

# PetaPoco 5.1

## Le micro - ORM

# PetaPoco

## Sommaire

- Pourquoi utiliser un ORM
- Les ORMs en .Net
- Un cas à part : les micro-ORM
- PetaPoco
- Mise en pratique
- Les Templates T4
- Génération des POCOs
- La gestion des relations

# Pourquoi utiliser un ORM ?

## Inconvénients

- « C'est une perte de temps ! Il va falloir former les développeurs alors qu'ADO tout le monde connaît » **TEMPS (ARGENT)**
- « On va perdre en performances, rien n'est plus rapide qu'un appel direct à la DB sans fioritures » **PERFORMANCE**
- « Il existe plusieurs ORMs, lequel est le meilleur pour le projet ?
- Lequel sera le mieux maintenu dans le temps ? » **CHOIX**
- « Si nous sommes liés à un ORM, pouvons-nous en changer ?  
En utiliser plusieurs conjointement ? Tout ceci ne va-t-il pas devenir complexe en cas d'évolution ? » **EVOLUTION**
- « Comment débbugger le code d'un ORM ? Va-t-on même se rendre compte d'un soucis lié à un ORM ? » **DEBUGGAGE**

# Pourquoi utiliser un ORM ?

## Avantages

- Gain de **productivité** : Ne pas réinventer la roue à chaque projet
- Evite les dérapages en proposant une seule **bonne pratique** pour toute la durée du projet
- Ajout de **flexibilité** : La plupart des ORMs permettent de partir soit d'une DB existante soit des modèles

# Les ORMs en .Net

## Liste non-exhaustive

### 1) NHIBERNATE

Mature, robuste, difficile à apprendre, ne supporte pas Linq

### 2) LinqToSql

Intégré au Framework, simple, support de Linq mais limité à SQLServer, tant à être moins utilisé

### 3) Entity Framework

Simple à apprendre et à utiliser, nombreuses fonctionnalités, parfois en deçà au niveau des performances

### 4) SUBSONIC

Léger, simple, compatible avec Linq mais vient d'un éditeur tiers, sur le déclin

### 5) BLToolKit

Simple, efficace mais peu utilisé

# Les micro-ORMs

## L'alternative

- L'idée des micro-ORMs est simple à comprendre:
- Réaliser les tâches de base attendues (relationnel → objet)
- Prendre uniquement les fonctionnalités les plus essentielles des ORM standards
- Sauvegarder les performances en laissant le développeur faire ses requêtes SQL sans couches aidantes (ADO apparent). Cacher le SQL n'est peut-être pas toujours une bonne idée

Performance of SELECT mapping over 500 iterations - POCO serialization

Method	Duration	Remarks
Hand coded (using a <code>SqlDataReader</code> )	47ms	
Dapper <code>ExecuteMapperQuery</code>	49ms	
<a href="#">ServiceStack.OrmLite</a> ( <code>QueryById</code> )	50ms	
<a href="#">PetaPoco</a>	52ms	<a href="#">Can be faster</a>
BLToolkit	80ms	
SubSonic CodingHorror	107ms	
NHibernate SQL	104ms	
Linq 2 SQL <code>ExecuteQuery</code>	181ms	
Entity framework <code>ExecuteStoreQuery</code>	631ms	

# PetaPoco

## Présentation

- Vient de « Massive » un autre micro-ORM abandonné très connu
- Utilise les templates T4 pour la génération des modèles
- Compatible avec un grand nombre de DB (SQLServer, MySQL,...)
- Bonnes performances
- Consiste en une seule classe à placer dans le projet
- Grande communauté y compris pour quelques forks
- Supporte les transactions
- Supporte .Net Core dans sa dernière version
- Possède une librairie de tests unitaires
- Pas encore de support d'Async

# PetaPoco

## Prérequis

PetaPoco ne gère pas l'équivalent d'un « Code first » comme le ferait Entity Framework. Il faut donc créer la base de données par nous même.

Plusieurs choix sont possibles :

- Un projet SQL Server Database (classique)
- Le Server Explorer de Visual Studio
- Le Server Management Studio d'SQL Server
- Une migration avec Entity Framework ...



# PetaPoco

## Mise en pratique

- 1) Il faut définir manuellement la connection string dans le fichier de configuration App.config ou Web.config:

```
<connectionStrings>  
<add name="PetaExample"  
  connectionString="Data Source=.\SQLSERVER;  
  Initial Catalog=Cinema;  
  Integrated Security=True;  
  Connect Timeout=300;"  
  providerName="System.Data.SqlClient" />  
</connectionStrings>
```

### 2) Créer la classe, la POJO

```
public class Movie
{
    public int MovieId { get; set; }
    public string Titre { get; set; }
    public string Realisateur { get; set; }
    public int Annee { get; set; }
    public DateTime DateAchat { get; set; }
}
```






Notez l'absence de « data annotations »

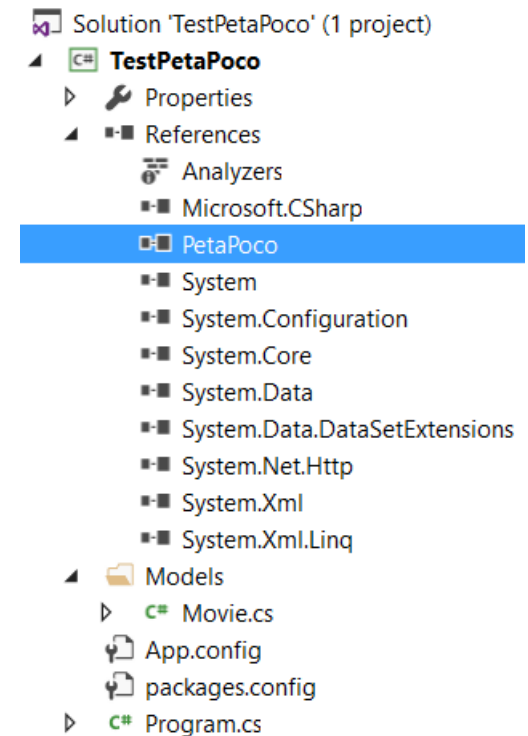
# PetaPoco

## Mise en pratique

### 3) Installer PetaPoco via Nuget

Pour ce premier test nous installerons la version « compilée ». Cette version ne comprend pas de templates. Seule la classe PetaPoco a été installée.

	<b>PetaPoco</b> by Collaborating Platypus, <b>138K</b> downloads PetaPoco is a tiny, single file .NET data access layer inspired by Massive that works with both non-dynamic POCO objects and dynamics.	✓ v4.0.3 v5.1.259
	<b>PetaPoco.Core</b> by Collaborating Platypus, <b>60,5K</b> downloads PetaPoco is a tiny, single file .NET data access layer inspired by Massive that works with both non-dynamic POCO objects and dynamics.	v5.1.259
	<b>PetaPoco.Core.Compiled</b> by Collaborating Platypus, <b>24,6K</b> downloads PetaPoco is a tiny, single file .NET data access layer inspired by Massive that works with both non-dynamic POCO objects and dynamics.	✓ v5.1.259
	<b>PetaPoco.Glimpse</b> by Adam Schröder, <b>2,35K</b> downloads Glimpse Plugin for PetaPoco	v1.0.1
	<b>PetaPocoRepository</b> by David McLean, <b>6,96K</b> downloads A very simple repository for PetaPoco, because you have better things to be getting on with!	v0.2.0



### 4) Utilisation de PetaPoco pour faire une requête « Select »

```
using (var db = new Database("PetaExample"))
{
    try
    {
        var result = db.Query<Movie>("select * from Movie").ToList();
        result.ForEach(PrintResults);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

N'hésitez pas à faire un F12 sur les méthodes et classes nouvelles pour découvrir les commentaires qui y sont associées. Et pensez à coder « PrintResults » !.

# PetaPoco

## Mise en pratique


- 5) Pour ajouter un film nous allons créer un objet de type Movie que nous allons compléter. Les 3 paramètres sont : le nom de la table, le nom de la clé primaire et enfin l'objet. Tout cela est nécessaire pour qu'après l'insertion l'objet possède l'ID tel qu'il a été créé en base de données.

```
// Création du film
var a=new Movie();
a.Titre="Alien: Covenant";
a.Realisateur="Ridley Scott";
a.Annee=2017;
a.DateAchat=DateTime.Now;
```

```
// Insertion
db.Insert("Movie", "Movied", a);
```

```
// Si la classe Movie est décorée alors :
db.Insert(a);
```

```
namespace TestPetaPoco.Models
{
    //Les décorations sont possibles
    //avec PetaPoco et permettent de
    //se passer de certains paramètres
    [TableName("Movie")]
    [PrimaryKey("MovieId")]
    2 references
    public class Movie
```



- 6) La logique de l'Update est similaire. Disons que je veux trouver un film, modifier son titre et le réenregistrer :

```
// Trouver un film
```

```
var a=db.SingleOrDefault<Movie>("SELECT * FROM Movie WHERE MovieId=@0", 3);
```

```
// Trouver un film si la classe est décorée
```

```
var a=db.SingleOrDefault<Movie>("WHERE MovieId=@0", 3);
```

```
// Modifier le titre
```

```
a.Titre="La soupe aux choux";
```

```
// Sauvegarder
```

```
db.Update("Movie", "MovieId", a);
```

```
// Si la classe Movie est décorée alors :
```

```
db.Update(a);
```

### 7) De même pour le Delete...

```
// Trouver un film
```

```
var a=db.SingleOrDefault<Movie>("SELECT * FROM Movie WHERE MovieId=@0", 3);
```

```
// Trouver un film si la classe est décorée
```

```
var a=db.SingleOrDefault<Movie>("WHERE MovieId=@0", 3);
```

```
//Supprime l'objet Movie
```

```
db.Delete("Movie", "MovieId", a);
```

```
//Si la classe Movie est décorée alors :
```

```
db.Delete(a);
```

# PetaPoco

## Les templates T4

PetaPoco utilise la technologie T4 pour générer automatiquement les modèles et le repository.

T4 est l'acronyme de **T**ext **T**emplate **T**ransformation **T**oolkit.

Le fonctionnement du template est simple :

- Trois fichiers sont présents (PetaPoco.Core.ttinclude, PetaPoco.Generator.ttinclude, Database.tt)
- Le fichier Database.tt est le seul à devoir être modifié conformément à la connection string et aux réglages désirés
- Sauvegarder le fichier déclenche immédiatement la génération des modèles



# PetaPoco

## Génération des POCOs

### Le fichier Database.tt

```
// Settings
ConnectionStringName = " PetaExample "; //Nom de la chaîne de connexion
Namespace = "TestPetaPoco.Models"; //Nom souhaité pour l'espace de noms
RepoName = ""; //Nom du repository auto-généré
GenerateOperations = true;
GeneratePocos = true; //Faut-il générer les classes ?
GenerateCommon = true;
ClassPrefix = ""; //Préfixe désiré pour chaque classe
ClassSuffix = "POCO"; //Suffixe désiré pour chaque classe
TrackModifiedColumns = false;
ExplicitColumns = true; //Les colonnes sont définies explicitement
ExcludePrefix = new string[] {}; //Exclure des tables d'après leurs préfixes
```

# PetaPoco

## La relation 1 - N

Prenons le cas de « Personne » possédant un « Genre ».

Nous souhaitons pour des raisons de facilité et d'affichage que lorsque nous requêtons la table Personne nous obtenions les personnes **ET** leur genre.

Après avoir ajouté une propriété « Civilité » à la classe Personne via la technique de la classe partielle, nous allons nous appuyer sur le mappage automatique de PetaPoco :

```
var personnes = db.Fetch<PersonnePOCO, GenrePOCO, PersonnePOCO>((p, g) =>
{ p.Civilite = g.Nom; return p; }, @"SELECT * FROM Personne left JOIN Genre ON
Personne.GenreId = Genre.GenreId");
```

Le résultat sera une liste de personnes dont la propriété « Civilité » sera complétée.