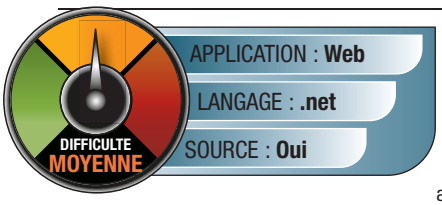
Entity Framework 4.1 : Microsoft à la conquête du marché des ORM !

Suite à la première introduction d'Entity Framework (EF) dans .NET 3.5 SP1, les développeurs ont fait de nombreux retours afin de compléter et d'améliorer cette première version. Microsoft a entendu leurs desiderata et a apporté plusieurs améliorations avec EF 4.0. Pourtant les puristes du code n'étaient pas totalement satisfaits, en effet cette version nécessitait encore l'utilisation de modèles de design pour la création et la gestion des entités.



Avec la nouvelle version EF 4.1 on peut non seulement avoir une approche Database First et Model First mais aussi et surtout une approche Code First. Cette extension va faire d'Entity Framework un Data Layer complet et très performant.

APPROCHE DATABASE FIRST ET MODEL FIRST

L'approche Database First se fonde sur une base de données déjà existante dans laquelle on utilise des outils (comme EF Designer de Visual Studio) pour générer les classes C# ou VB. Les classes générées ainsi que les relations entre classes et base de données peuvent être modifiées par le Designer d'EF ou, pour les plus téméraires, via les fichiers XML … La priorité de cette approche est d'abord à la base de données, le code et le modèle viennent en second. L'approche Model First, quant à elle, consiste en un commencement « from scratch ». On débute par le modèle des entités, là encore via EF Designer. Ce modèle peut être utilisé pour générer la base de données ainsi que les classes C# ou VB.
La priorité de cette approche est donc donnée au modèle, le code et la base de données viennent ensuite.

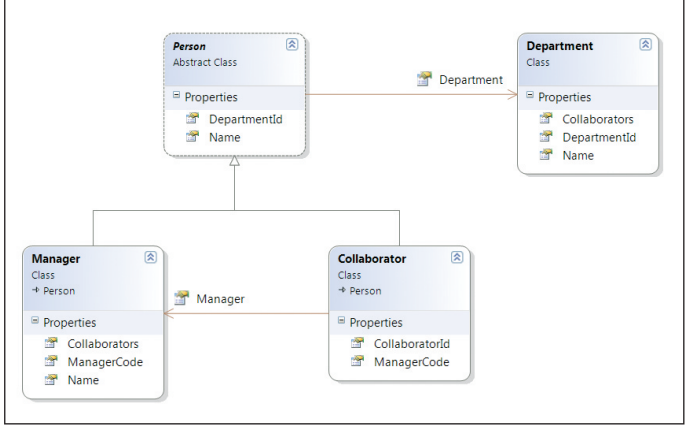
APPROCHE CODE FIRST

En plus de supporter le développement basé sur le modèle et le design, EF 4.1 offre une approche centrée en priorité sur le code. On appelle cela Code First : avec cette approche, on démarre par le code, il n'y a ni schéma, ni XML décrivant le modèle. On crée simplement les classes du domaine de l'application, Code First permettant leur utilisation avec EF. En utilisant un contexte, on peut écrire et exécuter des requêtes LINQ to Entities et tirer avantage du suivi de modification (Change Tracking) proposé par EF. Tout est géré de manière transparente, vous allez même oublier qu'EF est là !
EF 4.1 gère toutes les interactions avec la base de données pour vous. En arrière-plan, EF crée lors de l'exécution toutes les classes permettant de lier les objets avec la base de données : dans ce cas, le contexte n'est pas créé depuis les fichiers de configuration XML, mais calculé et généré en fonction de la configuration de la classe DbContext. Code First vous permet donc de :

- développer sans jamais avoir besoin d'utiliser le Designer ou définir des fichiers XML de mapping
- définir les objets du modèle selon la méthode «Plain Old CLR Objects » sans nécessiter des classes de base

Pour pouvoir utiliser ces nouvelles fonctionnalités, il faut installer la mise à jour d'EF 4.1. La Release Candidate est sorti mi-mars 2011, la version finale devrait déjà être disponible à la parution de cet article (en Release To Web et package NuGet). Cette mise à jour fonctionne bien évidemment avec VS 2010 et avec tout projet .NET 4 incluant ASP.NET et ASP.NET MVC. EF Code First permet d'utiliser des objets POCO (Plain Old CLR Objects) pour représenter les entités dans la base de données. Cela signifie qu'il n'est pas nécessaire de dériver les classes d'une classe de base spécifique à EF, ni d'implémenter des interfaces spécifiques ou des attributs de la persistance de données. Ceci permet aux classes de rester propres, facilement testables et réutilisables. Prenons le cas d'implémentation d'un domaine d'entreprise avec les entités *Managers*, *Collaborators* et *Departments*. [Fig.1]. Il suffit de créer les classes, comme vous le feriez de manière classique avec du code standard en utilisant les classes de base et les interfaces génériques. Par exemple, les classes Person et Collaborator ci-dessous sont définies comme des classes POCO standard, rien de spécifique n'a été rajouté. Elles peuvent être utilisées telles quelles avec EF 4.1 et Code First. A noter que pour que l'auto-mapping puisse fonctionner, il faut utiliser des propriétés nommées [Classname]Id pour les clés primaires.

Fig.1



```
public abstract class Person
{
    public string Name { get; set; }
    public int DepartmentId { get; set; }
    public virtual Department Department { get; set; }
}

public class Collaborator : Person
{
    public int CollaboratorId { get; set; }
    public string ManagerCode { get; set; }
    public virtual Manager Manager { get; set; }
}
```

EF vous permet de connecter très facilement les classes POCO à la base de données en utilisant la classe ObjectContext ou plus particulièrement la classe DbContext qui sert à relier les propriétés publiques de ces classes aux tables de la base de données. DbContext permet de travailler simplement avec les fonctionnalités les plus courantes d'ObjectContext. Ci-dessous, la classe CompanyContext hérite de la classe DbContext :

```
public class CompanyContext : DbContext
{
    public CompanyContext() : base(«CompanyDatabase») { }

    public DbSet<Collaborator> Collaborators { get; set; }
    public DbSet<Department> Departments { get; set; }
    public DbSet<Manager> Managers { get; set; }
}
```

Le code est maintenant prêt, la dernière étape consiste en la connexion avec la base de données. Il est possible par exemple d'utiliser le fichier de configuration (web.config ou app.config) pour définir la chaîne de connexion. EF Code First utilise la convention dans laquelle les classes DbContext cherchent par défaut la chaîne de connexion qui a le même nom que la classe du contexte. Il est aussi possible de définir un autre nom via un paramètre du constructeur (dans notre exemple : « CompanyDatabase »).

```
<?xml version=»1.0» encoding=»utf-8» ?>
<configuration>
  <connectionStrings>
    <add name=»CompanyDatabase» providerName=»System.Data.SqlClient» connectionString=»Server=.\SQLEXPRESS;Database=Products;Trusted_Connection=true;»/>
  </connectionStrings>
</configuration>
```

DATA ANNOTATIONS & CODE FIRST FLUENT API

La nouvelle version permet l'utilisation des Data Annotations. Les Data Annotations sont utilisées pour la validation ainsi que pour définir le mapping avec les colonnes de la base de données. Toutes les annotations sont implémentées comme attributs. EF 4.1 nous permet cela en s'appuyant sur des attributs se trouvant dans le namespace System.ComponentModel.DataAnnotations (ce sont les mêmes attributs utilisés pour la validation ASP.NET MVC).

Quelques attributs courants :

- [Key] – spécifie quelle(s) colonne(s) défini(ssen)t la clé primaire de la table.
- [StringLength] – spécifie la longueur maximale et la longueur minimale. La longueur minimale est utilisée uniquement pour la validation et non comme contrainte de la base de données.
- [MaxLength] – peut être utilisée à la place de [StringLength] pour spécifier la longueur maximale d'une colonne.
- [ConcurrencyCheck] – flag sur les colonnes qui active le check de concurrence.
- [Required] – spécifie quelle valeur est requise pour une propriété. La colonne est flaggée comme non nulle.
- [Timestamp] – flag sur les colonnes time stamp qui est utilisé pour le check de concurrence.
- [Column] – peut être utilisé pour spécifier le nom de colonne. Si vide, le nom de la propriété est repris dans le nom de la colonne. Cet attribut peut également être utilisé pour contrôler la position d'une colonne.
- [Table] – peut être utilisé pour spécifier le nom d'une table. Si vide, le nom de la classe est repris dans le nom de la table.
- [DatabaseGenerated] peut être utilisé pour signaler que la donnée est remplie via la base de données. Les valeurs par défaut sont « Computed », « Identity » ou « None ». « None » est renseigné par défaut.
- [NotMapped] Permet de définir une propriété dans une classe sans générer de colonne dans la base de données.
- [ForeignKey] et [InverseProperty] sont utilisés comme sur des colonnes avec des clés étrangères.

Voici un exemple d'utilisation de Data Annotations appliqué à notre domaine d'entreprise. La classe Manager ne contient pas de propriétés pouvant être identifiées automatiquement comme clé primaire. Par ajout de l'attribut [Key], on définit que la propriété Manager Code doit être interprétée comme clé primaire.

On ajoute aussi la vérification de concurrence ainsi que la longueur minimale et maximale (un message d'erreur est paramétré) de la chaîne de caractères pour la propriété Name. Ces attributs sont vus en détail plus loin.

```
public class Manager : Person
{
    [Key]
    public string ManagerCode { get; set; }
    [ConcurrencyCheck]
    [MinLength(5)]
    [MaxLength(20,ErrorMessage=»Le nom ne peut pas avoir plus de 20 caractères»)]
    public new string Name { get; set; }
    public virtual ICollection<Collaborator> Collaborators { get; set; }
}
```

Dans la classe Department, on rajoute aussi une Data Annotation [Required] pour que la propriété Name soit toujours renseignée.

```
public class Department
{
    public int DepartmentId { get; set; }
    [Required]
    public string Name { get; set; }
```

```
    public virtual ICollection<Collaborator> Collaborators { get;
 set; }
}
```

Les Data Annotations sont définitivement simples à utiliser, mais il est tout de même préférable d'utiliser de la programmation, afin que les classes n'aient aucune spécificité d'EF. Pour cela, il faut utiliser Code First Fluent API en implémentant la méthode OnModelCreating(…) de la classe dérivée CompanyContext et en définissant toutes les contraintes valables pour le domaine.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Department>().Property(dp => dp.Name).
IsRequired();
   modelBuilder.Entity<Manager>().HasKey(ma => ma.ManagerCode);
    modelBuilder.Entity<Manager>().Property(ma => ma.Name)
        .IsConcurrencyToken(true)
        .HasMaxLength(20);
}
```

On peut même aller plus loin et décrire les opérations complexes en caractérisant les relations entre tables et colonnes. Le code ci-dessous montre que l'entité Manager contient un Department, la clé reliant les deux est DepartmentId. De plus, la suppression de Manager entraîne la suppression du Department. (CascadeOnDelete).

```
modelBuilder.Entity<Manager>()
    .HasRequired(d => d.Department)
    .WithMany()
    .HasForeignKey(d => d.DepartmentId)
    .WillCascadeOnDelete();
```

VALIDATION DES DONNÉES

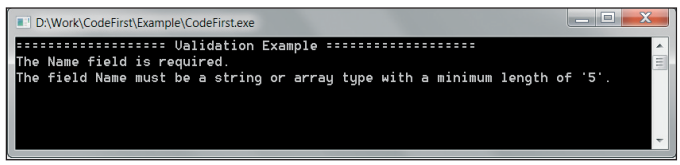
La validation des données est une problématique essentielle dans toute application. Il est important de s'assurer de la qualité et de l'intégrité des données afin de pouvoir les exploiter correctement et d'éviter les erreurs d'intégration dans la base.
Prenons l'exemple suivant : on s'assure que la propriété associée au champ Name est obligatoire et que sa valeur a une taille comprise entre 5 et 20 caractères via les Data Annotations.

```
[MinLength(5)]
[MaxLength(20,ErrorMessage=»Le nom ne peut pas avoir plus de 20
 caractères»)]
public new string Name { get; set; }
```

Par défaut, la validation va s'effectuer au moment de la sauvegarde des données (lors de l'appel de la méthode Save), mais il est également possible de valider les objets plus tôt en appelant la méthode GetValidationErrors(…).

```
public static void Validation()
{
    using (var context = new CompanyContext())
    {
        var manager = new Manager() { Name = string.Empty };
        context.Managers.Add(manager);

        var validationErrors = context.GetValidationErrors()
            .Where(vr => !vr.IsValid)
            .SelectMany(vr => vr.ValidationErrors);

        foreach (var error in validationErrors)
        {
            Console.WriteLine(error.ErrorMessage);
        }

        Console.ReadKey();
    }
}
```

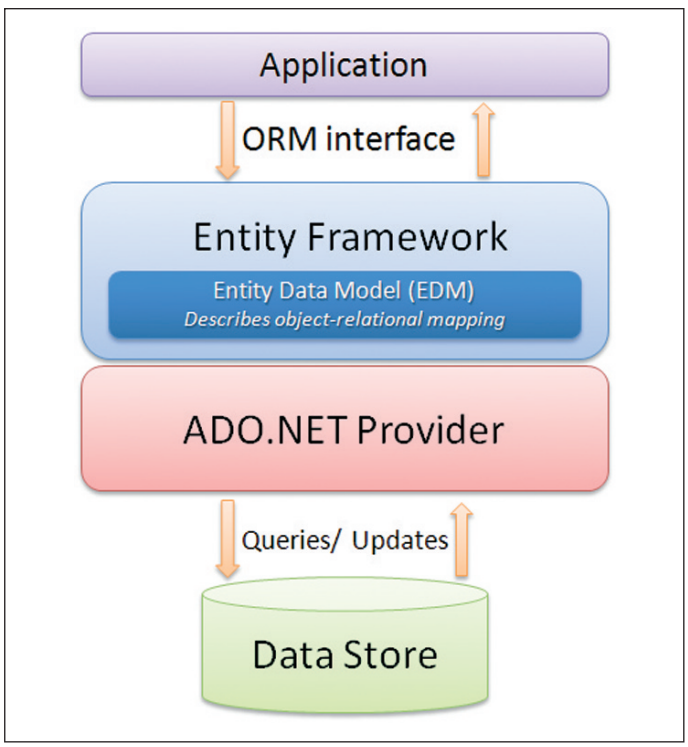
Quand le code s'exécute, la validation va échouer. Des messages d'erreurs apparaissent pour chaque contrainte non respectée. Dans notre cas, le minimum requis est 5 caractères, mais la propriété ne contient pas de valeur, ainsi un message générique est affiché [Fig.2].



Il existe bien évidemment d'autres Data Annotation permettant de valider les types, le format, les valeurs minimales et maximales autorisées, le respect d'une expression régulière…

AUDIT DES DONNÉES

Les propriétés de tout objet géré par EF peuvent être auditées. Cela est très utile lorsque l'on souhaite visualiser les changements effectués avant de sauvegarder, annuler les modifications en cours (on remplace la valeur actuelle par la valeur d'origine), ou encore anticiper un problème de concurrence lors de la sauvegarde.



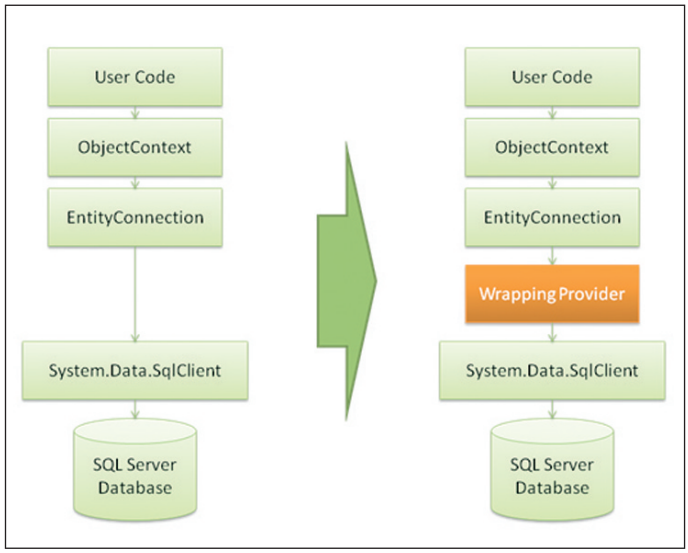
EF 4.1 vous permet d'accéder aux versions suivantes des propriétés
- d'une entité :
- la valeur en base de données (DataBase Value)
- la valeur initiale (Original Value)
- la valeur actuelle (Current Value)

```
public static void Audit()
{
    using (var context = new CompanyContext())
    {
        var manager = context.Managers.Find(«JDO»);

        using (var contextDB = new CompanyContext())
        {
            contextDB.Database.SqlCommand(
                «UPDATE managers SET Name = Name + '_DB' WHERE Manager
Code = 'JDO'»);
        }

        manager.Name = manager.Name + «_Memory»;

        string value = context.Entry(manager).Property(m => m.
Name).OriginalValue;
       Console.WriteLine(string.Format(«Original Value : {0}»,
 value));

        value = context.Entry(manager).Property(m => m.Name).
CurrentValue;
       Console.WriteLine(string.Format(«Current Value : {0}»,
 value));

       value = context.Entry(manager).GetDatabaseValues().Get
Value<string>(«Name»);
       Console.WriteLine(string.Format(«DB Value : {0}», value));

       Console.ReadKey();
    }
}
```

L'exécution de ce code permet de visualiser les différentes valeurs de la propriété Name de l'objet Manager [Fig.3]. Mettre en place l'audit est donc très simple, EF 4.1 contient déjà tout le nécessaire !

GESTION DE LA CONCURRENCE

L'un des problèmes fondamentaux liés à la gestion de base de données, est de pouvoir garantir la cohérence forte de celle-ci. Pour cela, on doit se munir d'un système de gestion de concurrence qui doit respecter les deux principes de base suivants :
- l'exécution simultanée des transactions produit les mêmes résultats que leur exécution séquentielle [Gardarin (1988)]
- l'exécution simultanée des transactions produit les mêmes résultats que l'exécution séquentielle des transactions dans l'ordre strict de leur arrivée [Rahgozar (1987)]



Nous allons voir dans cet exemple comment EF 4.1 nous permet, via la classe DbContext et ses multiples méthodes, de construire un gestionnaire de concurrence qui nous offre en cas de transactions simultanées l'application des deux solutions les plus courantes à cette problématique.

First Wins

First Wins est une stratégie de résolution de conflit dans laquelle la première modification est la gagnante. Les modifications apportées par la première transaction sont celles conservées, on notifie alors l'émetteur qui initie la seconde transaction qu'une modification des informations a déjà eu lieu (une exception peut être retournée).
Tout d'abord, nous devons déterminer quelles sont les propriétés sensibles à la concurrence, toutes les propriétés n'étant pas forcément critiques et ne nécessitant pas d'intégrer de processus de résolution de concurrence. Par exemple, si votre classe contient une propriété contenant la dernière date de mise à jour, il n'est pas nécessaire d'avoir de contrôle de concurrence sur cette propriété.
Pour réaliser ce check, il nous suffit d'ajouter l'attribut ConcurrencyCheck sur les propriétés de la classe de la manière suivante :

```
[ConcurrencyCheck]
public new string Name { get; set; }
```

On peut aussi utiliser la CodeFirst Fluent API sur les propriétés de la classe de la manière suivante :

```
modelBuilder.Entity<Manager>().Property(ma => ma.Name).IsConcur
rencyToken(true);
```

La question de la performance est à considérer. En effet, plus on aura de propriétés à contrôler, plus les performances se dégradent. Il est nécessaire d'utiliser cet attribut avec considération.

```
public static void FirstWins()
{
    using (var context = new CompanyContext())
    {
        var manager = context.Managers.Find(«JDO»);
        Console.WriteLine(«Initial Value: « + manager.Name);
```

```
        using (var contexDB = new CompanyContext())
        {
            contexDB.Database.SqlCommand(
                  «UPDATE managers SET Name = 'FirstWins' WHERE
 ManagerCode = 'JDO'»);
            Console.WriteLine(«DB Updated Value: FirstWins»);
        }

        manager.Name = «NewName»;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            ex.GetEntry(context).Reload();
        }

        manager = context.Managers.Find(«JDO»);
        Console.WriteLine(«Final Value: « + manager.Name);
    }
}
```

Premièrement, on récupère un manager depuis la base de données. Puis cet enregistrement est mis à jour directement en base afin de simuler un enregistrement concurrent. On modifie et on sauvegarde l'enregistrement avec la méthode SaveChanges(). Une exception de type DbUpdateConcurrencyException provenant du problème lié à la concurrence est retournée. C'est à ce niveau que le problème doit être géré en appliquant la règle choisie. Ici, on décide de conserver les premiers changements (First Wins) et on met à jour le contexte avec les valeurs issues de la base de données via la ligne de code ex.GetEntry(context).Reload().

Last Wins

C'est le mode de résolution par défaut appliqué par EF 4.1. C'est toujours celui qui écrit en dernier qui gagne. Les modifications liées à la première transaction sont écrasées. Si on souhaite être informé de la survenance de cas d'écrasement d'une transaction, on utilise l'attribut ConcurrencyCheck pour notifier l'utilisateur, écrire dans un fichier de log ou tout autre traitement.

```
public static void LastWins()
{
    using (var context = new CompanyContext())
    {
        bool savedChanges = false;

        var manager = context.Managers.Find(«JDO»);
        Console.WriteLine(«Initial Value: « + manager.Name);

        using (var contexDB = new CompanyContext())
        {
            contexDB.Database.SqlCommand(
               «UPDATE managers SET Name = 'LastWins' WHERE Mana
gerCode = 'JDO'»);
            Console.WriteLine(«DB Updated Value: LastWins»);
        }

        manager.Name = «NewName»;

        while (!savedChanges)
        {
            try
            {
                context.SaveChanges();
                savedChanges = true;
            }
            catch (DbUpdateConcurrencyException ex)
            {
                var entry = ex.GetEntry(context);
                entry.OriginalValues.SetValues(entry.GetDatabase
Values());
            }

        }

        manager = context.Managers.Find(«JDO»);
        Console.WriteLine(«Final Value: « + manager.Name);
    }
}
```

Contrairement au cas précédent, on relance la sauvegarde jusqu'au moment où celle-ci est réalisée sans exception. Dans le cas où une exception apparaît pendant cette opération, on synchronise le contexte (en remplaçant la valeur d'origine par celle remontée dans l'exception), puis on retente une sauvegarde. Ainsi, on est en mesure d'éviter la concurrence de transactions multiples tout en permettant aux développeurs de pouvoir ajouter des traitements (de log par exemple) lorsque l'exception DbUpdateConcurrencyException est levée.

CONCLUSION

Nous espérons que cette présentation de la nouvelle version d'EF de Microsoft vous a donné envie de jouer avec ! Même nous, qui sommes des puristes du code et du design, sommes conquis par la simplicité et la puissance du framework.
Grâce à ces nouveautés, EF devient l'un des produits les plus complet du marché, car il est le seul à supporter les 3 approches : Code First, Database First et Model First.
Evidemment, il reste encore des choses à améliorer : la possibilité de mettre à jour automatiquement la base de données lorsque la structure des classes est modifiée, ou encore certaines fonctionnalités manquantes comme le support des procédures stockées.
La bonne nouvelle, c'est que l'équipe en charge du développement d'EF est déjà retournée au travail. Vivement la prochaine version !

Retrouvez le **code** des exemples sur http://codefirst.codeplex.com !

■ Jason **De Oliveira**
Practice Manager & Solutions Architect | MVP C#
.Net Rangers by Sogeti
Blog: http://jasondeoliveira.com

■ Fathi **Bellahcene**
Software Architect .Net Rangers by Sogeti
Blog: http://blogs.codes-sources.com/fathi