02

Inserindo e Alterando Dados

Salvando uma nova entidade

Agora que o NHibernate está configurado, vamos aprender como gravar nossas entidades no banco de dados.

Quando queremos inserir dados no banco, precisamos abrir uma conexão. Com o NHibernate, utilizaremos um componente que faz o papel da conexão com o banco de dados chamado `NHibernate.ISession`.

Instâncias de `ISession` são contruídas por uma fábrica conhecida como `NHibernate.ISessionFactory`. Conseguimos uma instância dessa fábrica utilizando o método `BuildSessionFactory` da classe `Configuration` do NHibernate:

```
Configuration cfg = NHibernateHelper.RecuperaConfiguracao();
ISessionFactory sessionFactory = cfg.BuildSessionFactory();
```

Com o `ISessionFactory`, podemos construir uma instância de `ISession` utilizando o método `OpenSession`:

```
ISession session = sessionFactory.OpenSession();
```

Agora que já aprendemos como abrir a sessão do NHibernate, vamos utilizá-la para gravar um usuário no banco de dados:

```
Usuario novoUsuario = new Usuario();
novoUsuario.Nome = "Murilo";
```

Para gravar o usuário, utilizaremos o método `Save()` do objeto `session`, que recebe como parâmetro o objeto que será inserido:

```
session.Save(novoUsuario);
```

Toda modificação no banco de dados precisa ser feita dentro de uma transação. Antes de chamarmos `Save`, queremos começar uma transação e depois, queremos fazer o commit.

Transações são criadas através do método `BeginTransaction` da session, esse método devolve um `NHibernate.ITransaction` que representa a transação criada:

```
ITransaction transacao = session.BeginTransaction();
```

Para terminar a transação e efetivar as mudanças no banco de dados, utilizamos o método `Commit` da transacao:

```
transacao.Commit();
```

O código para Gravar um novo usuário no banco de dados fica da seguinte forma:

```
ITransaction transacao = session.BeginTransaction();
session.Save(novoUsuario);
transacao.Commit();
```

E agora que terminamos de usar a sessão do NHibernate, precisamos fechá-la:

```
session.Close();
```

Pronto! Já temos nosso primeiro usuário cadastrado no banco de dados. Reparem que não precisamos definir um valor para o campo Id, pois ele foi mapeado como um campo auto incrementável e será preenchido automaticamente pelo banco de dados.

Isolando a criação do ISession

No NHibernate, toda comunicação com o banco de dados é feita através do `ISession`. Como a criação desse objeto é complicada, vamos isolar essa lógica dentro de um novo método da classe `NHibernateHelper` chamado `AbreSession`.

```
public class NHibernateHelper
{
    public static ISession AbreSession()
    {
        Configuration cfg = NHibernateHelper.RecuperaConfiguracao();
        ISessionFactory fabrica = cfg.BuildSessionFactory();
        return fabrica.OpenSession();
    }
}
```

Porém toda vez que criamos o `ISessionFactory`, o NHibernate lê os arquivos de configuração e registra todas as entidades, portanto não queremos criar uma instância diferente de `ISessionFactory` para cada sessão, precisamos de uma instância da fábrica para a aplicação.

Para usarmos a mesma fábrica em todos os pontos da aplicação, vamos guardá-la dentro de um atributo estático do `NHibernateHelper`:

```
public class NHibernateHelper
{
    private static ISessionFactory fabrica;
}
```

Para garantir que criaremos a fábrica apenas uma vez, vamos utilizar a declaração com inicialização do c#:

```
public class NHibernateHelper
{
    private static ISessionFactory fabrica =
        NHibernateHelper.CriaSessionFactory();

    private static ISessionFactory CriaSessionFactory()
    {
        Configuration cfg = NHibernateHelper.RecuperaConfiguracao();
```

```
            return cfg.BuildSessionFactory();
        }
    }
```

O método `AbreSession` é simplificado para:

```
public static ISession AbreSession()
{
    return fabrica.OpenSession();
}
```

Buscando e removendo uma entidade

Toda tabela do banco de dados deve possuir uma chave primária, um identificador único para cada registro. Esse campo é utilizado para vincular uma tabela a outra e para conseguirmos diferenciar um registro do outro. Em nosso exemplo, a chave primária da tabela Usuario é o campo Id.

No NHibernate, quando queremos resgatar um objeto pelo Id, utilizamos o método `Get` da session:

```
Usuario usuario = session.Get<Usuario>(idDoUsuario);
```

Quando queremos apagar uma entidade, utilizamos o método `Delete` do `ISession` :

```
ISession session = // abre a session
ITransaction transacao = session.BeginTransaction();
Usuario usuario = session.Get<Usuario>(1);

session.Delete(usuario);
transacao.Commit();
```

Data Access Object (DAO)

Até agora, fizemos todos os exemplo utilizando a `ISession` em qualquer ponto do código, ou seja, estamos espalhando o código de acesso ao banco de dados em vários pontos da aplicação, isso faz com que o projeto fique desorganizado e difícil de manter.

Para resolver esse problema, podemos isolar o acesso ao banco de dados em uma classe cuja única função é acessar o banco de dados. Classes que isolam a lógica de acesso aos dados são chamadas de `Data Access Objects` , os DAOs.

Geralmente criamos uma classe DAO para cada entidade que possuímos, logo para isolarmos o acesso a tabela de usuários, criaremos o `UsuariosDAO` . No projeto Loja, vamos criar uma pasta chamada `DAO` e dentro dessa pasta criaremos uma classe chamada `UsuariosDAO` .

Dentro do `UsuariosDAO` , isolaremos o código que adiciona um usuário:

```
public class UsuariosDAO
{
    public void Adiciona(Usuario usuario)
```

```
        {
            ITransaction transacao = session.BeginTransaction();
            session.Save(usuario);
            transacao.Commit();
        }
    }
```

O método que busca um usuário por id:

```
public class UsuariosDAO
{
    public Usuario BuscaPorId(int id)
    {
        return session.Get<Usuario>(id);
    }
}
```

O `UsuariosDAO` fica da seguinte forma:

```
public class UsuariosDAO
{
    public void Adiciona(Usuario usuario)
    {
        ITransaction transacao = session.BeginTransaction();
        session.Save(usuario);
        transacao.Commit();
    }

    public Usuario BuscaPorId(int id)
    {
        return session.Get<Usuario>(id);
    }
}
```

O código do `UsuariosDAO` utiliza a session, porém essa variável não foi criada. Como todos os métodos do DAO precisam de um `ISession` , vamos abrir a sessão no construtor:

```
public class UsuarioDAO
{
    private ISession session;

    public UsuarioDAO()
    {
        this.session = NHibernateHelper.AbreSession();
    }
    // resto da classe
}
```

Se na lógica de negócio precisarmos de diversos DAOs, cada um deles abrirá uma nova session, porém a mesma session poderia ser compartilhada entre os diversos DAOs.

A estratégia de abrir a conexão no construtor parece boa, porém ela não consegue resolver o problema ilustrado acima. Os DAOs precisam da session para implementarem seus métodos, porém não queremos uma session por DAO, precisamos modificar os DAOs para que eles recebam a instância de ISession como argumento do construtor.

```
public class UsuarioDAO
{
    private ISession session;

    public UsuarioDAO(ISession session)
    {
        this.session = session;
    {
}
```

Agora a session pode ser compartilhada. Esse código que acabamos de implementar é uma das formas de um padrão de projeto conhecido como Injeção de Dependências.

Agora que o UsuarioDAO está recebendo a session no construtor, para gravarmos um novo usuario no banco de dados, utilizaremos o seguinte código:

```
Usuario usuario = new Usuario();
usuario.Nome = "Murilo";

ISession session = NHibernateHelper.AbreSession();
UsuariosDAO usuariosDAO = new UsuariosDAO(session);
usuariosDAO.Adiciona(usuario);
```

Estado dos objetos

Agora que aprendemos a utilizar o `ISession` do NHibernate, aprenderemos como ele gerencia internamente os estados dos objetos. Para ilustrar a explicação, utilizaremos a seguinte tabela de usuários:

```
+----+--------+
| id | nome   |
+----+--------+
|  1 | Victor |
|  2 | Murilo |
+----+--------+
```

Quando fazemos:

```
Usuario usuario = session.Get<Usuario>(1);
```

O NHibernate faz a consulta no banco de dados e nos devolve a referência para o usuário de id 1 (chamado Victor), mas antes de devolver a referência, ele a guarda dentro da `session`. Entidades armazenadas dentro de `ISession` estão em um estado chamado Persistent e são tratados de forma especial pelo NHibernate.

Quando começamos uma transação, o NHibernate guarda o valor inicial dos atributos das entidades no estado persistent. Ao final da transação, se o estado da entidade foi modificado, o NHibernate faz com que essa modificação seja refletida no

banco de dados, ou seja, ele executa um update. Por exemplo, no código abaixo, o Hibernate executará um update para modificar o registro de id 1:

```
ITransaction Transacao = Session.BeginTransaction();
// Nesse ponto o Usuario tem a propriedade nome com o valor Victor.
Usuario UsuarioDoBanco = Session.Get<Usuario>(1);

UsuarioDoBanco.Nome = "Victor Harada";
Console.WriteLine("No commit, o NHibernate detecta que o Usuario foi modificado e " +
                  "executa um Update no banco de dados");
Transacao.Commit();
```

Podemos também criar um novo usuário:

```
Usuario NovoUsuario = new Usuario();
```

O NovoUsuario não possui representação no banco de dados (nunca foi inserido) e nunca passou pelo `ISession`, objetos nessa situação, estão no estado Transient.

Vimos que o código abaixo deleta um usuário:

```
ITransaction transacao = session.BeginTransaction();
Usuario usuario = session.Get<Usuario>(1);

session.Delete(usuario);
transacao.Commit();
```

Nesse código o `usuario` está no estado persistent, porém logo em seguinda, chamamos o `Delete`. O método `Delete` remove o objeto do banco de dados e, portanto, deve mudar seu estado de persistent para transient.

Ao fecharmos `ISession`, todos os objetos que estavam gerenciados continuam existindo no programa, porém não estão mais associados com uma sessão válida. Objetos nessa condição estão em um estado conhecido como Detached.

Vimos que para fazermos a atualização, precisamos de um objeto no estado persistent, porém quando estamos em uma aplicação Asp.Net MVC, os objetos recuperados da requisição não estão nesse estado, portanto para realizarmos um update, precisamos carregar o objeto utilizando o `Get` e depois copiar as informações do objeto que foi recuperado da requisição para o objeto que foi carregado do banco de dados. Para resolver esse tipo de situação, o hibernate nos oferece o método `Merge` no `ISession`.

O `Merge` recebe uma entidade, que pode estar em qualquer estado, e faz com que os valores dos atributos dessa entidade sejam refletidos no banco de dados, ou seja, o `Merge` faz um update no banco de dados mesmo quando a entidade que passamos como argumento não está no estado persistent.