusieurs variables de même type.
- Déclaré en spécifiant son type d'élément.
- Tableau unidimensionnel / Tableau multidimensionnel.
- Indexé à partir de 0.
- Utilisation d'un **foreach** pour parcourir les valeurs.
- Utilisation dans des fonctions, peut être passé en paramètre, etc.

Les tableaux - exemple

Créer et initialiser des tableaux :

```
// Déclaration d'un tableau unidemensionnel d'entier
int[] array1 = new int[5];

// Déclaration d'un tableau unidimensionnel d'entier avec assignation de valeur
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
int[] array3 = { 1, 3, 5, 7, 9 };

// Déclaration d'un tableau multidimensionnel d'entier
int[,] multiDimensionalArray1 = new int[2, 3];

// Déclaration d'un tableau multidimensionnel d'entier avec assignation
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Parcourir des tableaux :

```
// parcourir un tableau unidimensionnel
foreach (int i in array2)
{
    System.Console.Write("{0} ", i);
}
// Sortie: 1 3 5 7 9

// parcourir un tableau multidimensionnel
foreach (int i in multiDimensionalArray2)
{
    System.Console.Write("{0} ", i);
}
// Output: 1 2 3 4 5 6
```

Les collections

- Permet de créer des « groupes d'objets » de même type.
- Le groupe d''objet peut augmenter ou diminuer de façon dynamiquement en fonction des besoins (chose qui n'est pas possible avec les tableaux).
- Différents types de collections :
  - List<T> : Représente une liste d'objets fortement typés accessibles par leur index.
  - Stack<T> : Représente une collection d'objet de type LIFO.
  - Queue<T> : Représente une collection de type FIFO.
  - Dictionnary<Tkey,Tvalue> : Représente une collection de paires clé/valeur organisées en fonction de la clé.

Les collections - exemple

Listes :

```
// Test avec une liste d'entier
List<int> listeEntier = new List<int>();
listeEntier.Add(1);
int valeur = listeEntier.ElementAt(0);
Console.WriteLine("ElementAt : {0}", valeur);
listeEntier.Remove(1);
```

- Accessibilité par index avec la méthode **ElementAt.**
- Ajout par la méthode **Add** (**AddRange** pour ajouter plusieurs éléments d'un coup).
- Suppression par la méthode **Remove.**

Les collections - exemple

Stacks :

```
// test avec une stack d'entier
Stack<int> stackEntier = new Stack<int>();
stackEntier.Push(1); // insère un élément en haut
stackEntier.Push(2);
int jeton = stackEntier.Peek(); // Retourne l'objet situé en haut sans le supprimer
Console.WriteLine("Peek : {0}", jeton);
jeton = stackEntier.Pop(); // Supprime et retourne l'objet en haut
Console.WriteLine("Pop : {0}", jeton);
jeton = stackEntier.Pop();
Console.WriteLine("Pop : {0}", jeton);
```

- Last In First Out.
- La méthode Push insère des éléments en haut.
- La méthode **Peek** et **Pop** retourne les éléments. Attention, la méthode **Peek** ne supprime pas l'élément.

Les collections - exemple

Dictionnaire :

```
// test avec un dictionnaire
Dictionary<int, string> listeCoureur = new Dictionary<int, string>();
listeCoureur.Add(12, "Dupont Jean");
listeCoureur.Add(19, "Gary Rémy");
string nomCoureurDossard12 = string.Empty;
listeCoureur.TryGetValue(12, out nomCoureurDossard12);
Console.WriteLine("Dossard 12 = {0}", nomCoureurDossard12);

// foreach sur dictionnaire
foreach(KeyValuePair<int, string> vp in listeCoureur)
{
    Console.WriteLine("Dossard {0} = {1}", vp.Key, vp.Value);
}
```

- Ensemble de {clé – valeur}
- Utilisation de la classe **KeyValuePair** pour un **foreach**
- Méthode **TryGetValue** pour récupérer une valeur
- **Attention**, on ne peut pas insérer deux fois la même clé dans un dictionnaire!

Les collections - exemple

Queues :

```
// test avec une queue d'entier
Queue<int> queueEntier = new Queue<int>();
queueEntier.Enqueue(1); // Ajoute un objet à la fin
queueEntier.Enqueue(2);
int queueValeur = queueEntier.Dequeue(); // Supprime et retourne l'objet
Console.WriteLine("Dequeue : {0}", queueValeur);
queueValeur = queueEntier.Dequeue();
Console.WriteLine("Pop : {0}", queueValeur);
```

- First In First out.
- La méthode **Enqueue** permet d'ajouter des éléments.
- La méthode **Dequeue** permet de récupérer les éléments (et les supprime au passage).

Les énumérations

- Le mot clé **enum** est utilisé pour déclarer une énumération.
- Une énumération est un ensemble de constantes nommées.
- Par défaut, le premier énumérateur a la valeur 0, chaque énumérateur suivant à la valeur n+1.
- A utiliser pour éviter des valeurs « hardcodées ».
- Peut être utilisé dans des conditions ou comme paramètre d'une fonction.

- IMPORTANT :
  - Il est impossible de créer une énumération de String.
  - Ne pas abuser des énumérations.

Les énumérations - exemple

Déclarer une énumération d'entier

```
0 références
public enum Droit : int
{
    NON = 0,
    LECTURE_SEULEMENT,
    LECTURE_ECRITURE,
    ALL
}
```

Exemple d'utilisation

```
Droit monDroit = Droit.ALL;
switch(monDroit) // Switch ou If
{
    case Droit.ALL:
        System.Console.Write("Droit maximal : {0} ", monDroit.ToString());
        break;
    default:
        System.Console.Write("Vous n'avez pas tous les droits");
        break;
}
// Sortie : Droit maximal : ALL
```

Exemple de « cast »

```
int codeMonDroit = (int)monDroit;
System.Console.Write("Code de mon droit : {0} ", codeMonDroit);
// Sortie : Code de mon droit : 3
```

Les méthodes d'extensions

- Les méthodes d'extensions permettent d'ajouter des méthodes à des types existants sans créer de type dérivé (ou hérité).
- Une méthode d'extension est statique (déclaré comme **static**).
- Une méthode d'extension doit forcément contenir un paramètre déclaré avec **this**. Le type de ce paramètre sera la type « étendu ».

- Exemples d'utilisations :
  - LINQ est composée de méthodes d'extensions.
  - Ajouter des méthodes à la classes String pour tronquer une chaine.
  - Etc.

Les méthodes d'extensions - exemple

Exemple d'extension : Compter les mots dans une chaine de caractère :

```
0 références
public static class MyExtension
{
    1 référence
    public static int CompterLesMots(this String str)
    {
        return str.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Tester la méthode :

```
string s = "J'adore le C#";
int mots = s.CompterLesMots();
System.Console.Write("Nombre de mots : {0} ", mots);
// Sortie : Nombre de mots : 3
```

C'est à vous !

- Ecrire une méthode d'extension pour réaliser la fonction miroir sur une chaine de caractère (exemple : « bonjour » devient « ruojnob »).

Le mécanisme d'exception

- Permet de gérer les situations inattendues ou exceptionnelles lors de l'exécution d'un programme.
- S'articule autour de trois mots clés : **try**, **catch** et **finally.**
- Permet d'éviter une interruption du programme.

```
try
{
    // code de traitement, opérations, fonctions, etc.
}
catch(Exception ex)
{
    // traitement de l'erreur contenu dans ex
}
finally
{
    // code exécuté dans tous les cas
}
```

Le mécanisme d'exception

- Les exceptions sont des types qui dérivent en définitive tous de **System.Exception.**
- Toute instruction susceptible de lever une exception doit être incluse dans un bloc **try.**
- Une exception est de nouveau « levable » à l'aide du mot **throw.**
- Le bloc **finally** permet d'exécuter du code mais si le bloc catch a intercepté une erreur.
- On peut créer nos propres expressions qui dérivent du type **System.Exception.**

String Interpolation

- Nouvelle syntaxe apparue depuis la version de C#6
- Permet de concaténer des chaines de caractères
  - Identique à un string.Format, string.Concat, etc

```
// deux variables (cela fonctionnerait aussi avec des objets)
string prenom = "Adrien";
string nom = "CHARGUERAUD";

string resultat = $"Je m'appelle {prenom} {nom}"; // string interpolation
string resultat2 = string.Format("Je m'appelle {0} {1}", prenom, nom); // string.Format
string resultat3 = "Je m'appelle " + prenom + " " + nom; // PAS BIEN
```

- Attention aux concaténations avec le symbole « + »
- Pensez au **StringBuilder** pour des volumes importants

Propagation du null (null conditionnal operator)

- Introduction d'un nouvel opérateur **?.**
- Accéder aux membres d'un sans lever une exception de type **NullReferenceException** dans le cas où l'objet est null.

```
// Soit l'objet MonObjet avec la propriété Code
MonObjet monObjet = new MonObjet { };
string code = monObjet?.Code; // marche dans le cas ou MonObjet est null
```

Introduction à LINQ

- LINQ = Language-Integrated Query.
- Langage basé sur l'utilisation de méthode d'extension.
- Outil permettant de créer des requêtes afin d'interroger des sources de données, on exécute une requête sur une source de donnée.
- Toutes les opérations LINQ comportent 3 actions distinctes :
  - Obtenir la source de données.
  - Créer la requête .
  - Exécuter la requête.

LINQ – Exemple de source – requête – exécution

Exemple utilisation de LINQ :

```
// 1) déclarer la source de donnée
int[] notes = new int[] { 4, 12, 18, 7, 11, 3, 15 };

// 2) Définir la requête
IEnumerable<int> notesQuery =
    from note in notes
    where note > 10
    select note;

// 3) Exécuter la requête
foreach(int i in notesQuery)
{
    Console.Write(i + " ");
}
```

- La source peut être une liste (type **List<T>**) et plus généralement les types qui prennent en charge **IEnumerable<T>.**
- La source peut aussi être un tableau car il prend en charge l'interface **IEnumerable<T>.**
- La requête sera présentée plus tard.
- L'exécution est « différée ».

LINQ – syntaxe des requêtes

- Il est possible d'écrire une requête LINQ de deux manières différentes :
  - La syntaxe de requête (Query Syntax).
  - La syntaxe de méthode (Method Syntax).
- Le résultat des deux syntaxes est identique.

Exemple de syntaxe avec LINQ :

```
int[] notes = new int[] { 4, 12, 18, 7, 11, 3, 15 };

// syntaxe de requête pour récupérer les notes au dessus de 10 triée dans l'ordre croissant
IEnumerable<int> notesQuery1 =
    from note in notes
    where note > 10
    orderby note
    select note;

// syntaxe de méthode pour récupérer les notes au dessus de 10 triée dans l'ordre croissant
IEnumerable<int> notesQuery2 = notes.Where(note => note > 10).OrderBy(note => note);
```

LINQ – syntaxe des requêtes

- La syntaxe de requête est plus simple et plus facile à lire.
- La syntaxe de méthode est la plus utilisée.
- La documentation de l'espace de nom **System.Linq** utilise la syntaxe de méthode.

- La suite du cours portera sur la **syntaxe de méthode**!

LINQ – la syntaxe de méthode

- La syntaxe de méthode s'appuie sur des méthodes d'extensions LINQ :
  - Where()
  - First(), FirstOrDefault(), Last(), LastOrDefault(), Single(), SingleOrDefault()
  - Sum(), Average()
  - Max(), Min(), Count()
  - Any(), Contains()
  - Select(), SelectMany()
  - Include()
  - OrderBy(), OrderByDescending(), ThenBy(), ThenByDescending(), GroupBy()
  - Etc.

- La syntaxe de méthode utilise principalement les **expressions lambda.**

LINQ – expressions lambdas

- Les expressions lambda sont aujourd'hui utilisée massivement dans le Framework .NET.
- LINQ utilise les expressions lambda.
- Plus facile d'utilisation (voir exemple) que de déclarer des délégués (mot clé **delegate**).
- Permet de faire abstraction du type.

LINQ – expressions lambdas

Exemple avec et sans expressions lambda :

```
// Déclaration de mon délégué
1 référence
static bool PlusGrandQue10(int arg)
{
    return (arg > 10);
}

1 référence
public static void TestExpressionsLambda()
{
    List<int> list = new List<int> { 4, 12, 18, 7, 11, 3, 15 };

    // sans expressions lambda et avec la déclaration de la fonction PlusGrandQue10
    Func<int, bool> maFonctionDelegue = PlusGrandQue10;
    IEnumerable<int> resultat = list.Where(maFonctionDelegue);
    foreach (var item in resultat)
    {
        Console.WriteLine("{0} ", item);
    }

    // avec expression lambda
    IEnumerable<int> resultat2= list.Where(n => n > 10);
    foreach (var item in resultat2)
    {
        Console.WriteLine("{0} ", item);
    }
}
```

LINQ – quelques exemples

```
List<int> liste = new List<int> { 4, 12, 18, 7, 11, 3, 15, 4, 0, 19, 17, 12, 12, 9 };

// compter le nombre de note au dessus de 10
int nbNotePlus10 = liste.Where(n => n > 10).Count();
Console.WriteLine("Nombre de note au dessus de 10 : {0} ", nbNotePlus10);

// calculer la moyenne
double moyenne = liste.Average(n => n);
Console.WriteLine("Moyenne : {0} ", moyenne);

//  trier les note par ordre croissant et prendre la plus faible et la plus grande
int plusPetiteNote = liste.OrderBy(n => n).First();
int plusPetiteNoteV2 = liste.Min(n => n);
int plusGrandeNote = liste.OrderBy(n => n).Last();
int plusGrandeNoteV2 = liste.Max(n => n);
Console.WriteLine("Plus basse : {0} - Plus haute : {1} ", plusPetiteNote, plusGrandeNote);

// déterminer si quelqu'un a eu 0
bool laBulle = liste.Any(n => n == 0);
Console.WriteLine("Des notes égales à 0 ? {0}", laBulle);
```

```
Nombre de note au dessus de 10 : 8
Moyenne : 10.2142857142857
Plus basse : 0 - Plus haute : 19
Des notes égales à 0 ? True
Appuyez sur une touche pour continuer...
```

LINQ – pour aller plus loin

Il existe de nombreuses utilisations possibles de LINQ :

- LINQ to XML permet de travailler avec des fichiers XML comme source de données.
- LINQ to SQL permet de travailler avec des bases de données relationnelles comme source de données.
- LINQ s'utilise aussi bien en asynchrone qu'en synchrone.

Object-relational mapping

- L'ORM introduit des notions de relations entre objet :
  - Relation One-To-One.
  - Relation One-To-Many.
  - Relation Many-To-Many.
- L'ORM permet d'accéder à la base de données, de l'interroger, de la modifier, etc.

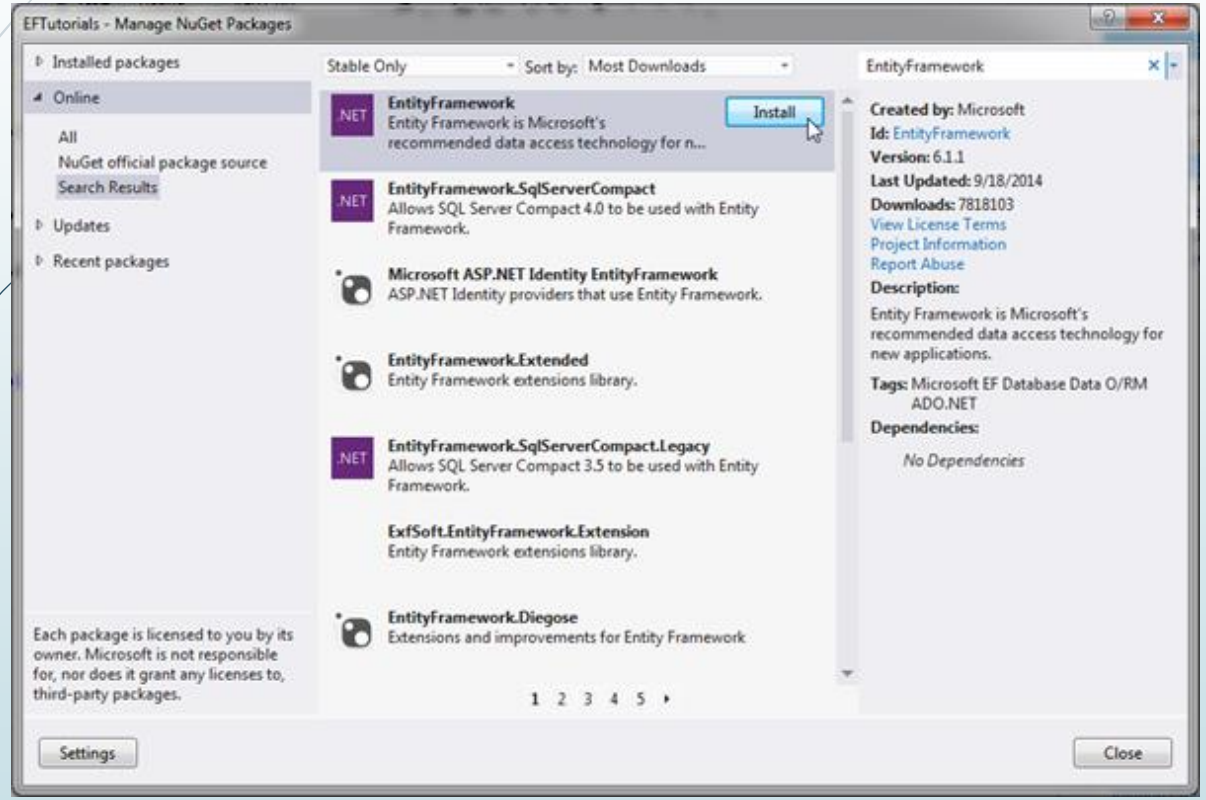
Découverte Entity Framework

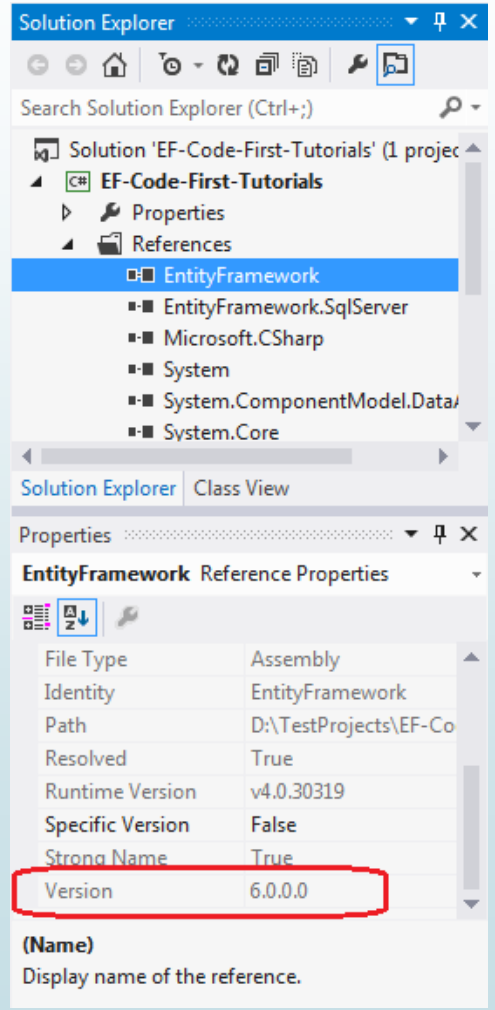
- Entity Framework est un ORM propulsé par Microsoft (acronyme EF)
- EF est actuellement dans sa version 6.4.x et offre l'avantage de :
  - Fournir aux développeurs un mappeur objet relationnel qui permet d'utiliser des données relationnelles à l'aide d'objet .Net.
  - Permettre de faire « quasi » abstraction du code que l'on doit généralement écrire quand on souhaite accéder à une source de donnée.
- EF n'est pas que compatible pour SQL Server, il peut fonctionner sur Oracle, PostGre ou encore MySQL.

Entity Framework - installation

- EF est totalement gratuit.
- Disponible dans les packages NUGET donc facile d'installation.
- Souvent mis à jour par la communauté.
- Recommandé par Microsoft.
- S'intègre dans n'importe quel type de projet (client léger, client lourd, application console, service, etc.).

Entity Framework - installation





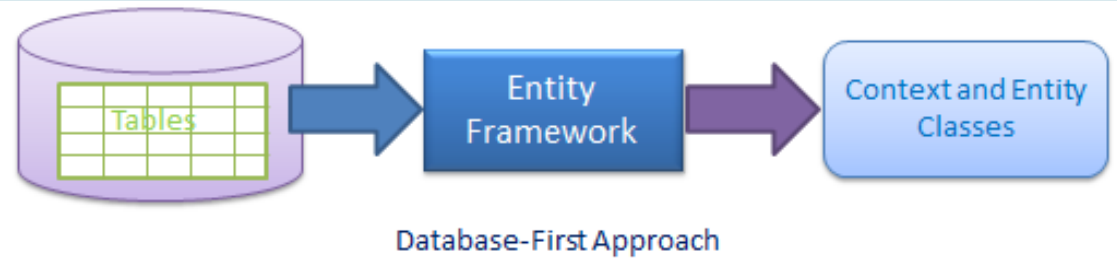
Entity Framework – approche

- Il existe trois approches différente pour intégrer Entity Framework dans un projet .NET :
  - Database First : Une DB existe, on souhaite l'utiliser pour générer un modèle de données (vue designer)
  - Model First : Aucune DB n'existe, on souhaite en générer une depuis un modèle de données (vue designer)
  - Code First : Aucune DB, on souhaite en générer une depuis du code C# (pas de vue designer)

- Aujourd'hui, Microsoft recommande l'approche Code First

Entity Framework – Database First

- Approche consistant à utiliser une base existante pour générer un modèle de données.
- Utilisation de la vue Designer.
- Création / génération d'un fichier au format EDMX.
- Possibilité de mettre à jour le modèle rapidement et facilement.



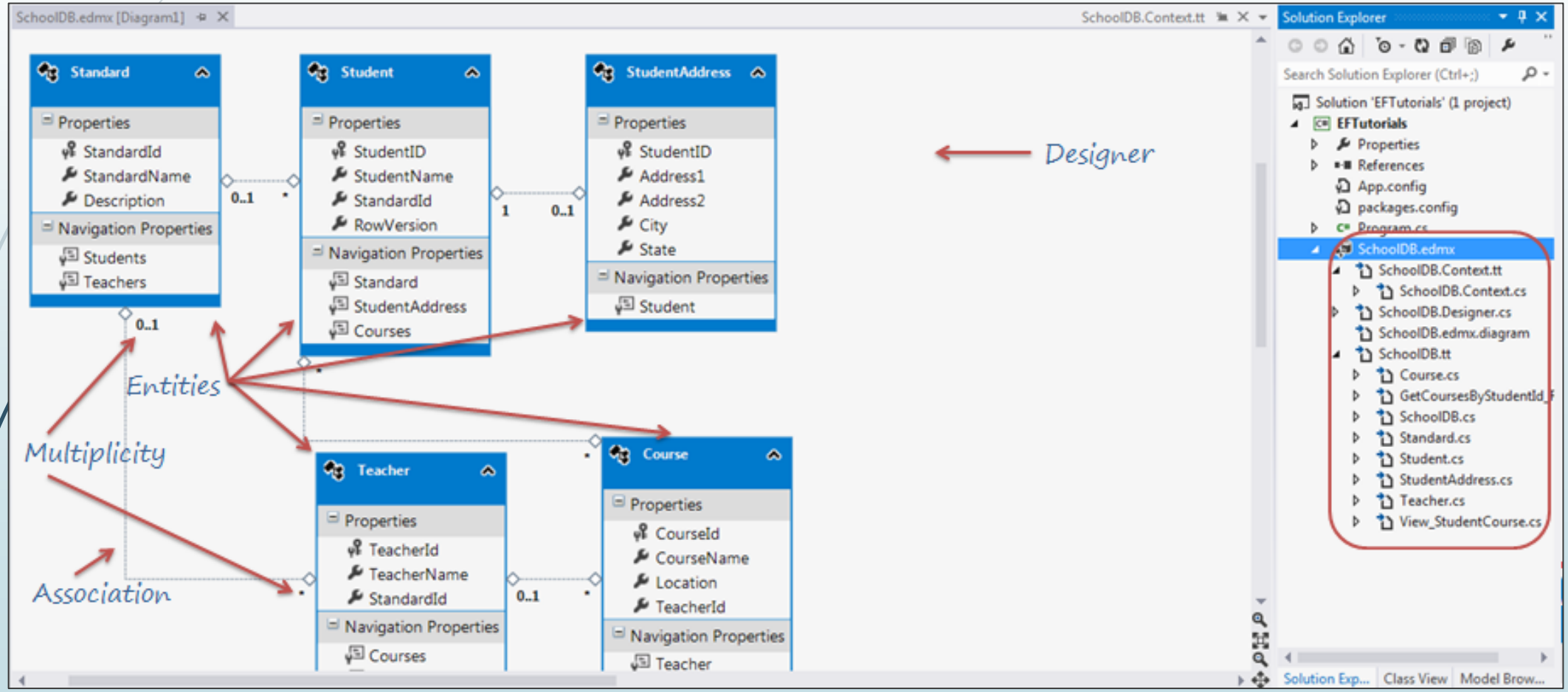
Entity Framework – Database First

- Avantages :
  - Simple d'utilisation.
  - Gain de temps dans le développement du modèle de données.
  - Très populaire dans certaines sociétés ou la base est développée séparément de l'application.

- Inconvénients :
  - Plus difficile à customiser.
  - Beaucoup de code généré à surcharger pour de la customisation approfondie.
  - Manque de « contrôle ».

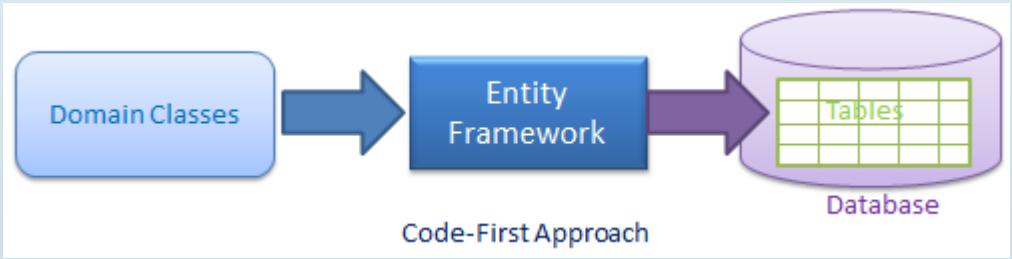
Entity Framework – Database First

Exemple d'un projet Database First :



Entity Framework – Code First

- Approche consistant à développer son propre modèle de données afin de générer la base de données correspondante.
- Pas nécessaire d'utiliser la vue Designer.
- Contrôle total du modèle de données.
- Génération automatique de la base de données.



Entity Framework – Code First

- Avantages :
  - Contrôle du modèle.
  - Possibilité de surcharger les classes facilement.
  - Pas de code généré, on génère la base.
  - Très populaire dans le monde du développement, pas de designer, code C# classique.
  - Très populaire dans certaines sociétés ou la base est développée séparément de l'application.

- Inconvénients :
  - Plus difficile et plus long à mettre en place.
  - Les changements « manuels » en base doivent être répercuté manuellement sur votre code.
  - Plus « verbeux ».

Entity Framework – Code First

- Afin de « lier » son propre modèle à la base de données (qui sera créée si elle n'existe pas), il existe deux méthodes:
  - Utiliser l'API Fluent.
  - Utiliser les DataAnnotations.

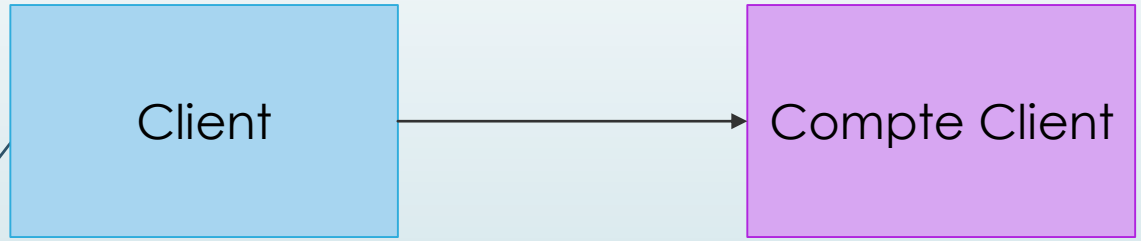
- Attention, c'est soit une méthode soit l'autre, il est vivement déconseillé de faire un « mix » des deux.

Code First - DataAnnotations

- EF fournit un certain nombre d'annotation pour arriver à créer un modèle de données en C# et ensuite pouvoir le retranscrire en base de données.
- On applique des annotations sur les propriétés
  - Key : Marque la propriété comme clé primaire de la table
  - Timestamp : Marque la propriété comme une colonne de type timestamp
  - Required : Marque la propriété comme obligatoire (doit contenir une donnée)
  - MinLength et MaxLength
  - StringLength
  - Column : Spécifie le nom de la colonne et le type de donnée qui sera mappée avec la propriété
  - Table : Spécifie le nom de la table qui sera mappée avec la classe
  - Index : Permet la création d'un index pour une colonne spécifiée
  - ForeignKey : Permet de définir une clé étrangère

Code First – DataAnnotations Exemple

- Exemple de classe utilisable avec du Code First



- Un client est une entité composé d'un identifiant auto-généré et d'un nom
- Un compte client est une entité composée d'un identifiant et d'un numéro
- Un client peut avoir plusieurs comptes

Code First – DataAnnotations Exemple

Classe Client.cs:

```
[Table("APP_Client")]
4 références
public class Client
{
    [Key]
    [Column("CLI_ID")]
    1 référence
    public int Id { get; set; }

    [StringLength(50)]
    [Required]
    [Column("CLI_NOM")]
    1 référence
    public string Nom { get; set; }

    2 références
    public ICollection<CompteClient> Comptes { get; set; }
}
```

Classe CompteClient.cs:

```
[Table("APP_CompteClient")]
3 références
public class CompteClient
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    0 références
    public int Id { get; set; }

    [Required]
    [MaxLength(10)]
    1 référence
    public string NumeroCompte { get; set; }

    0 références
    public int ClientId { get; set; }

    [ForeignKey("ClientId")]
    0 références
    public Client Client { get; set; }

    [NotMapped]
    0 références
    public string NomBanque { get; set; }
}
```

- Relation **one-to-many.**
- L'annotation **DatabaseGenerated** permet de spécifier si l'id sera auto-incrément ou non (par défaut il l'est).
- **NotMapped** ne créera pas le champ en base.

Code First – API Fluent

- L'API Fluent est plus complexe à mettre en place que les DataAnnotations.
- Elle offre cependant plus de fonctionnalité en terme de configuration.
- Consiste à « Surcharger » la méthode **OnModelCreating** du contexte d'EF (override) pour ajouter les informations de « mapping ».
- Une information de « mapping » est classe héritant de la classe EF **EntityTypeConfiguration<Tentity>.**

Code First – API Fluent Exemple

Classe Client.cs et ClientFluent.cs :

```
5 références
public class Client
{
    [Key]
    3 références
    public int Id { get; set; }

    [StringLength(50)]
    [Required]
    2 références
    public string Nom { get; set; }

    4 références
    public ICollection<CompteClient> Comptes { get; set; }
}

2 références
public class ClientFluent : EntityTypeConfiguration<Client>
{
    1 référence
    public ClientFluent()
    {
        ToTable("APP_Client");
        HasKey(c => c.Id);

        Property(c => c.Id).HasColumnName("CLI_ID").IsRequired().HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);
        Property(c => c.Nom).HasColumnName("CLI_NOM").IsRequired().HasMaxLength(50);

        HasMany(c => c.Comptes).WithRequired(cc => cc.Client).HasForeignKey(cc => cc.ClientId);
    }
}
```

- Une classe pour définir l'entité Client.
- Une classe pour définir le « mapping » avec l'API Fluent.
- L'API Fluent est prioritaire par rapport aux annotations.

Code First – API Fluent Exemple

Classe CompteClient.cs et CompteClientFluent.cs :

```
4 références
public class CompteClient
{
    [Key]
    2 références
    public int Id { get; set; }

    [Required]
    [MaxLength(10)]
    2 références
    public string NumeroCompte { get; set; }

    2 références
    public int ClientId { get; set; }

    2 références
    public Client Client { get; set; }

    1 référence
    public string NomBanque { get; set; }
}

2 références
public class CompteClientFluent : EntityTypeConfiguration<CompteClient>
{
    1 référence
    public CompteClientFluent()
    {
        ToTable("APP_CompteClient");
        HasKey(cc => cc.Id);

        Property(cc => cc.Id).HasColumnName("Id").IsRequired().HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
        Property(cc => cc.NumeroCompte).HasColumnName("NumeroCompte").IsRequired().HasMaxLength(10);
        Ignore(cc => cc.NomBanque);

        HasRequired(cc => cc.Client).WithMany(c => c.Comptes).HasForeignKey(c => c.ClientId);
    }
}
```

- Idem que dans la diapositive précédente.
- Relation **one-to-many** entre ces deux entités déclarées avec l'API Fluent (disparition de l'annotation **ForeignKey**).

Code First – API Fluent Exemple

- Attention, la création des **EntityTypeConfiguration** n'est pas suffisante, il faut ajouter ces classes dans le contexte !
- La notion de contexte est expliquée juste après.
- Surcharge de la méthode **OnModelCreating** .

Méthode OnModelCreating du contexte :

```
1 référence
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.HasDefaultSchema("dbo");
    modelBuilder.Configurations.Add(new ClientFluent());
    modelBuilder.Configurations.Add(new CompteClientFluent());
}
```

Entity Framework – Le contexte

- Le contexte est nécessaire pour « instancier » et utiliser EntityFramework.
- Il faut créer une classe héritant de la classe **DbContext** du Framework.
- Permet de créer / se connecter à la base et de récupérer / insérer / mettre à jour des éléments.

```
3 références
public class ContextDA : DbContext
{
    1 référence
    public ContextDA() : base("name=Cours1ConnexionString")
    {

    }

    1 référence
    public DbSet<Client> Clients { get; set; }
    0 références
    public DbSet<CompteClient> CompteClients { get; set; }
}
```

- Surcharge de **DbContext.**
- Présence du **constructeur** par défaut avec le nom de la chaine de connexion à utiliser.
- Présence de **DbSet** (ou **IDbSet**) pour se connecter aux tables (une par entité).

Entity Framework – La configuration

- EF a besoin d'un fichier app.config pour fonctionner.
- Ce fichier app.config permet de déterminer la chaine de connexion et quelques paramètres de bases d'EF.
- Fichier XML.

Exemple de app.config :

```
<configuration>
  <configSections>...</configSections>
  <startup>...</startup>
  <entityFramework>...</entityFramework>
  <connectionStrings>
    <add name="Cours1ConnexionString"
    connectionString='Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename="D:\a.chargueraud\Mes documents\Developpement\Support\Sol
    providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

Entity Framework – Exemple complet

- Un exemple complet c'est :
  - Un contexte avec une configuration.
  - Un modèle mappé avec Fluent ou avec les DataAnnotation.
  - Des manipulations diverses des collections définies dans le contexte et correspondant à ce qu'il y a en base.
  - D'éventuels ajouts / suppressions ou modifications de données.

- Voir solution Visual Studio fournie en annexe.

Entity Framework – pour aller encore plus loin

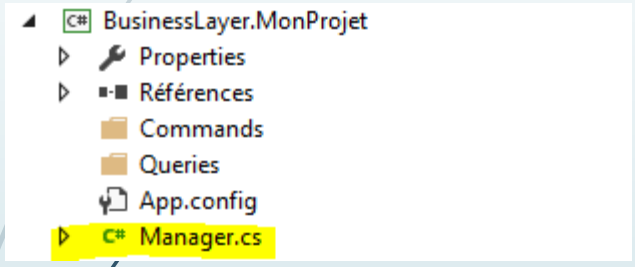
- Excellente documentation sur EF : http://www.entityframeworktutorial.net/
  - Notion de requête synchrone, sauvegarde asynchrone.
  - Intercepteur.
  - Lazy / eager / explicit loading.
  - Optimisations possibles.
  - Etc.

Business Logic Layer

- Le but de cette couche est d'éviter l'utilisation du contexte Entity Framework directement dans des contrôleurs, des view models, etc.
- Permet de factoriser du code.
- Couche d'accès aux données plus facilement réutilisable et ce peu importe le type d'application (client léger ou client lourd).
- Gain de temps.
- Existence de nombreux patrons de conceptions pour pallier à cette couche.

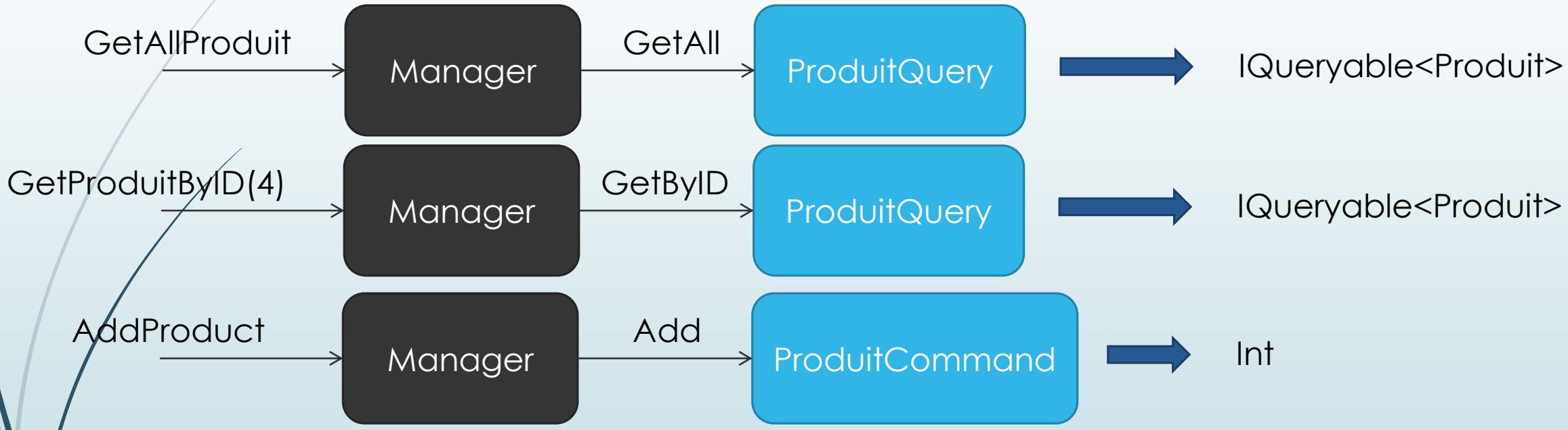
Business Logic Layer : Exemple

- Exemple d'une Business Logic Layer « simplifiée » en utilisant les Queries / Commands.



- Le **Manager** contient le constructeur qui va instancier le contexte Entity Framework.
- Le **Manager** contient toutes les méthodes pour interagir avec Entity Framework.
- Le répertoire **Queries** contient les classes pour récupérer les entités.
- Le répertoire **Commands** contient les classes pour mettre à jour, ajouter ou supprimer une entité.

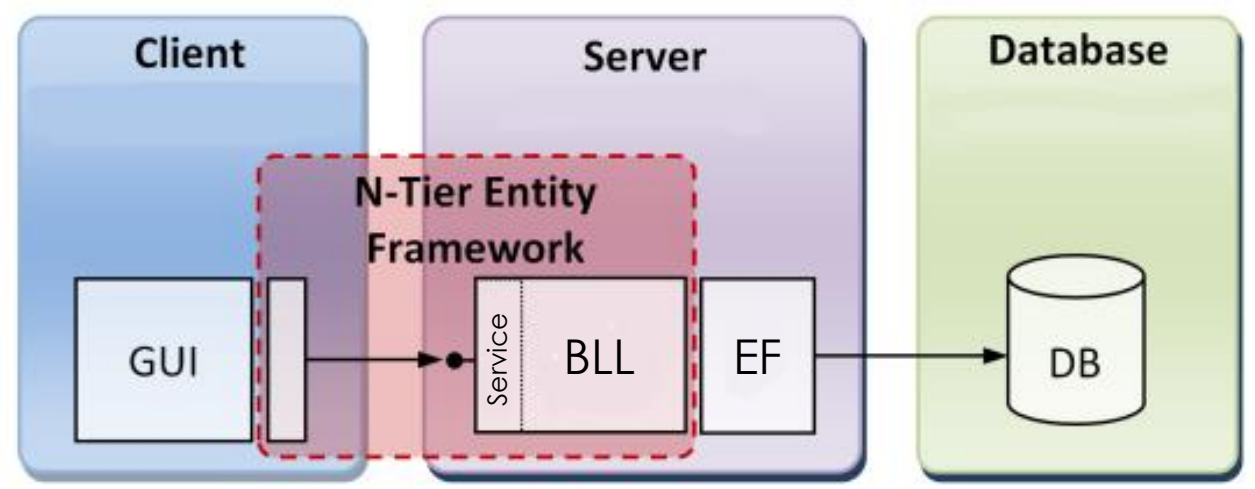
Business Logic Layer : Exemple



Business Logic Layer : Conclusion

- On distingue bien les **Commands** et les **Queries**.
- La partie applicative utilise cette couche pour accéder ou modifier les données : Ne pas ajouter de couplage inutile.
- On peut réutiliser ce manager en WPF dans un ViewModel, en ASP.NET dans un contrôleur, etc.
- Ce n'est pas la seule solution possible :
  - **Microservices**
  - Utilisation d'un service avec une simple interface.
  - D'autres patrons de conceptions (fabrique par exemple).
  - Etc.

Récapitulatif



- **Client** : ASP.NET MVC, WPF, etc.
- **Server** : IIS
- **Database** : SQL Server, Oracle, etc.

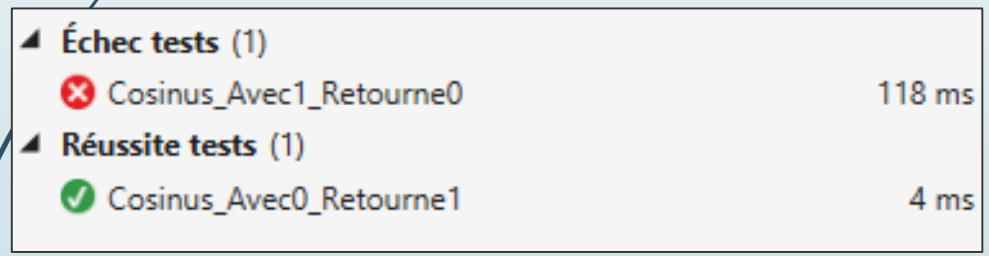
Test unitaire

- Les tests unitaires permettent de vérifier le bon fonctionnement d'une portion de code définie.
- Permet de valider le bon fonctionnement en testant un cas dont on connait le résultat à l'avance.
- Utilisation de « Mock » si nécessaire.

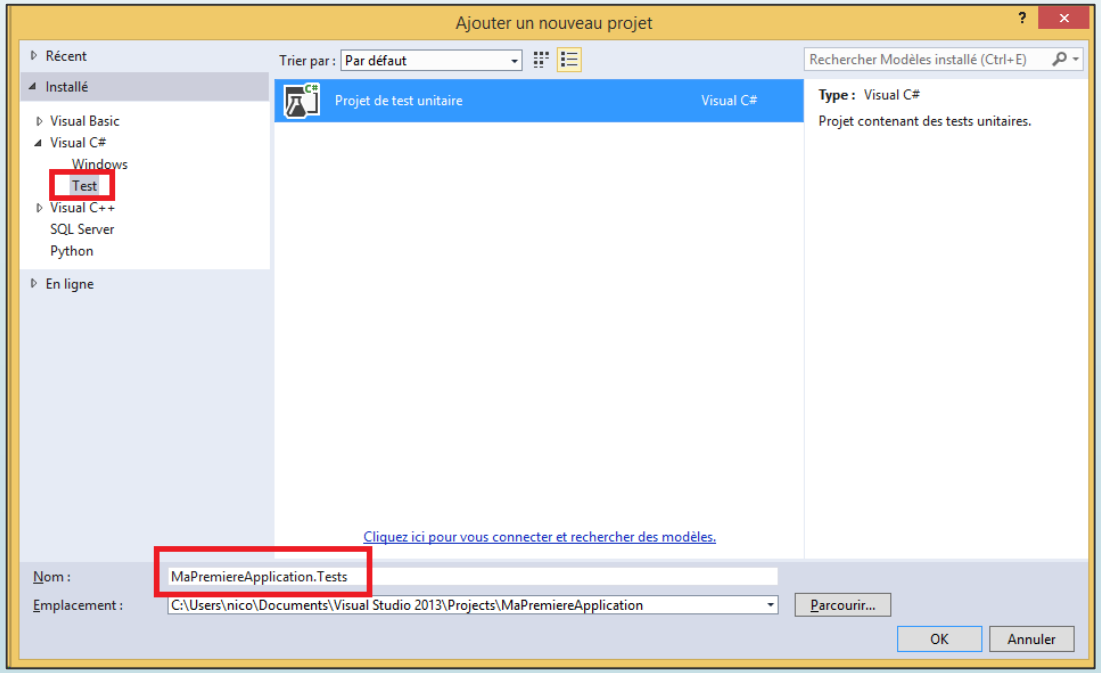
Test unitaire – intégration dans VS

- Nécessite la création d'un projet de test sous Visual Studio.
- Fourni un environnement structuré permettant l'exécution et le développement de test.

Exemple de résumé



Créer un projet



Test unitaire – classe et méthodes

- Une classe de test est définie par l'annotation **[TestClass].**
- Une méthode de test est définie par l'annotation **[TestMethod].**
- Utilisation du mot clé **Assert** pour définir des assertions.
- Pensez à inclure les **using** pour bénéficier des annotations.
- Exécuter et déboguer les tests.

- ATTENTION :
  - Si les annotations ne sont pas présentes, aucun test ne sera exécuté.

Test unitaire - assertions

- Un test défini au minimum un **Assert**. La liste des assertions (non exhaustives) est la suivante :
  - IsTrue(Boolean) : Vérifie que la condition spécifiée est **true.**
  - IsNotNull(Object) : Vérifie que l'objet spécifié n'est pas **null.**
  - IsFalse(Boolean) : Vérifie que la condition spécifiée est **false.**
  - IsInstanceOf(Object, Type) : Vérifie que **l'objet** spécifié est une instance du **type** spécifié.
  - Equals : Détermine si deux **objets** sont égaux.
  - Etc.

Test unitaire - exemple

```
[TestClass]
0 références
public class MaClasseDeTest
{
    [TestMethod]
    | 0 références
    public void Cosinus_Avec0_Retourne1()
    {
        double cosinus0 = Math.Cos(0);
        Assert.IsTrue(cosinus0 == 1);
    }

    [TestMethod]
    | 0 références
    public void Cosinus_Avec1_Retourne0()
    {
        double cosinus1 = Math.Cos(45);
        Assert.IsTrue(cosinus1 == 0);
    }
}
```

- Présence des **Assert** que le résultat attendu et le résultat obtenu.
- Présence des annotations.
- Nommage des méthodes de test.

C'est à vous !

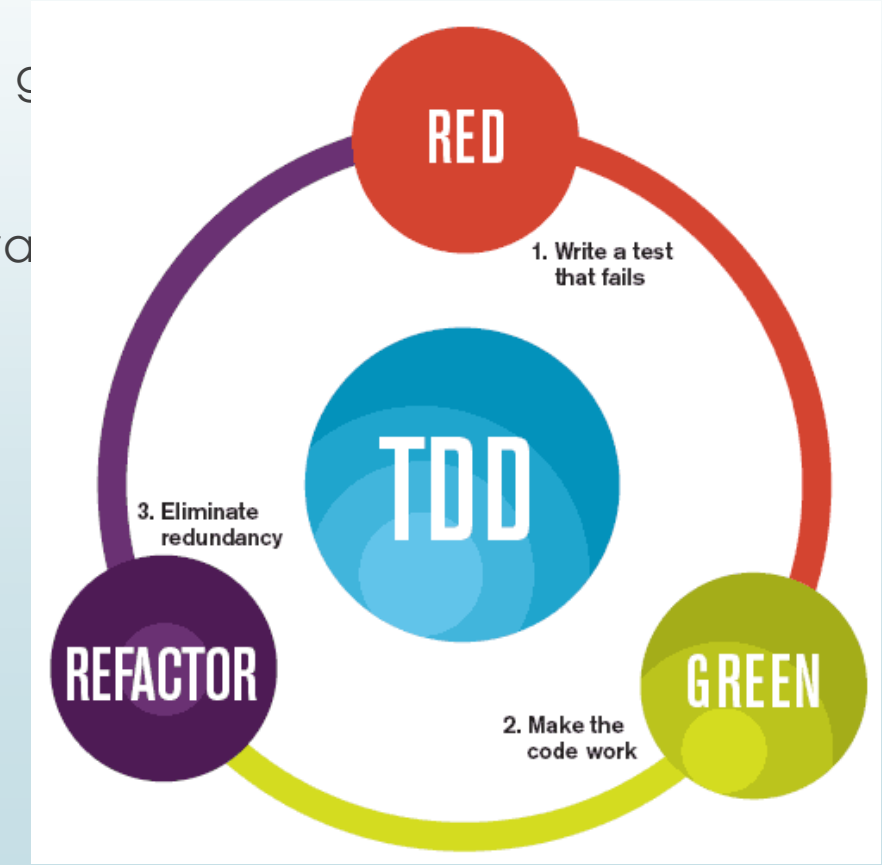
- Ecrire une classe de test pour tester la fonction factorielle (Math.Factorielle) qui avec 3 doit retourner 6.

Le pattern AAA

- Pattern permettant de structurer et d'uniformiser ses tests.
- On décompose un test en trois parties :
  - Arranger (Arrange) : Définir les objets, les variables nécessaires au bon fonctionnement de son test. On peut aussi introduire un mock à ce niveau-là dans certains cas.
  - Agir (Act) : On exécute l'action que l'on souhaite tester (en général, on appelle une méthode que l'on veut tester).
  - Auditer (Assert) : On vérifie le résultat obtenu par rapport aux données définies dans la première étape.

TDD : Test Driven Development

- Développement piloté par les tests.
- Permet de s'assurer de la solidité et de la g
- Respecte la méthodologie de travail suiva
  - Ecrire un test et vérifier qu'il échoue.
  - Ecrire le code minimal pour valider le test
  - Refactoriser le code si possible



Test unitaire – pour aller plus loin

- Il existe de nombreuses annotations supplémentaires à utiliser dans des classes de test :
  - TestInitialize : Identifie la méthode à exécuter avant le test pour configurer les ressources requises par tous les tests dans la classe de test.
  - TestCleanup : Identifie la méthode qui contient le code à utiliser une fois tous les tests exécutés pour libérer les ressources.

- Page MSDN de tout ce qui existe : TestTools.UnitTesting.