**Título**: Introdução ao NHibernate – Framework para Mapeamento Objeto-Relacional

**Descrição**: Neste artigo teremos um apanhado das principais funções do NHibernate. Passaremos pela conexão com o mesmo, como funciona o seu mapeamento, como é feita a operação de persistência.

Utilizaremos também padrões de Projeto para persistência de dados de forma a auxiliar melhor o leitor em sua modelagem demonstrando o poder deste framework.

**Autor**: gabrielsimas

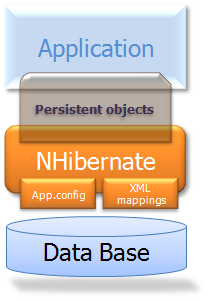


Neste artigo, teremos uma introdução ao NHibernate que é um framework de ORM portado para a plataforma .NET originário do framework Hibernate, um framework de bastante sucesso e utilizado até então, para a linguagem Java. Diante deste port, o NHibernate ainda é um projeto novo em constante evolução, tanto que ainda não acompanha seu “irmão” mais famoso com as grandes facilidades e opções de mapeamento mais rápido, principalmente com o uso de anotações, que em .NET são chamados de atributos – São interfaces que são utilizadas para “decorar” Classes, um substituo dos arquivos XML, tão odiados pelos programadores –que facilitam muito o dia-a-dia.

Tal framework foi criado pela JBoss, criadora do Hibernate, porém, com o tempo foi criada a NHForge que cuida diretamente do controle deste framework em site diferenciado, o NHForge.org.

**Estrutura do NHibernate**

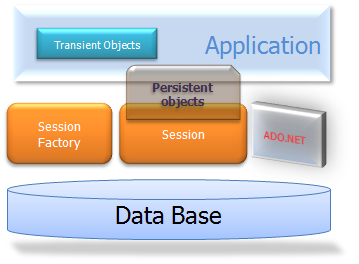
Vemos a estrutura básica do NHibernate na Figura 1.



**Figura 1. Diagrama arquitetônico do NHibernate**

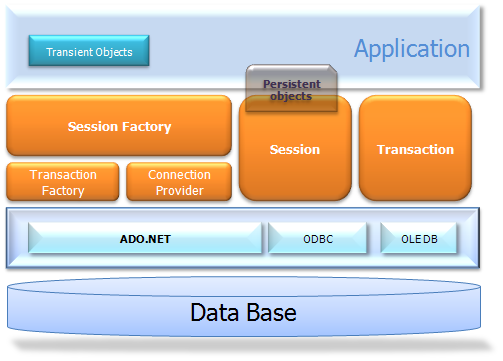
Na Figura 1, podemos observar que o NHibernate é acoplado diretamente entre O Banco de Dados a sua aplicação, porém, podemos observar também que os objetos persistentes – persistent objects – são utilizados através da aplicação e, através de conexão com o NHibernate, nos dá a entender que são as peças mais importantes desta engrenagem. Podemos também observar que dentro do domínio do NHibernate vemos o mapeamento em arquivos XML – Seção XML Mappings e o arquivo de configuração do .NET, App.config ou Web.config. Fique certo de que o NHibernate, como uma Camada de Acesso aos Dados e, posteriormente de persistência, de sua aplicação, estão fortemente acoplados porque todas as outras camadas, caso existam, são totalmente dependentes destes objetos de persistência para E/S de dados. Seria muito interessante mostrar uma visão mais detalhada da arquitetura em tempo de execução do NHibernate, porém isso não é possível pelas formas diferentes de abordagens e configurações, destas separamos duas: As arquiteturas simples e detalhada.

Na sua arquitetura simples, faz com que a aplicação disponibilize suas próprias conexões utilizando o ADO.NET e gerencie suas próprias transações. Esta abordagem usa um pequeno conjunto de APIs do NHibernate, como podemos verificar na figura 2.



**Figura 2. Subconjunto de APIs do NHibernate**

Na arquitetura Detalhada, existe uma abstração da aplicação totalmente diferente do uso das APIs do ADO.NET e deixa o NHibernate controlar e cuidar dos detalhes, permitindo assim que o mesmo cumpra o seu papel de Framework ORM, estes objetos dos quais o NHibernate faz uso internamente mostrados na Figura 3 e posteriormente explicados um a um.



**Figura 3. Objetos presentes na Arquitetura Detalhada do NHibernate**

Abaixo, listamos alguns dos objetos mais importantes presentes na figura 3:

***ISession (NHibernate.ISession)***

Um objeto single-threaded, ou seja, exclusivo por instância, e de curta duração que representa a conversa entre a aplicação e a unidade de persistência. Encapsula uma conexão ADO.NET. É uma factory para *ITransaction*. Mantém um cache de primeiro nível, mandatório, ou seja, sob demanda, de objetos persistentes. Este cache é utilizado quando o grafo de objetos é utilizado ou busca de objetos por identificador.

***Persistent Objects and Collections***

São objetos contendo estados de persistência e funções de negócio, tais objetos são sempre de curta duração e single-threaded. Estes podem ser Classes POCO, o que mais importa sobre estes é que são sempre associados com uma, somente uma ISession. Assim que uma ISession é finalizada, eles são desconectados e liberados para serem utilizados por quaisquer camadas na aplicação, isto quer dizer que pode ser utilizados como um DTO para transferência de dados de/para a Camada de Apresentação.

***Transient Objects and Collections***

Objetos Transientes (de curta duração, não persistidos em Banco de Dados) e Coleções

são instâncias de Classes que não estão atualmente associados com uma ISession. Eles podem ser instanciados ou não pela aplicação e sendo assim não foram ainda persistidos ou podem ser instanciados por uma ISession fechada.

***ITransaction (NHibernate.ITransaction)***

Esta Interface é opcional. É um objeto utilizado pela aplicação para especificar unidades de trabalho atômicas ou não. Aplica abstração de uma transação encapsulada do ADO.NET. Um ISession pode se utilizar de várias ITransaction em alguns casos.

***IConnectionProvider (NHibernate.Connection.IConnectionProvider)***

O Uso desta interface é opcional. Uma Factory para conexões e comandos ADO.NET. Aplica abstração de uma implementação específica e concreta de IDbConnection e IDbCommand. Não é exposto para a aplicação, mas, pode ser extendido/implementado pelo desenvolvedor.

***IDriver (NHibernate.Driver.IDriver)***

Uma interface que encapsula provedores ADO.NET diferentes, tais como convenções para nomes de parâmetros e convenções suportadas para uso com o ADO.NET. Seu uso é opcional.

***ITransactionFactory (NHibernate.Transaction.ITransactionFactory)***

Uma Factory para instancia de ITransaction. Não é exposta a aplicação, mas pode ser extendida/implementada pelo desenvolvedor. Seu uso é opcional.

Ao usar a arquitetura simples, a aplicação não utiliza nem ITransaction/ITransactionFactory e muito menos IConnectionProvider para utilizar o ADO.NET diretamente.

Neste artigo não utilizaremos todos estes objetos.

**Classes POCO para uso com o NHibernate**

Existem 4 regras principais para Classes POCO serem “reconhecidas” pelo NHibernate:

1. Use propriedades para campos persistentes.
2. Implemente um construtor padrão, ou seja, vazio.
3. Crie uma propriedade Identificadora, ou Id.
4. Utilize classes que não sejam sealed e use métodos virtuais.

**Mapeamento do NHibernate**

Atualmente, o mapeamento no NHibernate pode ser executado utilizando um arquivo XML ou com um novo projeto onde são utilizados atributos para decorarem as classes, primeiramente, vamos ao mapeamento utilizando o XML.

**Mapeamento por Arquivo XML**

O Mapeamento XML é a primeira etapa para a persistência com o NHibernate é onde tudo começa, na listagem 1, temos um exemplo de cabeçalho estável para uso com o NHibernate. Por motivos de compilação, o NHibernate solicita que os arquivos para mapeamento contenham também a extensão .hbm como forma de ser reconhecido pelo assembly do ORM. Estes arquivos precisam ter sua propriedade *Build Action* setados como *Embedded Resource*, posto que no momento da compilação do código, o arquivo seja reconhecido e convertido para um hierarquia de classes utilizando ponto, por exemplo: temos que mapear a Classe Pessoa para persistência, então criamos o arquivo *pessoa.hbm.xml* e nas suas propriedades, colocamos o atributo Build Action como Embedded Resource, como mostrado na Figura 3.

**Listagem 1. Cabeçalho de XML para mapeamento**

[CODE]

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

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

                   assembly="DAL"

                   namespace="DAL.Entidade">

...

...

...

....

...

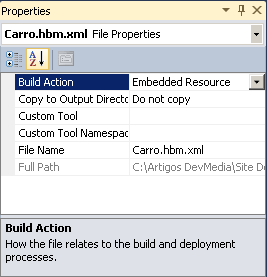
</hibernate-mapping>

[/CODE]

Como podemos ver na listagem 1, temos 2 atributos na tag *hibernate-mapping* que são *assembly* e *namespace*.

*Assembly* informa em qual Projeto estão localizadas as Classes a serem buscadas pelo NHibernate no qual está sendo apontada neste arquivo XML.

*Namespace* é o caminho completo de onde tais classes estão dentro de um Projeto. Lembrando que quanto mais informação neste sentido é menos tempo em que o compilador perde buscando as Classes para Mapeamento em memória.



**Figura 4. Arquivo XML configuração e nomenclatura padrão NHibernate**

Neste ponto, já podemos mapear as nossas Classes de Persistência. Não é obrigatório criar um arquivo para cada Classe, podemos em um arquivo só inserir todas, mas, como melhor prática, é bom que seja criado um arquivo para cada Classe, desta forma, o entendimento do Sistema fica melhor e separa bem as responsabilidades. Na Listagem 2, temos um exemplo de Entidade na qual vamos mapear via XML.

**Listagem 2. Classe Carro**

[CODE]

public class Carro

    {

        #region atributos

        private Int32 id;

        private String modelo;

        private String motor;

        #endregion

        #region Propriedades

        public virtual Int32 Id

        {

            get { return this.id; }

            set { this.id = value; }

        }

        public virtual String Modelo

        {

            get { return this.modelo; }

            set { this.modelo = value; }

        }

        public virtual String Motor

        {

            get { return this.motor; }

            set { this.motor = value; }

        }

        #endregion

        #region Construtores

        public Carro()

        {

        }

        public Carro(String modelo, String motor)

        {

            this.Modelo = modelo;

            this.Motor = motor;

        }

        #endregion

        #region Sobrescritas

        public override string ToString()

        {

            return this.id + ", " + this.modelo + ", " + this.motor;

        }

        #endregion

    }

}

[/CODE]

Para o correto mapeamento da Entidade, mapeamos apenas as Propriedades para o arquivo XML, como descrito na Listagem 3.

**Listagem 3. Arquivo XML Carro.hbm.xml**

[CODE]

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

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

                   assembly="DAL"

                   namespace="DAL.Entidade">

  <class name="Carro" table="Carro">

    <id name="Id" column="id">

      <generator class="identity" />

    </id>

    <property name="Modelo" column="modelo" />

    <property name="Motor" column="motor" />

  </class>

</hibernate-mapping>

[/CODE]

Na Listagem 3, temos algumas informações importantes, a tag *id* significa que estamos mapeando a propriedade de identificação única da Entidade, ou seja, valor exclusivo que identifica os dados gravados na Classe. Dentro desta tag, temos uma tag *generator* que como no Banco de Dados ela é um tipo de auto incremento, utilizamos o valor de atributo *identity*. Os valores possíveis e permitidos pela tag *generator* dentro do atributo *class* são:

increment - Gera identificadores de qualquer tipo desde que sejam únicos e que nenhum outro processo esteja inserindo dados na mesma tabela. Não use este valor com um Cluster.

identity - Suporta campos do tipo identity, ou seja, auto incremento nos Bancos de Dados DB2, MySQL, MS SQL Server e Sybase. O identificador que é devolvido pelo Banco de Dados é convertido para o tipo da propriedade usando Convert.ChangeType. Qualquer propriedade inteira é permitida.

sequence - Usa uma Sequence criada em DB2, PostgreSQL, Oracle ou um generator do Firebird. O identificador retornado pelo Banco de Dados é convertido para o tipo da propriedade usando Convert.ChangeType. Qualquer propriedade inteira é permitida.

hilo - Usa um algoritmo de hi/lo para gerar de forma bem eficiente qualquer tipo inteiro de qualquer Tabela ou campo (por padrão, hibernate\_unique\_key e next\_hi respectivamente) como uma fonte de valores hi. O Algoritmo de hi/lo gera identificadores que são únicos apenas para um Banco de Dados em específico. Não use este hilo com uma conexão gerenciada pelo usuário, ou seja, quando as threads não são as mesmas.

Você pode usar o parâmetro "where" para especificar um registro usado em uma tabela. Isto é muito útil se você quer usar uma tabela única para seus identificadores, com registros diferentes de cada tabela.

***seqhilo*** - Usa um algoritmo de hi/lo para gerar de forma bem eficiente qualquer tipo inteiro, dada uma SEQUENCE criada no Banco de Dados.

***uuid.hex*** - Usa System.Guid e seu método ToString para gerar identificadores do tipo String. O tamanho da String depende do formato selecionado.

***uuid.string*** - Usa um novo System.Guid para criar um tipo byte[] que é convertido para String.

***guid*** - Usa um novo System.Guid como identificador.

***guid.comb*** - Usa o algoritmo para gerar um novo System.Guid.

***native*** - captura tipos identity, sequence ou hilo dependendo das capacidade existentes no Banco de Dados.

***assigned*** - Deixa a aplicação decidir por um identificador para o objeto antes de Save() ser chamado.

***foreign*** - Usa o identificador de outro objeto associado. Normalmente usado em conjunção com uma associação com chave primária com a tag <one-to-one>

A Tag *property*, não é necessária muita informação, são as propriedades da Classe Carro que refletem os campos da Tabela.

**Mapeamento por Anotação/Atributo**

Todos nós já sabemos que o NHibernate precisa de mapeamentos efetuados através de arquivos XML para que possa colocar em memória o ORM para a persistência. Para facilitar este trabalho foi criado um Projeto desenvolvido inicialmente por John Morris e que depois passou às mãos de Pierre Henri Kuaté, mais conhecido como KPixel. Este projeto decora as Classes com atributos o que torna o arquivo XML desnecessário posto que o Projeto gera um Stream para a memória como o XML faria, este projeto se chamado NHibernate Attributes que utiliza o assmbly NHibernate.Mapping.Attribute. Um exemplo de Classe mapeada com o assembly NHibernate.Mapping.Attributes na Listagem 4.

**Listagem 4. Classe “decorada” com atributos NHibernate**

    [Serializable]

    [Class(Schema = "RevistaArtigo02", Table = "Carro")]

    public class Carro

    {

        #region atributos

        private Int32 id;

        private String modelo;

        private String motor;

        #endregion

        #region Propriedades

        [Id(Name="Id",Column="id"),Generator(1,Class="identity")]

        public virtual Int32 Id

        {

            get { return this.id; }

            set { this.id = value; }

        }

        [Property(Column="modelo")]

        public virtual String Modelo

        {

            get { return this.modelo; }

            set { this.modelo = value; }

        }

        [Property(Column="motor")]

        public virtual String Motor

        {

            get { return this.motor; }

            set { this.motor = value; }

        }

        #endregion

        #region Construtores

        public Carro()

        {

        }

        public Carro(String modelo, String motor)

        {

            this.Modelo = modelo;

            this.Motor = motor;

        }

        #endregion

        #region Sobrescritas

        public override string ToString()

        {

            return this.id + ", " + this.modelo + ", " + this.motor;

        }

        #endregion

    }

}

**Arquivo de Configuração do NHibernate**

O Nhibernate é um framework bastante versátil no qual permite que o configuremos via Código ou via arquivo XML, neste artigo vamos abordar o mapeamento via XML, para maiores informações sobre a inicialização do NHibernate via código,e só acessar o site <http://nhforge.org/doc/nh/en/index.html#session-configuration>. O Nhibernate precisa ser inicializado através da interface *ISessionFactory*, com ela, todas as Entidades são compiladas e colocadas na memória, assim como os comandos deste arquivo XML são aceitos e colocados em memória aguardando sua execução. Como padrão, o NHibernate aceita como arquivo de configuração contendo a extensão *.cfg.xml*, normalmente cria-se um arquivo na raiz do Projeto chamada *hibernate.cfg.xml*. Neste arquivo, registramos inclusive a string de conexão, dialeto envolvido e outras configurações de ambiente. Na Listagem 6, vemos um exemplo deste arquivo.

**Listagem 6. Arquivo de configuração hibernate.cfg.xml**

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

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

  <session-factory>

    <property name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>

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

      <!-- Aqui você deve coloca o seu Connectio String -->

      <property name="connection.connection\_string">

        Data Source=ALLSPARK\SQLEXPRESS;Initial Catalog=RevistaArtigo02;Integrated Security=True

      </property>

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

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

    <property name="current\_session\_context\_class">thread</property>

  </session-factory>

</hibernate-configuration>

Diante desta informação, já podemos instanciar a nossa *ISessionFactory*, que funciona como uma fachada. O mais importante é que devemos sempre prezar pela performance, é necessário que criemos uma Classe que gere um Singleton para que o NHibernate possa ser carregado e controlado de forma independente e desacoplada do código, na Listagem 6, temos um exemplo de Classe que contém este padrão de Projeto. Esta Classe acessa o arquivo de configuração e faz com que toda a “mágica” do que fizemos até agora aconteça e que faça a conexão com o Banco de Dados

**Listagem 6. Classe NHibernateUtil**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using NHibernate;

using NHibernate.Cfg;

namespace DAL.Reuso

{

    public sealed class NHibernateUtil

    {

        #region Atributos

        private static readonly ISessionFactory fabricaDeSessao;

        private const String CodSessaoAtual = "nhibernate.current\_session";

        #endregion

        #region Construtor

        static NHibernateUtil()

        {

            try

            {

                ISessionFactory fabrica = new Configuration().Configure().AddAssembly("DAL").BuildSessionFactory();

            }

            catch (Exception)

            {

                throw;

            }

        }

        #endregion

        #region Métodos de Controle

        public static ISession PegaSessaoEmUso()

        {

            ISession sessao = fabricaDeSessao.GetCurrentSession();

            if (sessao == null)

            {

                sessao = fabricaDeSessao.OpenSession();

            }

            return sessao;

        }

        public static void FecharSessao()

        {

            ISession sessao = fabricaDeSessao.GetCurrentSession();

            if (sessao != null)

            {

                sessao.Close();

            }

        }

        public static void FecharFabricaDeSessao()

        {

            if (fabricaDeSessao != null || !fabricaDeSessao.IsClosed){

                fabricaDeSessao.Close();

            }

        }

        #endregion

    }

}

Esta Classe, o leitor pode já utilizá-la como está em seus projetos e até aprimorá-la mais para que fique mais genérica a seu uso, assim como em futuros artigos que faremos de forma mais prática demonstrando o uso de Generics, Fachadas e outros padrões de projeto que facilitem a implementação nos projetos de nossos leitores. Já criamos uma Classe que irá e muito facilitar o nosso trabalho de acesso a base de dados, diante disto, vamos mostrar a melhor forma de utilizá-la.

**Acessando a Base de Dados através do NHibernate**

Criamos na Listagem 6 uma Classe para acesso a Base de Dados, agora, veremos como que NHibernate se comunica com a Base de Dados para permitir que a persistência seja efetuada.

Inicialmente, como uma melhor prática, o NHibernate trabalha com transações, esta é uma forma bem segura de trabalho, posto que caso algo dê errado, um Rollback pode ser dado, caso contrário, um commit grava as alterações no Banco de Dados e a conexão é finalizada, salvo para operações de busca, onde a sessão continua em aberto. Devemos ter isso me mente até mesmo quando o projeto solicitar que utilizemos mais de uma Classe em uma transação. O Nhibernate abre uma transação, efetua o trabalho que precisa fazer, trata este trabalho, caso execute com sucesso, executa um commit, caso não executa um Rollback e posterimente fecha a sessão. Uma Sessão é composta de Transação, execução e Fechamento, o fechamento pode ser um *finally* com um comado de fechamento ou simplesmente uma execução de operações, neste caso, por exemplo, seria um INSERT, UPDATE e DELETE suficientes para fechar a Sessão o que também não seria necessário que a fechássemos, um fragmento deste trabalho, pode ser visto na Listagem 7 já com o uso de nossa Classe NHibernateUtil.

**Listagem 7.NHibernate interagindo com o Banco de Dados**

        public void Gravar(Carro carro)

        {

            ISession sessao = NHibernateUtil.PegaSessaoEmUso();

            ITransaction transacao = sessao.BeginTransaction();

            sessao.Save(carro);

            transacao.Commit();

            NHibernateUtil.FecharSessao();

        }

    }

Como sabemos, o Banco de Dados isola as operações através das transações, diante disso, temos um singleton abrindo e fechando a sessão de forma protegida.

Agora, é só criar um Projeto e implementar a solução. E isso é assunto para um outro artigo.

**Conclusão**

O NHibernate é mais uma opção frente aos outros frameworks ORM existentes no mercado, ele não perde em nada com relação ao Microsoft ADO.NET Entity Framework e outros. Vimos neste artigo o poder deste framework que já faz bastante sucesso no meio Java e que pode também fazer bastante sucesso no mundo .NET. Vimos apenas 20% de todo o framework e com esta informação, já podemos trabalhar em um Projeto de médio porte. Em artigos próximos, vamos falar de outras partes do NHibernate. Este por ser um framework um tanto extenso se fizéssemos aqui a cobertura deste, com certeza ficaria cansativo ao leitor.

Forte Abraço e até a próxima, deixando bem claro que é um grande prazer compartilhar informação.