02

Configurando e Entendendo o NHibernate

O que é uma ferramenta de mapeamento objeto-relacional?

Todos nós, desenvolvedores de software, sabemos da importância de banco de dados em nossos sistemas. Necessitamos, a todo instante, realizar consultas aos dados para atender as solicitações do usuário, bem como, registrar qualquer informação passada por este.

Atualmente possuímos muitos bancos de dados relacionais, dentre os mais conhecidas podemos citar: Oracle, MS SQL Server, MySql, etc. Nessas ferramentas, as informações são armazenadas como registros de tabelas.

Por exemplo, imagine o sistema para gerenciamento de produtos em uma loja virtual. O banco de dados para armazenar os produtos disponíveis para venda teria uma tabela chamada `Produtos`, composta por campos que caracterizam um produto, tais como: Código, Nome, Preço, Categoria, etc.

Sem a utilização de alguma ferramenta, o trabalho do programador seria árduo e muito vulnerável a falhas. Imagine o mapeamento de uma tabela Usuários criada através do seguinte comando SQL:

```
create table Users (
    id int UNSIGNED NOT NULL AUTO_INCREMENT,
    email varchar (255) NOT NULL,
    password varchar (20) NOT NULL,
    name varchar (255) NOT NULL,
    active bit (1),
    date datetime NOT NULL,
    PRIMARY KEY(id)
)
```

Em nosso sistema temos a necessidade de listar todos os usuários cadastrados no banco. Quando queremos acessar o banco de dados, precisamos de uma conexão:

```
SqlConnection conexao = new SqlConnection("String de conexão com o MS SQL");
conexao.Open();
```

Depois de abrir a conexão, precisamos preparar um comando para o banco de dados que enviará um `SELECT` que devolverá a lista de usuários:

```
SqlCommand comando = new SqlCommand(
    "SELECT id, email, password, name, active, date FROM Users",
     conexao);
comando.CommandType = System.Data.CommandType.Text;
```

O comando preparado é executado através do método `ExecuteReader`, esse método devolve um objeto especializado em ler o resultado da busca:

```
IDataReader reader = comando.ExecuteReader();
```

Agora que temos o leitor do resultado, precisamos transformá-lo em uma lista de modelos da aplicação. O `IDataReader` possui o método `Read` que é responsável por ler os resultados devolvidos pela busca.

Cada vez que chamamos o `Read`, estamos avançando para o próximo registro da busca. Enquanto for possível avançar, o `Read` devolve o valor `true`, o `false` é devolvido quando não há mais registros na busca. O código para ler todos os registros fica da seguinte forma:

```
while (reader.Read())
{
    // lê o registro
}
```

Podemos acessar as informações do registro atual utilizando a notação de array, da mesma forma que fazemos com um dicionário do C#, porém os tipos dos dados devolvidos pelo `reader` são tipos do banco de dados e, portanto, precisamos convertê-los para os tipos do c#. Esse trabalho será feito pela classe `Convert`:

```
while(reader.Read())
{
    int id = Convert.ToInt32(reader["id"]);
}
```

Os dados que estamos lendo pertencem ao modelo `Usuario`, então vamos guardá-los em uma instância dessa classe

```
while(reader.Read())
{
    Usuario usuario = new Usuario();
    usuario.Id = Convert.ToInt32(reader["id"]);
    usuario.Email = Convert.ToString(reader["email"]);
    usuario.Senha = Convert.ToString(reader["pasword"]);
    usuario.Nome = Convert.ToString(reader["name"]);
    usuario.Ativo = Convert.ToBoolean(reader["active"]);
    usuario.DataCadastro = Convert.ToDateTime(reader["date"]);
}
```

O loop lê o registro, cria o usuário com os dados preenchidos e imediatamente perde a referência para o usuário criado. Para não perdemos a referência, vamos guardá-la em uma lista:

```
IList<Usuario> usuarios = new List<Usuario>();
while(reader.Read())
{
    Usuario usuario = new Usuario();
    usuario.Id = Convert.ToInt32(reader["id"]);
    usuario.Email = Convert.ToString(reader["email"]);
    usuario.Senha = Convert.ToString(reader["pasword"]);
    usuario.Nome = Convert.ToString(reader["name"]);
    usuario.Ativo = Convert.ToBoolean(reader["active"]);
    usuario.DataCadastro = Convert.ToDateTime(reader["date"]);
    usuarios.Add(Usuario);
}
```

Agora que terminamos de fazer a query, precisamos fechar o reader e a conexão.

```
reader.Close();
conexao.Close();
```

O código para acessar o banco de dados é trabalhoso e repetitivo, além disso, quando fazemos queries que envolvem valores passados pelo usuário, podemos facilmente inserir vulnerabilidades que permitem ataques como o `SQL Injection` (Técnica utilizada por hackers para enviar comandos nocivos à base de dados, através de campos do formulário ou URLs, por exemplo).

Além do código repetitivo e dos problemas de segurança, o que acontece quando precisamos trocar o banco MySQL, utilizado atualmente, para o Oracle? Nesse caso, teremos que modificar o código do sistema inteiro, além das SQLs, para poder suportar o novo banco.

Esses são apenas alguns dos problemas que temos quando lidamos com o banco de dados diretamente, mas reparem que para construirmos uma query na tabela de usuários, precisamos apenas olhar a estrutura da classe `Usuario` , ou seja, podemos ter uma ferramenta que dada uma classe, consegue construir as queries necessárias. Ferramentas que fazem o mapeamento do mundo orientado a objetos (classes e objetos) para o mundo relacional são chamadas de mapeadores objeto relacional, ou ORM (Object Relational Mapper). Nesse mapeamento, classes se transformam em tabelas e objetos em registros das tabelas.

Nesse curso, utilizaremos o NHibernate, uma das ferramentas ORM mais utilizadas no mercado.

NHibernate

NHibernate é um framework de persistência de dados, desenvolvido para a plataforma .NET, que encapsula toda a complexidade do acesso ao banco de dados relacional, facilitando o mapeamento de tabelas em objetos e a construção de comandos SQL, que na maior parte dos casos são criados automaticamente. Ele possui o código fonte aberto e foi desenvolvido com base no Hibernate do Java. É comum pesquisarmos determinado recurso do framework e encontrarmos fontes relacionadas para Hibernate. Geralmente a mesma implementação funciona para ambos. A versão atual é a versão 3.3.1 e pode ser baixada no site oficial do NHibernate, (http://nhforge.org/)http://nhforge.org/ (http://nhforge.org/).

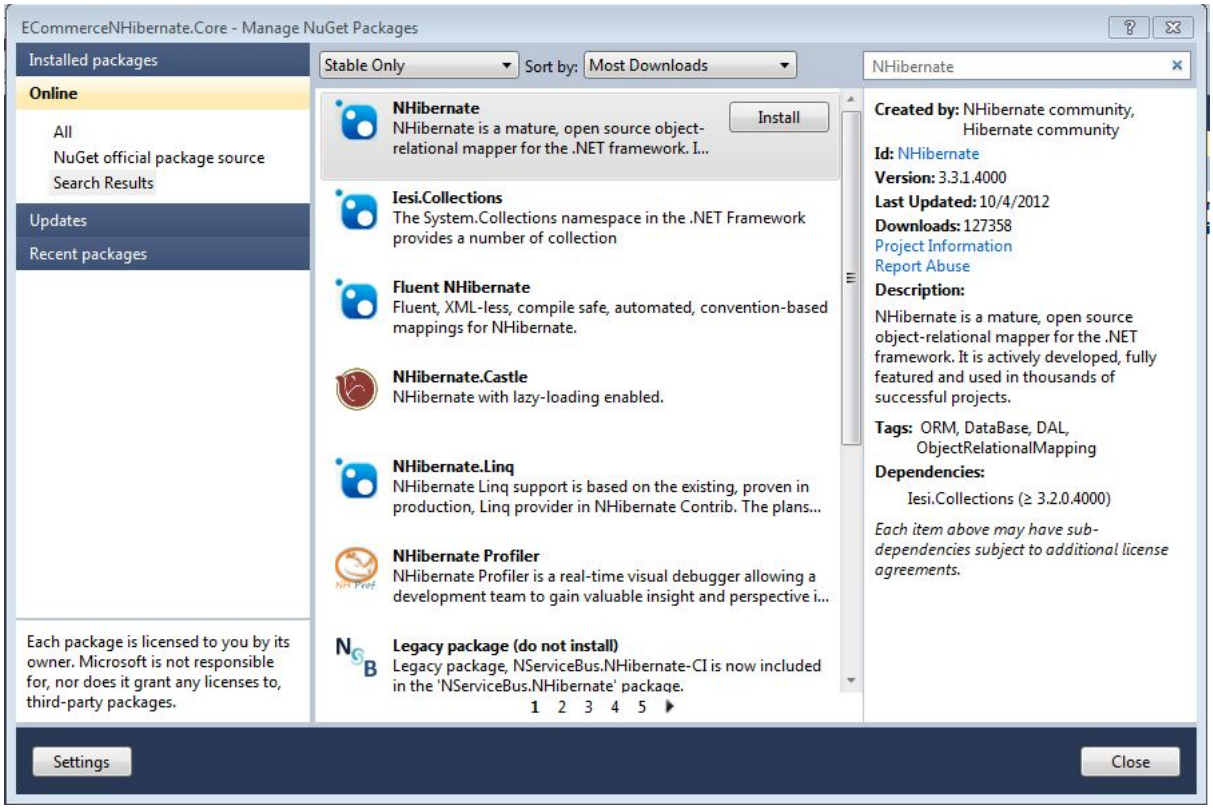
No site oficial encontramos também toda a documentação do framework, bem como exemplos de implementação e fórum de discussões.

Referenciando o NHibernate

Para trabalharmos com o NHibernate, precisamos baixar a última versão estável do framework (hoje é a versão 3.3.1). O download pode ser feito diretamente no site oficial, ou através de uma extensão do Visual Studio chamada Nuget (http://nuget.org/ (http://nuget.org/))

Para exemplificarmos o uso do NHibernate, criaremos uma loja virtual simplificada, em um projeto chamado Loja. Nesse projeto, criaremos classes que possuem representação no banco de dados, chamaremos essas classes de Entidades. Vamos criar um projeto do tipo `Console Application` , que chamaremos de Loja

Dentro do Visual Studio, clique com o botão direito no projeto Loja e escolha a opção `Manage Nuget Packages` . Uma janela como a da imagem abaixo será aberta. Na aba `Online Packages` , busque por NHibernate e após encontrar clique no botão Install.



O próprio Nuget se encarrega de efetuar o download do framework e já referenciar as dlls necessárias ao seu projeto. Pronto! Você já pode utilizar o NHibernate em seu projeto.

Mapeando a primeira classe

Agora que temos o NHibernate instalado no projeto, podemos escrever a primeira entidade, a classe `Usuario`. Ela será uma entidade do domínio da aplicação, que ficará dentro do namespace `Loja.Entidades`

O usuário terá um Id, que será o identificador único do usuário, e um Nome. No NHibernate, as propriedades de uma entidade devem ser marcados como `virtual`, portanto o `Usuario` fica da seguinte forma:

```
public class Usuario
{
    public virtual int Id { get; set; }

    public virtual string Nome { get; set; }
}
```

Quando utilizamos o NHibernate, não precisamos mais nos preocupar com o banco de dados. Essa responsabilidade agora é do NHibernate, nossa preocupação será apenas com o domínio da aplicação.

Agora que a entidade foi criada, precisamos dizer ao NHibernate como ela deve ser mapeada para uma tabela no banco de dados. Essa configuração é feita em um arquivo xml.

Para mantermos a organização do projeto, colocaremos os xmls de mapeamento do NHibernate dentro da pasta `Mapeamentos`. Os arquivos xml que tem o nome terminado com `.hbm.xml` são automaticamente reconhecidos pelo NHibernate como arquivos de mapeamento, logo dentro da pasta `Mapeamentos` criaremos o arquivo `Usuario.hbm.xml`.

Vamos começar informando ao NHibernate, qual entidade estamos mapeando, em nosso caso a entidade `Usuario`. Dizemos isso através da tag `class` e do atributo name dessa tag.

```
<class name="Usuario">

</class>
```

Em um banco de dados relacional, toda tabela deve possuir um identificador único do registro. Na tabela `Users` apresentada no exemplo anterior, temos um campo id que é um inteiro auto incrementável, ou seja, a cada registro inserido na tabela o valor desse campo é automaticamente setado para o próximo número da sequência.

O mapeamento de uma chave primária é feito através do atributo `name` da tag `id`. Dentro dessa tag podemos definir qual será o mecanismo responsável pela geração dos ids através da tag `generator`. Para utilizarmos o auto increment, precisamos configurar o `identity` como mecanismo de geração de ids.

```
<id name="Id">
    <generator class="identity" />
</id>
```

Agora, precisamos informar como os atributos da classe Usuario devem ser mapeados. Esse mapeamento é feito através do atributo `name` da tag `property`. O mapeamento do `Nome` fica da seguinte forma:

```
<property name="Nome" />
```

O arquivo `Usuario.hbm.xml` deve ficar parecido com o código abaixo:

```
<class name="Usuario">
    <id name="Id">
        <generator class="identity" />
    </id>
    <property name="Nome" />
</class>
```

A raíz do xml de mapeamento de entidades deve ser a tag `hibernate-mapping`. Nessa tag dizemos ao NHibernate qual é o namespace da entidade, através do atributo `namespace`, e também o nome do projeto, através do atributo `assembly`:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
  assembly="Loja"
  namespace="Loja.Entidades">
    <!-- Mapeamentos -->
</hibernate-mapping>
```

O arquivo Usuario.hbm.xml fica da seguinte forma:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
  assembly="Loja"
  namespace="Loja.Entidades">
    <class name="Usuario">
        <id name="Id">
            <generator class="identity" />
```

```
        </id>
        <property name="Nome" />
    </class>
</hibernate-mapping>
```

Para que o NHibernate encontre o mapeamento, o xml deve ser colocado dentro do código compilado do programa (`Assembly` do programa), porém o Visual Studio, por padrão, só coloca o código C# no Assembly gerado. Para mudar o comportamento padrão da compilação (build) do Visual Studio, precisamos configurar o `Build Action` do arquivo.

Para fazer com que o Visual Studio coloque o xml dentro do assembly gerado, vamos clicar com o botão direito no nome do arquivo e selecionar a opção `Properties` . Dentro da janela properties, vamos mudar o valor da propriedade `Build Action` para `Embedded Resource` . Dessa forma, sempre que o projeto for compilado, o Visual Studio copiará o xml para dentro do Assembly.

Pronto nosso mapeamento está finalizado! Repare que em nenhum momento informamos o tipo dos campos. O próprio NHibernate se encarrega da definição do tipo do campo no banco de dados que atende ao tipo especificado para o atributo da classe.

Configurando o NHibernate

Já criamos e configuramos a entidade Usuario, agora é hora de usar o NHibernate para acessar o banco de dados!

Para o NHibernate se comunicar com o banco de dados, ele precisa de uma conexão. Quando abrimos a conexão com o banco de dados, precisamos informar qual é o nome do usuário, a senha, além de outras informações, todas essas configurações são passadas para o NHibernate através do arquivo `hibernate.cfg.xml`

Vamos criar um novo arquivo xml chamado `hibernate.cfg.xml` no projeto. Nesse xml, as configurações ficam dentro da tag `hibernate-configuration`

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <!-- Configurações ficam aqui -->
</hibernate-configuration>
```

As configurações de conexão ficam dentro de outra tag chamada `session-factory` :

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory>
    </session-factory>
</hibernate-configuration>
```

Dentro do hibernate.cfg.xml, precisamos definir qual será o driver utilizado para acessar o banco de dados

```
<property name="connection.driver_class">
    NHibernate.Driver.MySqlDataDriver
</property>
```

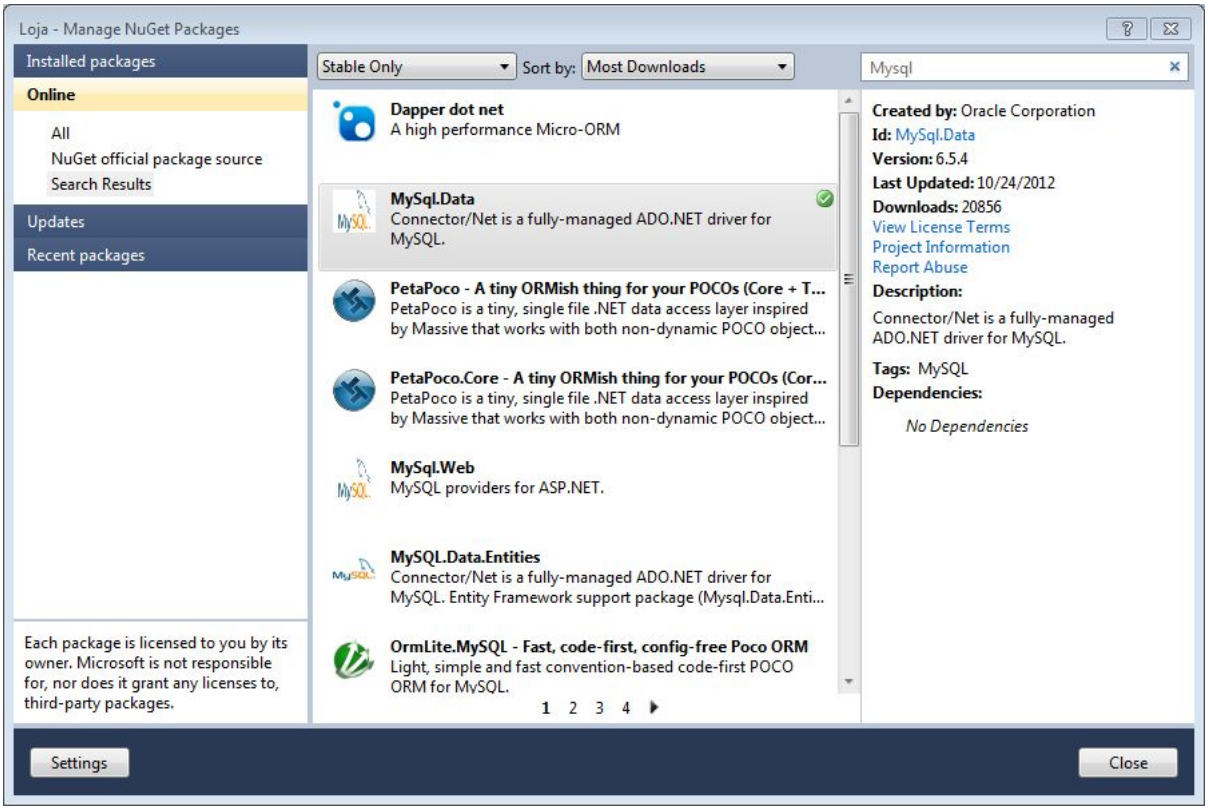
E também a classe que será responsável por abrir as conexões com o banco de dados.

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

Além disso, como cada banco de dados fala sua própria versão do SQL, precisamos definir qual o dialeto que o NHibernate utilizará para gerar as SQLs.

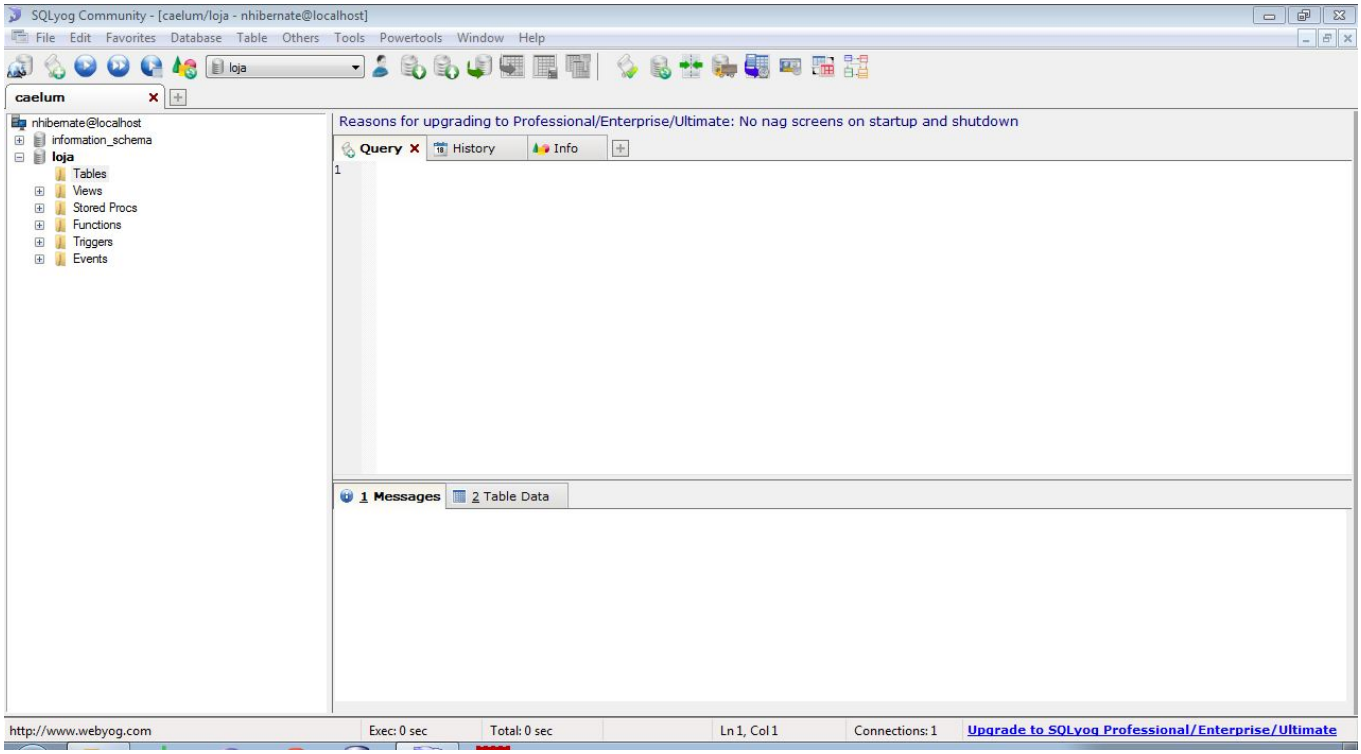
```
<property name="dialect">
    NHibernate.Dialect.MySQL5Dialect
</property>
```

Como utilizaremos o Mysql, precisamos instalar as dlls do driver desse banco. Da mesma forma que fizemos com o NHibernate, adicionaremos o driver do MySql através do Nuget:



Agora precisamos criar o banco de dados que o NHibernate utilizará. Podemos criar o banco diretamente pelo prompt de comando ou através de um programa de sua preferência. Uma boa ferramenta para Windows é o SqlYog.

Crie uma base de dados chamada loja e um usuário com acesso a essa base. Criei um usuário chamado `nhibernate` com senha `caelum`.



Além das informações sobre o driver, precisamos informar qual é o nome do usuário que será utilizado, sua senha, qual é o banco que queremos utilizar e, além disso, o ip do banco de dados. Essas configurações ficam dentro de uma string de conexão (connection_string).

```
<property name="connection.connection_string">
    Server=localhost;Database=loja; Uid=nhibernate; Pwd=caelum;
</property>
```

Para visualizarmos as queries que o hibernate enviará para o banco de dados, precisamos adicionar a seguinte linha:

```
<property name="show_sql">true</property>
```

O `hibernate.cfg.xml` fica da seguinte forma:

```
<?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="dialect">
            NHibernate.Dialect.MySQL5Dialect
        </property>
        <property name="connection.driver_class">
            NHibernate.Driver.MySqlDataDriver
        </property>
        <property name="connection.connection_string">
            Server=localhost;Database=loja; Uid=nhibernate; Pwd=caelum;
        </property>
        <property name="show_sql">true</property>
    </session-factory>
```

```
</hibernate-configuration>
```

O arquivo `hibernate.cfg.xml` deve ficar junto com o arquivo executável (Assembly) da aplicação, porém, por padrão, o Visual Studio não copia o arquivo de configuração na compilação do projeto, logo o NHibernate não será configurado. Para mudarmos esse comportamento padrão, devemos abrir o Solution Explorer, clicar com o botão direito no arquivo e escolher a opção Properties, dentro dessa janela, vamos mudar a configuração `Copy To Output Directory` para o valor `Copy Always`, com isso, toda vez que o projeto for compilado, o arquivo de configuração será copiado para a pasta do assembly da aplicação.

Agora vamos testar se a configuração do NHibernate está correta, para isso criaremos uma rotina para deixar que o NHibernate crie as tabelas a partir do mapeamento que fizemos. No nosso caso, ele deve criar uma tabela chamada `Usuario` com dois campos, `Id` e `Nome`. No projeto `Loja`, vamos criar uma pasta chamada %%Infra%% e dentro dela adicionar uma classe chamada `NHibernateHelper`.

O NHibernate possui uma classe chamada `NHibernate.Cfg.Configuration`, reponsável por encontrar os arquivos de mapeamento e configuração do NHibernate, parseá-los e nos fornecer os mecanismos necessários para o acesso aos dados. Para configurarmos o NHibernate a partir dessa classe, utilizamos o método `Configure`:

```
Configuration cfg = new Configuration();
cfg.Configure();
```

O `Configure()` busca hibernate.cfg.xml, para fazer a configuração do NHibernate. Precisamos agora informar ao NHibernate quais entidades do projeto devem ser mapeadas. Queremos que ele mapeie todas as entidades do projeto, o assembly atual:

```
cfg.AddAssembly(Assembly.GetExecutingAssembly());
```

Essa linha faz com que o NHibernate procure todos os arquivos `hbm.xml` do assembly atual e adicione as entidades mapeadas à configuração. Vamos criar no `NHibernateHelper` um método chamado `RecuperaConfiguracao` que devolve uma instância de `Configuration` com os mapeamentos adicionados:

```
public static Configuration RecuperaConfiguracao()
{
    Configuration cfg = new Configuration();
    cfg.Configure();
    cfg.AddAssembly(Assembly.GetExecutingAssembly());
    return cfg;
}
```

Agora que o NHibernate está configurado, vamos utilizá-lo para gerar as tabelas do banco de dados. Na classe `NHibernateHelper` vamos adicionar um método que criará as tabelas (schema) do banco de dados. Esse método utilizará o `RecuperaConfiguracao` criado acima e, através da classe `NHibernate.Tool.hbm2ddl.SchemaExport`, criará as tabelas do banco de dados.

```
public static void GeraSchema() {
    Configuration cfg = RecuperaConfiguracao();
    new SchemaExport(cfg).Create(true, true);
}
```

Vamos utilizar o GeraSchema no Main da classe Program:

```
static void Main(string[] args) {
    NHibernateHelper.GeraSchema();
}
```

Rodando o projeto vemos no console do programa as queries utilizadas pelo NHibernate para criar o schema de tabelas do banco de dados. Se tudo estiver correto você verá em sua database a tabela Usuario definida exatamente como configuramos no `Usuario.hbm.xml` .