**[Utiliser ASP.NET MVC et NHibernate (1° partie)](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-1/)**

2010-01-29 #mvc,#nhibernate

Ceci est la traduction du billet “[Using ASP.NET MVC and NHibernate (Part 1)](http://forerunnerg34.wordpress.com/2009/11/03/using-asp-net-mvc-and-nhibernate-part-1/)” de César Intriago.

Dans cet article je vais expliquer comment installer et configurer NHibernate pour l’utiliser dans une application ASP.NET MVC. Ceci est le premier article d’une série dans laquelle je montrerai comment installer NHibernate pour une application .Net.



**NHibernate c’est quoi ?**

NHibernate est un outil de mapping Objet-Relationnel (object-relational mapping ou ORM en anglais) pour .Net qui permet de faire correspondre des modèles orientés objets avec une base de données. Comme vous le verrez dans cet article, NHibernate s’occupera de la plupart des tâches relatives à la couche de persistance. Vous pouvez avoir plus d’informations au sujet de NHibernate à partir du site consacré à la [communauté NHibernate](http://nhforge.org/).

Vous pouvez [télécharger la dernière version de NHibernate sur SourceForge](http://sourceforge.net/projects/nhibernate/files/). Les exemples de code dans cet article sont basés sur NHibernate 2.1.1 (la version la plus récente disponible au moment de la rédaction de cet article).

**Installer NHibernate**

Téléchargez et dézippez NHibernate sur votre ordinateur. Ca y est, NHibernate est installé.

**Créer le projet ASP.NET MVC**

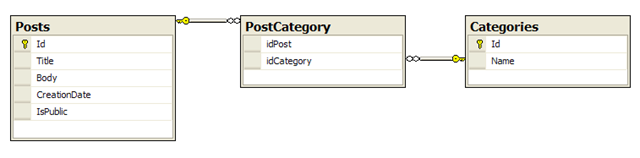
Créez un nouveau projet ASP.NET MVC et **n’oubliez pas de créer un projet de test**.

Ajoutez deux nouveaux projets de type “Bibliothèque de classe” à votre solution : **Infrastructure** et **Core** (pour que tout soit structuré du mieux possible).

Dans le projet Core, ajoutez une référence à l’assemblie **NHibernate**.

**La base de données**

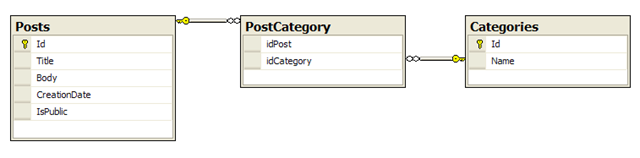
Pour notre exemple, nous allons créer un modèle contenant des **Posts** (des billets en français) et des **Categories**. C’est un modèle très simple qui nous aidera à bien comprendre comment NHibernate fonctionne : un billet (ou post) peut appartenir à une ou plusieurs catégories et une catégorie peut contenir zéro ou plus de zéro billets :



Nous allons utiliser SQL Server 2008 Express Edition pour créer notre base de données, une autre solution étant de créer notre modèle dans Visual Studio et de demander à NHibernate de créer la base de données pour nous. Mais dans notre cas, nous allons créer manuellement la base de données.

**Créer le Modèle**

L’étape suivante va consister à créer notre modèle qui est une représentation orientée objet de notre base de données. Pour cela nous allons utiliser le concepteur de classes de Visual Studio :



**Les Repository**

Un Repository nous permet de créer, sélectionner, modifier et supprimer nos objets et il est indépendant de la base de données. Pour ce tutoriel, nous avons besoin de créer deux repositorys : **PostRepository** et **CategoryRepository**. Tout deux vont implémenter la même interface présentée ci-dessous :

**IRepository.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Core

{

public interface IRepository<T>

{

void Save(T entity);

void Update(T entity);

void Delete(Guid id);

T GetById(Guid id);

T GetAll();

}

}

Pour créer nous deux repositorys, nous avons d’abord besoin d’une une classe helper pour créer une **session NHibernate** sur notre base de données :

**NHibernateHelper.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using NHibernate.Cfg;

using NHibernate;

namespace Core.Domain.Repositories

{

public class NHibernateHelper

{

private static ISessionFactory \_sessionFactory;

private static ISessionFactory SessionFactory

{

get

{

if (\_sessionFactory == null)

{

var configuration = new Configuration();

configuration.Configure();

\_sessionFactory = configuration.BuildSessionFactory();

}

return \_sessionFactory;

}

}

public static ISession OpenSession()

{

return SessionFactory.OpenSession();

}

}

}

Ensuite, nous pouvons créer les repositorys :

**PostRepository.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Core.Domain.Model;

using NHibernate;

using NHibernate.Criterion;

namespace Core.Domain.Repositories

{

public class PostRepository: IRepository<Post>

{

#region IRepository<Post> Members

void IRepository<Post>.Save(Post entity)

{

using (ISession session = NHibernateHelper.OpenSession())

{

using (ITransaction transaction = session.BeginTransaction())

{

session.Save(entity);

transaction.Commit();

}

}

}

void IRepository<Post>.Update(Post entity)

{

using (ISession session = NHibernateHelper.OpenSession())

{

using (ITransaction transaction = session.BeginTransaction())

{

session.Update(entity);

transaction.Commit();

}

}

}

void IRepository<Post>.Delete(Guid id)

{

using (ISession session = NHibernateHelper.OpenSession())

{

using (ITransaction transaction = session.BeginTransaction())

{

session.Delete(id);

transaction.Commit();

}

}

}

Post IRepository<Post>.GetById(Guid id)

{

using (ISession session = NHibernateHelper.OpenSession())

return session.CreateCriteria<Post>().Add(Restrictions.Eq("Id", id)).UniqueResult<Post>();

}

Post IRepository<Post>.GetAll()

{

throw new NotImplementedException();

}

#endregion

}

}

**CategoryRepository.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Core.Domain.Model;

using NHibernate;

using NHibernate.Criterion;

namespace Core.Domain.Repositories

{

public class CategoryRepository: IRepository<Category>

{

#region IRepository<Category> Members

void IRepository<Category>.Save(Category entity)

{

using (ISession session = NHibernateHelper.OpenSession())

{

using (ITransaction transaction = session.BeginTransaction())

{

session.Save(entity);

transaction.Commit();

}

}

}

void IRepository<Category>.Update(Category entity)

{

using (ISession session = NHibernateHelper.OpenSession())

{

using (ITransaction transaction = session.BeginTransaction())

{

session.Update(entity);

transaction.Commit();

}

}

}

void IRepository<Category>.Delete(Guid id)

{

using (ISession session = NHibernateHelper.OpenSession())

{

using (ITransaction transaction = session.BeginTransaction())

{

session.Delete(id);

transaction.Commit();

}

}

}

Category IRepository<Category>.GetById(Guid id)

{

using (ISession session = NHibernateHelper.OpenSession())

return session.CreateCriteria<Category>().Add(Restrictions.Eq("Id", id)).UniqueResult<Category>();

}

Category IRepository<Category>.GetAll()

{

throw new NotImplementedException();

}

#endregion

}

}

Comme vous pouvez le constater, c’est dans les repositorys que nous plaçons le code pour appeler les méthodes NHibernate, ce que nous faisons en créant un objet **Session** en premier.

**Où est-ce que nous en sommes…**

Avant de continuer, revoyons un peu ce que nous avons déjà fait :

* Nous avons créé un projet *\*Core* - qui référence la librairie NHibernadte.dll
* Dans ce projet **Core**, nous avons créé deux classes qui représentent notre modèle : “Post.cs” et “Category.cs”, la classe Post ayant une collection de catégories.
* Nous avons cré deux repositorys pour enregistrer, modifier, supprimer et retrouver les objets de notre modèle.

**Les Mappings**

Il est temps de faire quelque chose dans notre projet **Infrastructure** où nous mapperont notre modèle à notre base de données, ce qui se fait via des fichiers XML avec NHibernate. Nous allons suivre la convention de nommage qui existe au sujet des fichiers maps : [ClassName].hbm.xml.

Nous allons créer deux nouveaux fichiers : **Category.hbm.xml** et **Post.hbm.xml**. Le contenu de chacun de ces fichiers fait correspondre une classe à une table, une propriété à une colonne, et indique aussi le type de données. Pour cette première partie du tutoriel, nous avons besoin de créer le fichier **Category.hbm.xml** :

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

namespace="Core.Domain.Model"

assembly="Core">

<class name="Category" table="Categories" dynamic-update="true">

<cache usage="read-write"/>

<id name="Id" column="Id" type="Guid">

<generator class="guid"/>

</id>

<property name="Name" length="100"/>

</class>

</hibernate-mapping>

Important : Vous devez définir l’action de génération de ces deux fichiers à “**Ressource incorporée**” afin que NHibernate puisse trouver le bon fichier dans l’assembly.

**Configurer NHibernate**

Nous sommes presque prêts. La prochaine étape va consister définir la chaine de connexion à la base de données et quelques paramètres pour NHibernate. Cette configuration doit être stockée dans un fichier “hibernate.cfg.xml” (**définissez l’action de génération de ce fichier à “Ressource incorporée”**). NHibernate dispose d’une fonctionnalité spéciale de “Lazy-Loading” pour *dynamic proxy systems*, et nous devons ajouter les références suivantes à notre projet Infrastructure pour qu’il puisse l’utiliser :

* Castle.Core
* Castle.DynamicProxy2
* NHibernate.ByteCode.Castle.dll

Nous pouvons alors créer le fichier de configuration XML suivant :

**hibernate.cfg.xml**

<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">

<session-factory>

<property name="connection.driver\_class">NHibernate.Driver.SqlClientDriver</property>

<property name="connection.connection\_string">server=.\SQLExpress;database=NHibernate101;Integrated Security=true;</property>

<property name="show\_sql">true</property>

<property name="dialect">NHibernate.Dialect.MsSql2008Dialect</property>

<property name="cache.use\_query\_cache">false</property>

<property name="adonet.batch\_size">100</property>

<property name="proxyfactory.factory\_class">NHibernate.ByteCode.Castle.ProxyFactoryFactory, NHibernate.ByteCode.Castle</property>

<mapping assembly="Infrastructure" />

</session-factory>

</hibernate-configuration>

La ligne <property name="proxyfactory.factory\_class">... utilise la fonctionalité de “lazy loading” au sujet de laquelle vous trouverez plus d’informations sur le site d’[Hibernate](https://www.hibernate.org/162.html).

**Test unitaire de NHibernate**

Il est enfin temps de tester notre configuration de NHibernate ! Nous allons ajouter quelques données à notre table “Categories”. Pour cela nous avons besoin que notre projet de test référence l’assembly NHibernate et d’y copier le fichier hibernate.cfg.xml.

Ajouter les références suivantes :

* Castle.Core
* Castle.DynamicProxy2
* Infrastructure
* NHibernate
* NHibernate.ByteCode.Castle

Puis ajouter la méthode de test suivante :

[TestMethod]

[DeploymentItem("hibernate.cfg.xml")]

public void CanCreateCategory()

{

IRepository<Category> repo = new CategoryRepository();

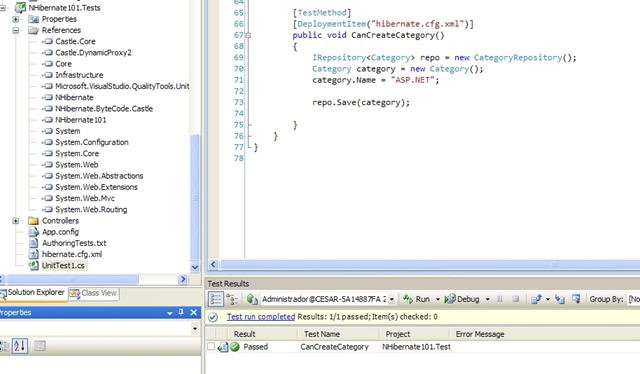
Category category = new Category();

category.Name = "ASP.NET";

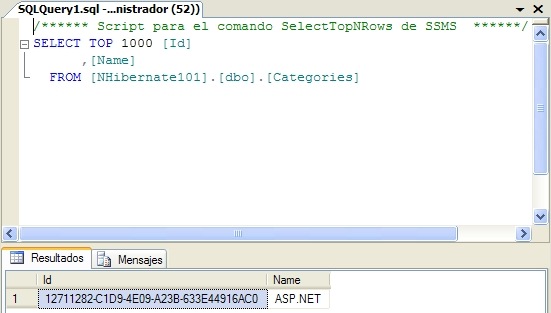
repo.Save(category);

}

Lancer le projet de test et notre méthode de test CanCreateCategory devrait réussir le test (avec un peu de chance) :



Nous pouvons alors vérifier dans la base de données que la nouvelle catégorie a bien été ajoutée :



Dans la [prochaine partie](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-2/) de ce tutoriel, nous complèterons le modèle et les tests unitaires puis nous attaquerons la création de notre application ASP.NET MVC.

[**Utiliser ASP.NET MVC et NHibernate (2° partie)**](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-2/)

2010-01-29 #mvc,#nhibernate

Ceci est la traduction du billet “[Using ASP.NET MVC and NHibernate (Part 2)](http://forerunnerg34.wordpress.com/2009/11/03/using-asp-net-mvc-and-nhibernate-part-2/)” de César Intriago.

Et voici la deuxième partie d’une série d’article consacré à l’utilisation de NHibernate dans une application ASP.NET MVC. Vous pouvez lire la [première partie de ce tutoriel ici](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-1/).

**Représenter une relation many-to-many**

Nous allons continuer en créant un fichier de mapping pour la classe Post. Celui-ci est légèrement différent de celui pour la classe Category étant donné que nous devons représenter une relation plusieurs-à-plusieurs entre les tables Posts et Categories, ce que nous réaliserons en utilisant un élément BAG dans le fichier de mapping. Voici le code de notre fichier Post.hbm.xml :

Important : Vous devez définir l’action de génération de tous les fichiers de mapping à “**Ressource incorporée**” afin que NHibernate puisse trouver le bon fichier dans l’assembly.

**Post.hbm.xml**

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"

namespace="Core.Domain.Model"

assembly="Core">

<class name="Post" table="Posts" dynamic-update="true">

<cache usage="read-write"/>

<id name="Id" column="Id" type="Guid">

<generator class="guid"/>

</id>

<property name="Title" length="100"/>

<property name="Body"/>

<property name="CreationDate" type="datetime"/>

<property name="IsPublic" type="bool"/>

<bag name="Categories" table="PostCategory" lazy="false" >

<key column="idPost" ></key>

<many-to-many class="Category" column="idCategory" ></many-to-many>

</bag>

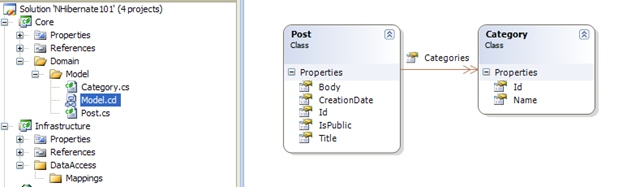
</class>

</hibernate-mapping>

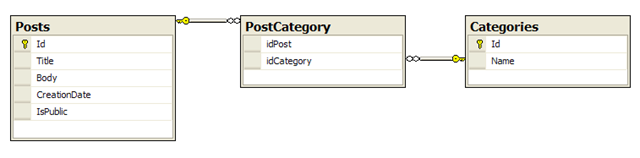
Voici quelques explications sur ce que nous avons fait au niveau de l’élément <bag> :

* L’attribut name défini le nom de la propriété de la classe Post où nous stockerons la collection des catégories.
* L’attribut table correspond au nom de la table dans la base de données qui relie les tables Posts et Categories.
* L’attribut key.column est le nom de l’identifiant de la table Posts.
* L’attribut class représente le nom de la classe Category dans le modèle d’objet.
* L’attribut many-to-many.column est le nom de l’identifiant de la table Categories.

Pour vous aider à mieux comprendre ce fichier de mapping, voici tout d’abord à quoi ressemble le diagramme de classe :



puis le schéma de la base de données correspondante :



Comme vous pouvez le constater dans le diagramme de classe, nous n’avons pas besoin de créer une classe pour la table PostCategory.

**Test unitaire**

Nous allons maintenant tester nos repositorys et vérifier que nous pouvons créer des billets à l’aide des tests unitaires suivants :

using System;

using System.Text;

using System.Collections.Generic;

using System.Linq;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Core.Domain.Model;

using Core.Domain.Repositories;

using Core;

namespace NHibernate101.Tests

{

[TestClass]

public class RepositoriesTest

{

IRepository<Category> categoriesRepository;

IRepository<Post> postsRepository;

Post testPost;

Category testCategory1;

Category testCategory2;

public RepositoriesTest()

{

}

private TestContext testContextInstance;

public TestContext TestContext

{

get

{

return testContextInstance;

}

set

{

testContextInstance = value;

}

}

[TestInitialize()]

public void CreateRepositories()

{

categoriesRepository = new CategoryRepository();

postsRepository = new PostRepository();

}

[TestMethod]

[DeploymentItem("hibernate.cfg.xml")]

public void CanCreateCategory()

{

testCategory1 = new Category() { Name = "ASP.NET" };

categoriesRepository.Save(testCategory1);

}

[TestMethod]

[DeploymentItem("hibernate.cfg.xml")]

public void CanCreatePost()

{

testPost = new Post();

testPost.Title = "ASP.NET MVC and NHibernate";

testPost.Body = "In this article I'm going to cover how to install and configure NHibernate and use it in a ASP.NET MVC application.";

testPost.CreationDate = DateTime.Now;

testPost.IsPublic = true;

testCategory2 = new Category() { Name= "ASP.NET MVC"};

categoriesRepository.Save(testCategory2);

testPost.Categories.Add(testCategory2);

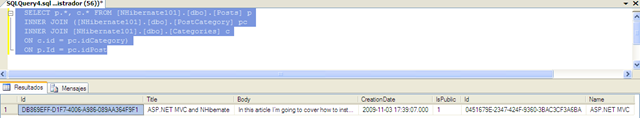
postsRepository.Save(testPost);

}

}

}

Lancer le test et s’il réussi, nous devrions retrouver le nouveau billet et la nouvelle catégorie dans la base de données :



Il s’agit là des rudiments de NHibernate. Je vous encourage à vous documenter plus en détail sur le site de la [communauté NHibernate](http://nhforge.org/). Dans la [prochaine partie](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-3-fin/) de ce tutoriel nous commencerons à développer notre application ASP.NET MVC pour gérer les billets et les catégories !

[**Utiliser ASP.NET MVC et NHibernate (3° partie)**](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-3-fin/)

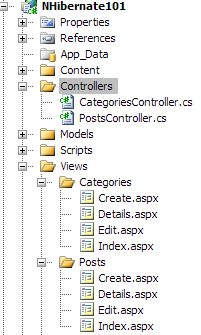
2010-01-29 #mvc,#nhibernate

Ceci est la traduction du billet “[Using ASP.NET MVC and NHibernate (Part 3)](http://forerunnerg34.wordpress.com/2009/11/05/using-asp-net-mvc-and-nhibernate-part-3-final/)” de César Intriago.

Voici le troisième et dernier billet d’une série d’articles consacré à l’utilisation de NHibernate pour développer des applications ASP.NET MVC. Les liens vers les deux premières parties de ce tutoriel sont disponibles ci-dessous :

* [Utiliser ASP.NET MVC et NHibernate (1° partie)](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-1/)
* [Utiliser ASP.NET MVC et NHibernate (2° partie)](https://blog.pagesd.info/2010/01/29/utiliser-aspnetmvc-nhibernate-2/)

Dans cet article, nous allons créer notre application ASP.NET MVC pour gérer les Posts et les Categories. Notre projet aura la structure suivante :

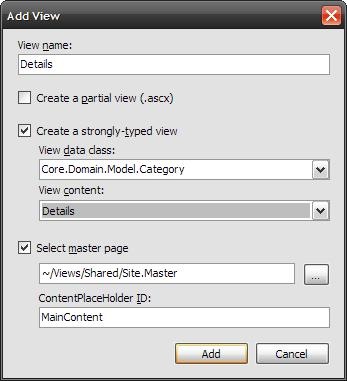


Le contrôleur CategoriesController va nous permettre de retrouver, créer, modifier et supprimer des catégories de notre repository.

Le contrôleur PostController va nous permettre de faire la même chose pour les Posts. En ce qui concerne les vues pour les Posts, je vais utiliser une **ViewModel** afin de représenter la combinaison des Posts et des Categories.

Dans notre projet ASP.NET MVC nous allons ajouter des références vers nos projets Core et Infrastructure afin de pouvoir utiliser les composants NHibernate pour la persistance que nous avons développé dans les deux premières parties.

Nous utiliserons des vues fortement typées basées sur notre modèle et ViewModel afin que Visual Studio puisse *scaffolder* les vues en s’appuyant sur ces types d’objet.

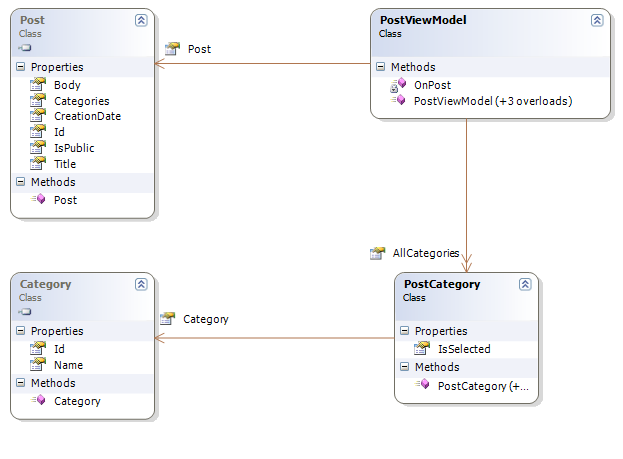


J’ai apporté quelques modification au code par rapport à la deuxième partie car j’ai détecté quelques bugs grâce aux test unitaires. Par conséquent, il est nécessaire de [télécharger la version complète du projet](http://go2.wordpress.com/?id=725X1342&site=forerunnerg34.wordpress.com&url=http://cid-926d6677262767bd.skydrive.live.com/self.aspx/ForerunnerG34/NHibernate101%20Final.zip).

**ViewModel**

Les vues pour les Posts ont besoin des objets Post et Category pour fonctionner correctement. Pour cela, un *pattern* très répandu est de passer par un modèle de vue qui sert de conteneur pour ces deux objets afin de faciliter leur utilisation au niveau de la vue.

Par conséquent, dans le cas des vues destinées aux Posts, je n’ai pas utilisé directement les objets du modèle (c’est à dire les classes Post et Category). A la place, j’ai créé une classe PostViewModel qui contient ces deux objets du modèle. De cette façon, je peux simplifier la manipulation des Posts et des Categories auxquelles ils sont rattachés comme s’il s’agissait d’une seule entité.



**Conclusion**

Configurer NHibernate pour la première fois peut demander un petit effort (ou un effort certain), mais il vous fera gagner énormément de temps pour réaliser la couche données de votre application. Et combiné à un framework tel que ASP.NET MVC, vous parvenez à une franche séparation des préoccupations (*separation of concerns* en anglais) qui vous aidera à conserver un projet bien organisé, facile à comprendre et à tester. Je suis persuadé qu’en suivant cette méthode vous obtiendrez un projet avec un code et une structure tirés à quatre épingle, d’autant plus que les modèles de Visual Studio pour ASP.NET MVC rendent ce travail encore plus facile étant donné que vous pouvez automatiser la création des vues.

C’est extraordinaire tellement c’est facile et rapide de créer une application avec ASP.NET MVC. J’attends avec impatience la sortie de la version 2.0 (en fait je teste d’ores et déjà sa Preview). C’est vraiment le moment idéal pour [apprendre ASP.NET MVC](https://blog.pagesd.info/2010/01/12/vous-devriez-apprendre-mvc/) et NHibernate.

[**Premiers essais avec LINQ to NHibernate**](https://blog.pagesd.info/2010/02/17/premiers-essais-linq-to-nhibernate/)

2010-02-17 #linq,#nhibernate

Etant donné que je débute avec NHibernate et LINQ to NHibernate, j’ai fait quelques essais pour comprendre ce qui se passe côté SQL quand je fais telle ou telle requête LINQ.

**Essai numéro 1**

Quand je fais la requête LINQ suivante :

var linq1 = (from c in \_session.Linq<Contact>()

where c.Id == id

select c).FirstOrDefault();

NHibernate génère la requête SQL ci-dessous :

SELECT top 1

this\_.Id as Id1\_0\_,

this\_.FirstName as FirstName1\_0\_,

this\_.LastName as LastName1\_0\_,

this\_.Phone as Phone1\_0\_,

this\_.Email as Email1\_0\_,

this\_.groupId as groupId1\_0\_

FROM Contacts this\_

WHERE this\_.Id = @p0; // @p0 = 13

Ensuite, lorsque j’accède à une propriété liée de l’objet Contact qui a été retourné par la requête LINQ :

string NomGroupe = linq1.Group.Name;

Alors NHibernate va générer une seconde requête SQL pour charger l’objet Group associé au contact :

SELECT group0\_.Id as Id0\_0\_,

group0\_.Name as Name0\_0\_

FROM Groups group0\_

WHERE group0\_.Id=@p0; // @p0 = 1

**Essai numéro 2**

Après quelques recherches, j’ai trouvé qu’il était possible de reproduire la méthode Include(“Entité”) de ADO.NET Entity Framework en utilisant la méthode Expand(). Cela me permet donc d’écrire ma requête LINQ de la façon suivante :

var linq2 = (from c in \_session.Linq<Contact>().Expand("Group")

where c.Id == id

select c).FirstOrDefault();

Et dans ce cas, NHibernate insère automatiquement une jointure avec la table Groups dans la requête SQL qu’il génère :

SELECT top 1

this\_.Id as Id1\_1\_,

this\_.FirstName as FirstName1\_1\_,

this\_.LastName as LastName1\_1\_,

this\_.Phone as Phone1\_1\_,

this\_.Email as Email1\_1\_,

this\_.groupId as groupId1\_1\_,

group2\_.Id as Id0\_0\_,

group2\_.Name as Name0\_0\_

FROM Contacts this\_

inner join Groups group2\_ on this\_.groupId=group2\_.Id

WHERE this\_.Id = @p0; // @p0 = 13

Comme je suis d’un naturel méfiant, je suis allé jusqu’à vérifier ce qui se passait quand j’accédais à la propriété Group de l’objet Contact :

string NomGroupe2 = linq2.Group.Name;

Et fort heureusement il ne se passe rien et NHibernate ne génère pas de requête SQL supplémentaire.

**Interlude**

OK. Mais NHibernate ça ne peut pas être aussi simple que ça et il y des tas de trucs à prendre en compte. Jusqu’à maintenant, j’ai fait mes deux essais avec le fichier de mapping suivant pour les contacts :

<?xml version="1.0" encoding="utf-8" ?>

<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" namespace="ContactManager.Models" assembly="ContactManager">

<class name="Contact" table="Contacts" dynamic-update="true">

<cache usage="read-write" />

<id name="Id" column="Id" type="integer">

<generator class="identity" />

</id>

<property name="FirstName" type="string" />

<property name="LastName" type="string" />

<property name="Phone" type="string" />

<property name="Email" type="string" />

<many-to-one name="Group" column="groupId" not-null="true" />

</class>

</hibernate-mapping>

Dans ce fichier, il y a une ligne <many-to-one name="Group" column="groupId" not-null="true" /> qui sert à définir la relation entre les tables Contacts et Groups :

* 1 contact appartient à 1 groupe
* 1 groupe peut contenir 0 à N contact(s)

Par défaut, cette relation “fonctionne” en mode lazy-loading, ce qui signifie que le groupe ne sera chargé que lorsqu’on en aura réellement besoin. Et au vu de mes deux premiers essais, je ne peux qu’être satisfait puisque c’est exactement comme cela que ça s’est passé.

**Essai numéro 3**

Et maintenant, supposons que je ne veuille plus faire de lazy-loading ? Pour commencer, je dois ajouter lazy="false" à mon fichier de mapping :

...

<many-to-one name="Group" column="groupId" not-null="true" lazy="false" />

...

Et ce coup-ci, quand je refais la même requête LINQ que lors de mon premier essai :

var linq3 = (from c in \_session.Linq<Contact>()

where c.Id == id

select c).FirstOrDefault();

Je peux constater que NHibernate a généré directement les 2 requêtes SQL suivantes :

SELECT top 1

this\_.Id as Id1\_0\_,

this\_.FirstName as FirstName1\_0\_,

this\_.LastName as LastName1\_0\_,

this\_.Phone as Phone1\_0\_,

this\_.Email as Email1\_0\_,

this\_.groupId as groupId1\_0\_

FROM Contacts this\_

WHERE this\_.Id = @p0; // @p0 = 13

SELECT group0\_.Id as Id0\_0\_,

group0\_.Name as Name0\_0\_

FROM Groups group0\_

WHERE group0\_.Id=@p0; // @p0 = 1

Par acquit de conscience j’ai re-vérifié mon premier essai sans lazy="false". Pas de problème : NHibernate ne génère bien que la 1° requête SQL si je me contente de faire la requête LINQ et que je ne cherche pas à accéder à l’objet Group.

**Premier bilan**

Au moins, ça commence à être un peu plus clair pour ce côté de la relation. Si tout va bien, je ferais plus tard quelques essais supplémentaires pour étudier comment ça se passe dans le cas où je pars d’un groupe et que j’accède ensuite à ses contacts.

J’ai pas trainé : la suite de mes [essais avec LINQ to NHibernate](https://blog.pagesd.info/2010/02/19/suite-essais-linq-to-nhibernate/) !

[**Afficher le SQL généré par NHibernate**](https://blog.pagesd.info/2010/02/19/afficher-sql-genere-par-nhibernate/)

2010-02-19 #nhibernate

**Problème**

Quand on débute avec un ORM (et avec NHibernate en l’occurrence), il n’est pas très très simple de comprendre ce qui va se passer au niveau de la base de données quand on fait tel ou tel truc avec NHibernate et encore moins quand on s’essaie à faire des requêtes LINQ to NHibernate. Pour quelqu’un qui comme moi écrit du SQL depuis des années, l’idéal ce serait donc d’arriver à voir de mes propres yeux le code SQL que fabrique NHibernate.

Au début, j’ai donc eu la réaction classique et je me suis demandé pourquoi diable je ne parvenais pas à voir le source des requêtes SQL générées par NHibernate alors que j’avais pourtant bien initialisé la propriété “show\_sql” à “true” :

<property name="show\_sql">true</property>

Puis j’ai finalement remarqué que ça c’était bon pour quand je faisais des tests unitaires ou de petits essais sous forme d’applications consoles, mais que ça n’était pas prévu pour les applications web (et même pas du tout puisqu’il n’y avait rien).

Qu’à cela ne tienne, la réponse est dans Google. Mais là, le problème c’est qu’il y a beaucoup de réponses. Trop même. Et la plupart du temps il y a surtout tout un tas d’explications sur le fonctionnement de NHibernate et de log4net, la façon dont NHibernate utilise log4net, les différentes manipulations et configurations à faire pour qu’ils travaillent ensemble dans de bonnes conditions, comment sélectionner ce qu’on veut loguer…

Enfin beaucoup trop de trucs à assimiler quand comme moi on veut juste savoir à quoi ressemble le code SQL généré par NHibernate et qu’on n’a pas le temps et les moyens de s’intéresser aux coulisses.

Mais comme je suis ~~persistant~~ persévérant, j’ai quand même réussi à trouver ~~la~~ une solution et même qu’après un ou deux essais j’ai réussi à isoler ce qu’il était nécessaire et indispensable de faire pour que cela fonctionne de façon satisfaisante :

* il faut bien utiliser [log4net](http://logging.apache.org/log4net/)
* il n’y a pas énormément de trucs à faire

**Solution**

La première phase pour utiliser log4net nécessite de le configurer au niveau du fichier Web.config. Il faut commencer par ajouter la ligne suivante à l’intérieur de la section <configSections> afin de déclarer le nom de la section de configuration qui sera dédiée à log4net :

<section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />

Ensuite, il faut ajouter cette section <log4net> (juste après la section <configSections>) dans le Web.config pour configurer réellement log4net. Comme pour l’instant mon seul objectif est de pouvoir afficher les requêtes SQL au fur et à mesure que NHibernate les génère, je peux me contenter d’insérer les quelques lignes suivantes :

<log4net>

<appender name="TraceAppender" type="log4net.Appender.TraceAppender">

<layout type="log4net.Layout.PatternLayout">

<conversionPattern value="%message%newline" />

</layout>

</appender>

<logger name="NHibernate.SQL" additivity="false">

<level value="ALL"/>

<appender-ref ref="TraceAppender" />

</logger>

</log4net>

En gros, ça sert à dire que NHibernate doit loguer les requêtes SQL qu’il génère et que log4net doit afficher ces logs dans la fenêtre de débugage de Visual Studio (c’est peut être pas exactement ça, mais le résultat sera bientôt là).

Une fois cette première phase de configuration terminée, il reste à activer log4net. Pour cela, il faut tout d’abord ajouter la librairie log4net.dll dans les références du projet. Ceci fait, il reste à “démarrer” log4net, ou plus exactement à lui indiquer qu’il doit se paramétrer en utilisant la configuration définie dans le fichier Web.config.

Pour ça, il suffit d’ajouter une ligne log4net.Config.XmlConfigurator.Configure(); au niveau de la procédure Application\_Start dans le Global.axax.cs (et peut-être un using log4net; en début de fichier) :

using log4net;

...

protected void Application\_Start()

{

log4net.Config.XmlConfigurator.Configure();

RegisterRoutes(RouteTable.Routes);

}

Et c’est tout ! En fait, on n’a même pas besoin d’avoir initialisé la propriété “show\_sql” à “true” pour que ça marche :

