Prefácio

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Este documento é uma tradução da documentação original

Página principal

Trabalhar com orientação a objetos junto de banco de dados relacionais pode ser complicado e muito demorado em ambientes empresariais atuais. O NHibernate é uma ferramenta de mapeamento de objeto-relacional para ambientes .Net Framework. O termo mapeamento objeto-relacional (ORM) refere-se à técnica de mapear uma representação dos dados de um modelo de objeto para o modelo de dados relacionais baseados em SQL Shema.

NHibernate não somente faz o mapeamento de classes .Net para tabelas no banco de dados (e de tipos de dados .NET para tipos de dados SQL), mas também oferece facilidades nas consultas e pode reduzir o gasto de tempo de desenvolvimento se comparado com a forma convencional de SQL e ADO.NET.

NHibernate corresponde a 95% das tarefas comuns relacionadas a persistência de dados. NHibernate pode não ser a melhor solução para centralizar o acesso a dados nas aplicações que utilizam apenas *store procedures* para implementações da lógica de negócio, ele é mais útil com arquiteturas orientadas a modelo de domínio e lógicas de negócio baseadas mas entidades. Entretanto, NHibernate pode com certeza ajudar a remover ou encapsular seus comandos SQL e irá ajudar nas tarefas mais comuns como consulta de dados retornando diretamente objetos.

Se você é novo no NHibernate, em mapeamento de objeto-relacional ou no .Net Framework, siga os passos a seguir:

- Leia o Capítulo 1 para, Iniciando com IIS e Microsoft SQL Server em 30 minutos.
- Leia o Capítulo 2, Entendendo a arquitetura e como usar o ambiente do NHibernate.
- Use essa documentação como referência primária de informação. Leia também o "Hibernate in Action" ou o "NHibernate in Action" case necessite de uma ajuda passo-a-passo. Em [1] existem tutoriais com diversos exemplos.
- Veja as respostas das perguntas mais frequentes em blogs da comunidade NHibernate.
- Leia os exemplos de membros da comunidades NHIbernate.
- O site oficial do NHibernate é uma grande fonte de design patterns e várias soluções de integração.

Se você tiver dúvidas, use o grupo NHIbernate. Nós também fornecemos o JIRA como sistema de relatório de bugs onde é possivel enviar novos bugs e pedidos de novos recursos. Se você está interessado em ajudar no desenvolvimento do NHibernate, entre

na lista de email do projeto. Se você se interessou em traduzir essa documentação, fale com um dos desenvolvedores da lista de email.

Capítulo 1. Iniciando com IIS e Microsoft SQL Server

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Capítulo 1. Iniciando com IIS e Microsoft SQL Server

Como iniciar com NHibernate

Este tutorial explica como configurar o NHibernate 2.0.0 com um ambiente Microsoft. As ferramentas usadas nesse tutorial são:

- Microsoft Internet Information Services (IIS) - um servidor que suporta ASP.NET.
- Microsoft SQL Server 2005 - um servidor de banco de dados. Neste tutorial usamos a versão Express, você pode baixar gratuitamente no site da Microsoft. Para suportar outros bancos de dados basta apenas alterar a proprietdade de dialeto do NHibernate e a configuração do driver que ele irá usar.
- Microsoft Visual Studio .NET 2005 - o ambiente de desenvolvimento.

Primeiramente temos que criar um novo projeto Web. Nós daremos o nome que *QuickStart* ao projeto e ele usará o diretório virtual semelhante a http://localhost/quickstart. Adicione a referência NHibernate.dll ao projeto. O Visual Studio irá automaticamente copiar as bibliotecas e dependencias para o diretório de saida do projeto. Se você estiver utilizando um banco de dados diferente do SQL Server, adicione as referências corretas desse banco ao seu projeto.

Agora iremos configurar a conexão com o banco de dados para o NHibernate. Para isso abra o arquivo Web.config gerado automaticamente pelo Visual Studio e adicione os elemetos de acordo com a lista a seguir:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
       <section
           name="hibernate-configuration"
           type="NHibernate.Cfg.ConfigurationSectionHandler,
NHibernate" />
   </configSections>

   <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
       <session-factory>
           <property
name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
           <property
name="connection.provider">NHibernate.Connection.DriverConnectionProvi
der</property>
           <property name="connection.connection_string">
               Server=(local);initial catalog=quickstart;Integrated
Security=SSPI
           </property>
           <mapping assembly="QuickStart" />
       </session-factory>
   </hibernate-configuration>

   <system.web>
   ...
   </system.web>
</configuration>
```

O elemento <configSections> contém as definições e handlers que serão usados para processar conteúdos. Nós declaramos um manipulador de configuração de sessão. A sessão <hibernate-configuration> contém sua própria configuração, descrevendo que o NHibernate irá usar o Microsoft SQL Server 2005 como banco de dados e definimos também a string de conexão. O *dialect* é uma configuração requerida pois diferentes bancos de dados interpretam diferente nos comandos SQL. NHibernate irá cuidar de tratar as diferenças pois ele encapsula as diferenças dos principais bancos de dados comerciais e open source existentes no mercado.

Um ISessionFactory é um conceito do NHibernate de persistência única em que múltiplos bancos de dados podem ser usados criando-se diversos arquivos de configuração e criando multiplas configurações e objetos ISessionFactory na sua aplicação.

O último elemento da sessão <hibernate-configuration> declara que o assemblie *QuickStart* contém as classes e os arquivos de mapeamento. Os arquivos de mapeamento contém metadados que mapeiam nossas classes POCO para as tabelas no banco de dados (ou múltiplas tabelas). Iremos falar sobre arquivos de mapeamento posteriormente. Agora vamos escrever nossa primeira classe POCO e declarar os metadados do mapeamento dela.

Primeira classe de persistência

A melhor maneira de trabalharmos com o NHibernate é através de modelo de programação de classes persistentes chamado *Plain Old CLR Objects (POCO)*, que são chamados também de *Plain Ordinary CLR Objects*. Uma classe POCO tem os seus dados acessíveis através dos mecanismo do .Net chamado *propriedades*, que encapsula sua representação interna através de uma interface visível.

```
namespace QuickStart
{
   public class Cat
   {
       private string id;
       private string name;
       private char   sex;
       private float  weight;

       public Cat() {}

       public virtual string Id
       {
           get { return id; }
           set { id = value; }
       }

       public virtual string Name
       {
           get { return name; }
           set { name = value; }
       }

       public virtual char Sex
       {
           get { return sex; }
           set { sex = value; }
       }

       public virtual float Weight
       {
           get { return weight; }
           set { weight = value; }
       }
   }
}
```

O NHibernate não restringe o uso de tipos nas propriedades, todos os tipos primitivos do .Net (como string, char e DateTime) podem ser mapeados, inclusive classes da namespace System.COllections. Vocô pode mapear valores, coleção de valores ou associações a outras entidades. O *Id* é uma propriedade especial que representa um identificador de banco de dados (chave primária ou primary key) nas classes, ele é muito recomendado para entidades como a classe *Cat* listada acima. O NHibernate pode usar identificadores apenas internamente, sem ter que declara-los na classe, mas dessa forma perdemos em flexibilidade na arquitetura de nossa aplicação.

Nenhuma interface especial é necessária para nossas classes de persistência e nem precisamos ter subclasses de uma raiz especial para as classes de persistência. O NHibernate também não utiliza nenhum construtor em tempo de processamento como os manipuladores da linguagem intermediária (IL), ele se baseia exclusivamente em reflexão (System.Reflection) das classes .NET e recursos em tempo de execução

(através da biblioteca Castle.DynamicProxy2). Então, sem nehuma dependência em nossas classes POCO, nós podemos mapear as tabelas do banco de dados.

Para os usar os recursos do NHibernate mencionados em tempo de execução, as propriedades das classes devem ser definidas como publicas e declaradas como *virtual*.

Mapeando a classe Cat

O arquivo de mapeamento Cat.hbm.xml contém os metadados necessários para o mapeamento do nosso objeto-relacional. Os metadados incluem declaração da classe de persistência e o mapeamento das propriedades (para colunas e chaves estrangeiras para outras entidades) às tabelas do banco de dados.

Note que o arquivo Cat.hbm.xml deve ser configurado como **Embedded Resource**0.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
   namespace="QuickStart" assembly="QuickStart">

   <class name="Cat" table="Cat">


        <id name="Id">
            <column name="CatId" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex" />
        </id>

        <property name="Name">
            <column name="Name" length="16" not-null="true" />
        </property>
        <property name="Sex" />
        <property name="Weight" />
   </class>

</hibernate-mapping>
```

Todas as classes de pesistência devem ter um atributo de identificação (na verdade, apenas classes representam entidades, não dependem de objetos de valor que são mapeados como componentes das entidades). Esta propriedade é usada para distinguir objetos de persistência, por exemplo, duas classes Cat são iguais se CatA.Id.Equals(CatB.Id) seja *true*, esse conceito é chamado de identidade de banco de dados. O NHibernate conta com outros diversos geradores de identificação para diferentes cenários (incluindo geradores nativos de banco de dados, identificadores de tabela *hi/lo* e identificadores atribuidos pela aplicação). Nós usamos *UUID* como gerador de identificação (apenas recomendado para teste, prefira identificadores do tipo *int*) e também especificamos a coluna *CatId* da nossa tabela *Cat* no banco de dados para o NHibernate gerar o valor de identificação (uma chave primária na tabela).

Todas as outras propriedades da classe *Cat* são mapeadas da mesma maneira. No caso da propriedade *Name*, nós mapeamos explicitamente como uma tag <*column*>. Isso é útil quando desejamos gerar nossas tabelas do banco de dados (com o uso de cláusulas SQL DDL) através das declarações do mapeamento com a ferramenta SchemaExport do NHibernate. Para as outras propriedades usams as configurações padrão do NHibernate

que é o que necessitamos nesse momento. A tabela *Cat* no banco de dados deve ser semelhante a isso:

```
 Column |     Type      | Modifiers
--------+---------------+----------------------
 CatId  | char(32)      | not null, primary key
 Name   | nvarchar(16)  | not null
 Sex    | nchar(1)      |
 Weight | real          |
```

Por enquanto você deve criar as tabelas do baco de dados manualmente, posteriormente no Capítulo 17 você verá como automatiza esse trabalho com a ferramenta SchemaExport. Essa ferramenta pode criar completas cláusulas SQL DDL incluindo definições de tabela, personalisação de *constraints*, *unique constraints* e índices. Se você estiver usando SQL Server você deve conceder as devidas permissões para que o usuário ASPNET possa utilizar o banco de dados.

Brincando com os *Cats*

Iremos agora iniciar com uma ISession do NHibernate. Ela é uma interface de mapeamento que usamos para persistir e consultar nossas entidades *Cat* do banco de dados. Mas primeiramente, precisamos ter uma ISession de uma ISessionFactory (NHibernate é Unit-of-Work):

```
ISessionFactory sessionFactory = new
Configuration().Configure().BuildSessionFactory();
```

Uma ISessionFactory é responsável por um banco de dados e deve usar apenas um arquivo de configração (Web.config ou hibernate.hbm,xml). Você pode configurar outras propriedades (ou até mesmo alterar os metadados de mapeamento) para acessar as configurações antes de você construir uma ISessionFactory (ela é imutável). Mas onde podemos criar uma ISessionFactory e como podemos acessa-la em nossa aplicação?

Uma ISessionFactory é frequentemente apenas construido uma única vez quando se inicia a aplicação, como o evento *Application_Start*. Isso significa também que você não deve instanciala em suas páginas ASP.NET, mas em algum outro lugar. Aprofundando o assunto, nós presimamos de algum tipo de *Singleton Pattern* para que possaos acessar a ISessionFactory facilmente no código. Essa abordagem resolve alguns problemas como configuração e fácil acesso a ISessionFactory.

Nós implementamos uma classe chamada NHibernateHelper da seguinte maneira:

```
using System;
using System.Web;
using NHibernate;
using NHibernate.Cfg;

namespace QuickStart
{
   public sealed class NHibernateHelper
   {
```

```
        private const string CurrentSessionKey =
"nhibernate.current_session";
        private static readonly ISessionFactory sessionFactory;

        static NHibernateHelper()
        {
            sessionFactory = new
Configuration().Configure().BuildSessionFactory();
        }

        public static ISession GetCurrentSession()
        {
            HttpContext context = HttpContext.Current;
            ISession currentSession = context.Items[CurrentSessionKey]
as ISession;

            if (currentSession == null)
            {
                currentSession = sessionFactory.OpenSession();
                context.Items[CurrentSessionKey] = currentSession;
            }

            return currentSession;
        }

        public static void CloseSession()
        {
            HttpContext context = HttpContext.Current;
            ISession currentSession = context.Items[CurrentSessionKey]
as ISession;

            if (currentSession == null)
            {
                // No current session
                return;
            }

            currentSession.Close();
            context.Items.Remove(CurrentSessionKey);
        }

        public static void CloseSessionFactory()
        {
            if (sessionFactory != null)
            {
                sessionFactory.Close();
            }
        }
    }
}
```

Esta classe não apenas trata uma ISessionFactory como um atributo estático mas também *lembra* que uma ISession está na atual em uma requisição HTTP.

Uma ISessionFactory é *threadsafe*, diversas threads podem acessa-la e requisitarem uma ISession. Uma ISession não é um objeto *threadsafe*, ele representa uma unidade-de-trabalho simples com o banco de dados. ISession´s são abertas por uma ISessionFactory e são fechadas quando todo o trabalho é completado.

```
ISession session = NHibernateHelper.GetCurrentSession();

ITransaction tx = session.BeginTransaction();
Cat princess = new Cat();

princess.Name = "Princess";
princess.Sex = 'F';
princess.Weight = 7.4f;

session.Save(princess);
tx.Commit();

NHibernateHelper.CloseSession();
```

Em uma ISession, toda operação com o banco de dados ocorre dentro de uma transação que isola as operações do banco de dados (todas operações *ready-only*). Nós usaremos a API ITransaction do NHibernate para abstrair o resumo de uma estratégia de transação (em nosso caso, transações ADO.NET). Observe que o exemplo acima não possúi qualquer manipulador de excessão.

Observe também que podemos chamar NHibernateHelper.GetCurrentSession(); quantas vezes forem necessárias, você terá sempre a atual ISession da sua requisição HTTP. Devemos também garantir que a ISession será fechada depois de concluírmos nossas unidades-de-trabalho, podendo ser no evento *Application_EndRequest* da sua aplicação ou em um HttpModule antes da resposta HTTP ser enviada. Um efeito muito interessante é quando fazemos requisições *lazy*: a ISession ainda estará aberta quando nossa *view* é renderizada para que então o NHibernate possa carregar obejtos ainda não inicializados quando você necessitar deles.

O NHibernate possúi diversos métodos que podemos usar para consultar objetos do banco de dados. **O mais flexivel** é usarmos o que chamamos de *Hibernate Query Language (HQL)* que possúi uma curva de aprendizdo rápida e poderosos recursos para consultas de nossos objetos-relacionais.

```
using(ITransaction tx = session.BeginTransaction())
{
   IQuery query = session.CreateQuery("select c from Cat as c where
c.Sex = :sex");
   query.SetCharacter("sex", 'F');
   foreach (Cat cat in query.Enumerable())
   {
       Console.Out.WriteLine("Female Cat: " + cat.Name);
   }
   tx.Commit();
}
```

O NHibernate também oferece uma API orientada a objeto para efetuarmos nossas consultas. NHibernate naturalmente usa *IDbCommands* e parametros vinculados em todas as consultas SQL enviadas ao banco de dados. Você pode usar também conultas SQL diretamente ou obter uma conexão do ADO.NET de ISession em casos raros.

Finalmente

No nosso primeiro capítulo, vimos superficialmente algumas caracteristicas do NHibernate. Observe que nós não incluimos quaisquer códigos ASP.NET em nossos exemplos. Você deve criar suas páginas ASP.NET sozinho e inserir os códigos do NHibernate que você viu nesse capítulo.

Tenha em mente que NHibernate, como uma camada de acesso aos dados, é hermeticamente integrada a sua aplicação. Frequentemente todas as outras camadas dependem de um mecanismo persistência. Esteja certo de que você compreendeu esse conceito.

Capítulo 2. Arquitetura

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

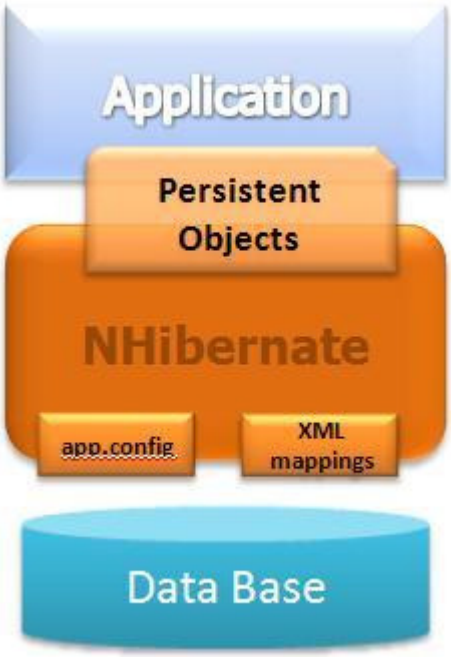
Tabela de conteúdo

[esconder]

Capítulo 2. Arquitetura

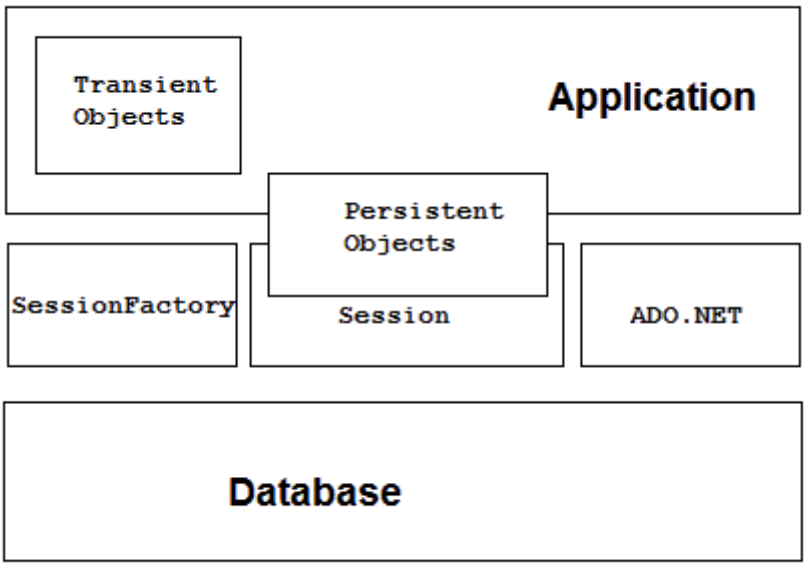
Visão Geral

Vejam uma visão em alto nível da arquitetura do NHibernate:

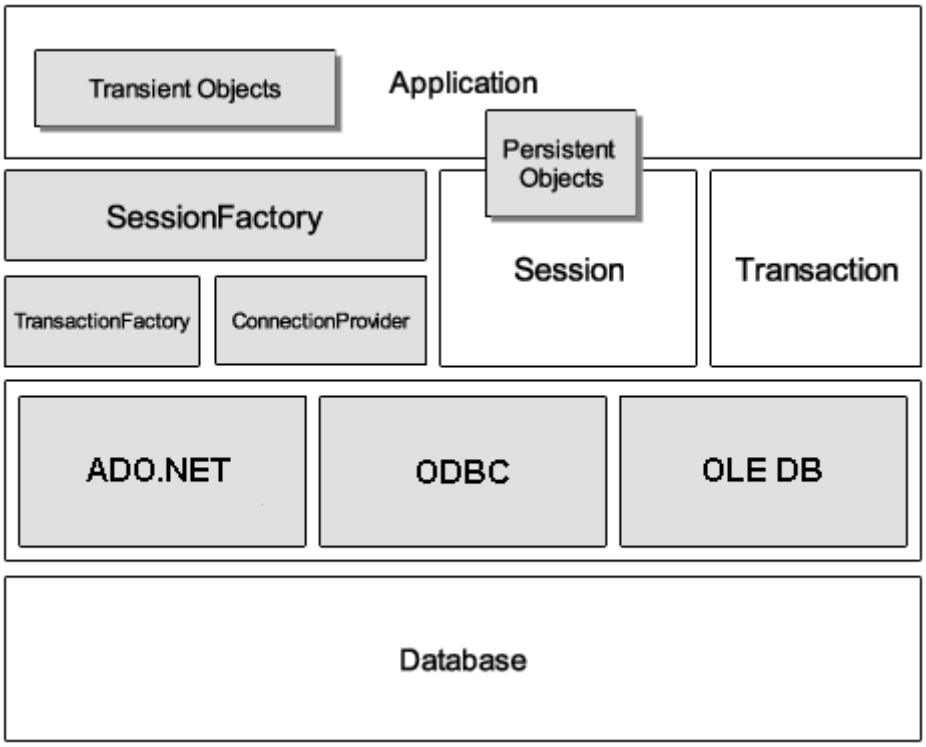


Este diagrama mostra o uso do banco de dados e dados de configuração que fornecem serviços de persistência (e objetos de persistência) ao NHibernate para a aplicação.

Gostaríamos de mostrar também uma visão mais detalhada da arquitetura em tempo de execução. Infelizmente, o NHibernate é flexível e suporta várias abordagens. Nós iremos mostrar dois extremos. A arquitetura "lite" em que a aplicação já tem seu próprio gerenciamento de conexões e transações com ADO.NET. Esta abordagem utiliza apenas um pequeno conjunto das API´s do NHibernate:



Em uma arquitetura mais completa faz do NHibernate um gerenciador que cuida de outros detalhes.



Aqui abaixo temos algumas definições dos objetos dos diagramas:

ISessionFactory (NHibernate.ISessionFactory)

Um objeto threadsafe (imutável) com o cache que contém os mapeamentos compilados para um único banco de dados. Uma "fábrica" de ISession e um cliente de IConnectionProvider. Ele pode conter um cache opcional (segundo nível) de dados que são reutilizáveis entre transações em um processo ou um cluster.

ISession (NHibernate.ISession)

Um objeto de thread única, de curta duração que representa uma conversação entre a aplicação e a camada de persistência. Possui uma conexão ADO.NET. Possui uma "fábrica" de ITransaction. Possui (primeiro nível) um cache contendo os objetos persistentes usados quando navegamos pelos objetos ou consultamos o identificador dos objetos.

Objetos de Persistência e Coleções

De curta duração, de thread única, mantém estado e regras de negócio. São também chamados de POCO´s, estão associados a uma ISession. Logo que as sessões são fechadas, eles são desatachados da sessão e ficam livres na camada de aplicação (diretamente como uma transferência de objetos para a camada de apresentação)

Objetos Transidentes e Coleções

Instancias de classes persistentes que estão associadas a uma ISession. Elas podem ser instanciadas pela aplicação e não persistidas ou terem sido instanciadas por uma ISession que estava fechada.

ITransaction (NHibernate.ITransaction)

(Opcional) Uma thread única, objeto de curta duração usado pela aplicação para especificar unidades de trabalho. Abstrai a utilização de transações ADO.NET. Uma ISession poderá conter várias ITransaction´s em alguns casos.

IConnectionProvider (NHibernate.Connection.IConnectionProvider)

(Opcional) Uma "fábrica" de conexões e comandos ADO.NET. Abstrai a aplicação de implementações de IDbConnectin e IDbCommand de diferentes fornecedores. Não é exposta para a aplicação mas pode ser extendida/implementada.

IDriver (NHibernate.Driver.IDriver)

(Opcional) Uma interface que encapsula diferentes provedores de ADO.NET, tais como convenções de parametros nomeados e suporte a recursos do ADO.NET.

ITransactionFactory (NHibernate.Transaction.ITransactionFactory)

(Opcional) Uma "fábrica" de instâncias ITransaction. Não é exposta à aplicação mas pode ser extendida/implementada.

Dado uma arquitetura pequena, os pedidos da aplicação passam por ITransaction/ITransactionFactory e/ou API´s do IConnectionProvider para chamadas ADO.NET diretamente.

Estado das Instâncias

Uma instância de uma classe de persistência pode ser encontrada em três diferentes estados que são definidas de acordo com o contexto de persistência. Encontramos nos contextos de pesistência do objeto ISession do NHibernate os seguintes estados:

transient

A instância não é nem nunca foi associada a um contexto de persistência. Ela não tem uma identidade de pesistência (um valor na chave-primária)

persistent

A instância está atualmente associada a um contexto de persistência. Ela tem uma identidade de persistência (um valor na chave-primária) e, talvez, um registro correspondente no banco de dados. Para um determinado contexto de persistência, o NHibernate garante que a identidade de persistência é igual a identidade CLR (objeto localizado em memória).

detached

A instância já foi associada a um contexto de persistência mas o contexto já foi fechado ou a instância foi serializada para outro processo. Ela tem uma identidade de persistência e, talvez, um registro correspondente no banco de dados. Para instâncias "desatachadas" do contexto, o NHibernate não garante os relacionamentos entre a identidade de persistência e as identidades CLR.

Contextualização de Sessões

Na maioria das aplicações que utilizam NHibernate precisam de alguma forma para "contextualizar" as sessões, onde uma determinada sessão é na verdade um determinado contexto. Contudo, os contextos são diferentes. Diferentes contextos definem diferentes escopos.

Implementado na versão 1.2, foi adicionado ao NHibernate o método ISessionFactory.GetCurrentSession(). O processamento por trás de ISessionFactory.GetCurrentSession() é plugável. Uma interface de extenção (NHibernate.Context.ICurrentSessionContext) e um novo parametro de configuração (hibernate.current_session_context_class) foram adicionados para permirir "plugabilidade" do escopo e do contexto das sessões atuais.

Veja a documentação da API para a interface NHibernate.Context.ICurrentSessionContext para uma discução detalhada desse contrato. Ela define um método simples, CurrentSession(), na qual a implementação é responsável por monitorar o atual contexto da sessão. Na nova versão, NHibernate 2.0.0 teremos como as principais implementações dessa interface:

- *NHibernate.Context.ManagedWebSessionContext* - as atuais sessões são monitoradas por HttpCOntext. Contudo, você será responsável por vincular e desvincular uma instância de ISession com métodos estáticos dessa classe; ela nunca abre, faz "flush" ou fecha ela mesma.

- *NHibernate.Context.CallSessionContext* - atuais sessões são monitoradas por CallContext. Você será responsável por vincular e desvincular de uma instância de ISession com métodos estáticos da classe CurrentSessionContext.

- *NHibernate.Context.ThreadStaticSessionContext* - atuais sessões são armazenadas em uma variável thread-static. Este contexto suporta somente uma "fábrica" de sessões. Você será responsável por vinvular e desvincular uma instância de ISession com métodos estáticos da classe CurrentSessionContext.

- *NHibernate.Context.WebSessionContext* - análogo ao ManagedWebSessionContext descrito acima, armazena a sessão atual em um HttpContext. Você será responsável por vinvular e desvincular uma instância de ISession com métodos estáticos da classe CurrentSessionContext.

Os parametros de configuração de hibernate.current_session_context_class definem que a implementação de NHibernate.Context.ICurrentSessionContext deve ser usada. Frequentemente, o valor desse parametro seria o nome da classe que implementa seu uso (incluindo o nome do assembly), para essa nova versão do NHibernate existem

"apelidos" curtos correspondentes a: "managed_web", "call", "thread_static", e "web", respectivamente.

Capítulo 3. Configuração do ISessionFactory

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Capítulo 3. Configuração do ISessionFactory

O NHibernate foi desenhado para trabalhar com diferentes ambientes, por esse motivo existem um grande número de parametros de configuração. Felizmente, os parametros mais comuns de configuração do NHibernate são distribuidos como o exemplo do arquivo App.config (encontrado em src\NHibernate.Test) e diversos templates para suportar diversos gerenciadores de sistema de bancos de dados relacionais (encontrado em src\NHibernate.Config.Templates). Normalmente temos apenas que inserir um desses arquivos e customiza-los de acordo com nossas necessidades.

Configuração via Programação

Uma instância de NHibernate.Cfg.Configuration representa um conjunto completo de mapeamentos de tipos .NET para aplicação de um banco de dados SQL. A configuração

é usada para construir um ISessionFactory (imutável). Os mapeamentos são compilados de vários arquivos XML de mapeamento.

Você pode ter uma instância de Configuration instanciando-a diretamente. Aqui um exemplo de criação de um "datastore" definidos através de dois arquivos de configuração XML:

```
Configuration cfg = new Configuration()

.AddFile("Item.hbm.xml")
.AddFile("Bid.hbm.xml");
```

Uma alternativa (algumas vezes melhor) é a forma de deixar o NHibernate carregar o mapeamento do arquivo através de recursos embutidos (Embedded Resource):

```
Configuration cfg = new Configuration()

.AddClass(typeof(NHibernate.Auction.Item))
.AddClass(typeof(NHibernate.Auction.Bid));
```

Dessa forma o NHibernate irá mapear os arquivos nomeados da seguinte forma: "NHibernate.Auction.Item.hbm.xml" e "NHibernate.Auction.Bid.hbm.xml" embutidos como recursos do assembly. Essa abordagem elimina arquivos com nome codificado.

Outra alternativa (provavelmente a melhor) é a forma em que o NHibernate carrega todos os arquivos de mapeamento contidos no assembly:

```
Configuration cfg = new Configuration()
.AddAssembly( "NHibernate.Auction" );
```

Dessa forma o NHibernate iá procurar no assembly pelos recursos que tenham a extensão .hbm.xml. Essa abordagem elimina arquivos com nome codificado e garante que os arquivos de mapeamento no assembly sejam carregados.

Se uma ferramenta como o Visual Studio .NET ou [1] for usada para dar um build no assembly, certifique que os arquivos .hbm.xml são compilados no assembly como *Embedded Resources*.

Uma configuração (NHibernate.Cfg.Configuration) também especifica várias propriedades opcionais:

```
IDictionary<string, string> props = new Dictionary<string, string>();
...
Configuration cfg = new Configuration()
   .AddClass(typeof(NHibernate.Auction.Item))
   .AddClass(typeof(NHibernate.Auction.Bind))
   .SetProperties(props);
```

Uma configuração (NHibernate.Cfg.Configuration) é necessária em "tempo de configuração de objeto" e é desnecessária após ISessionFactory ser instanciado.

Obtendo um ISessionFactory

Quando todos os arquivos de mapeamento estiverem sido verificados pelo objeto Configuration, a aplicação obtém uma "fábrica" de instancias ISession. Esta "fábrica" destina-se a ser compartilhada com todas as threads da aplicação.

```
ISessionFactory sessions = cfg.BuildSessionFactory();
```

Contudo, o NHibernate permite sua aplicação instanciar mais de um ISessionFactory. Isso é útil quando você necessitar usar mais de um banco de dados.

Usando ADO.NET desde a Conexão

Um ISessionFactory pode abrir uma ISession de uma conexão ADO.NET. Este design obtém uma conexão ADO.NET:

```
IDbConnection conn = myApp.GetOpenConnection();
ISession session = sessions.OpenSession(conn);

// faça algum trabalho de acesso a dados
```

O pedido deve ser feito com cuidado para não abrirmos duas ISession´s concorrentes da mesma conexão ADO.NET!

NHibernate provided ADO.NET connection

Alternatively, you can have the ISessionFactory open connections for you. The ISessionFactory must be provided with ADO.NET connection properties in one of the following ways:

- Pass an instance of IDictionary mapping property names to property values to Configuration.SetProperties().
- Add the properties to a configuration section in the application configuration file. The section should be named nhibernate and its handler set to System.Configuration.NameValueSectionHandler.
- Include <property> elements in a configuration section in the application configuration file. The section should be named hibernate-configuration and its handler set to NHibernate.Cfg.ConfigurationSectionHandler. The XML namespace of the section should be set to urn:nhibernate-configuration-2.2.
- Include <property> elements in hibernate.cfg.xml (discussed later).

If you take this approach, opening an ISession is as simple as:

```
ISession session = sessions.OpenSession(); // open a new Session
// do some data access work, an ADO.NET connection will be used on
demand
```

All NHibernate property names and semantics are defined on the class NHibernate.Cfg.Environment. We will now describe the most important settings for ADO.NET connection configuration.

NHibernate will obtain (and pool) connections using an ADO.NET data provider if you set the following properties:

Table 3.1. NHibernate ADO.NET Properties

Nome da Propriedade	Propósito
connection.provider_class	The type of a custom IConnectionProvider. eg. full.classname.of.ConnectionProvider if the Provider is built into NHibernate, or full.classname.of.ConnectionProvider, assembly if using an implementation of IConnectionProvider not included in NHibernate.
connection.driver_class	The type of a custom IDriver, if using DriverConnectionProvider. full.classname.of.Driver if the Driver is built into NHibernate, or full.classname.of.Driver, assembly if using an implementation of IDriver not included in NHibernate. This is usually not needed, most of the time the hibernate.dialect will take care of setting the IDriver using a sensible default. See the API documentation of the specific dialect for the defaults.
connection.connection_string	Connection string to use to obtain the connection.
connection.connection_string_name	The name of the connection string (defined in <connectionStrings> configuration file element) to use to obtain the connection.
connection.isolation	Set the ADO.NET transaction isolation level. Check System.Data.IsolationLevel for meaningful values and the database's documentation to ensure that level is supported. eg. Chaos, ReadCommitted, ReadUncommitted, RepeatableRead, Serializable, Unspecified
connection.release_mode	Specify when NHibernate should release ADO.NET connections. See Section 10.7, "Connection Release Modes". eg. auto (default) | on_close | after_transaction Note that this setting only affects ISessions returned from ISessionFactory.OpenSession. For

	ISessions obtained through ISessionFactory.GetCurrentSession, the ICurrentSessionContext implementation configured for use controls the connection release mode for those ISessions. See Section 2.3, “Contextual Sessions”.
command_timeout	Specify the default timeout of IDbCommands generated by NHibernate.
adonet.batch_size	Specify the batch size to use when batching update statements. Setting this to 0 (the default) disables the functionality. See Section 16.6, "Batch updates".

This is an example of how to specify the database connection properties inside a web.config:

```
<configuration>
    <configSections>
        <section name="hibernate-configuration"
                 type="NHibernate.Cfg.ConfigurationSectionHandler,
NHibernate" />
    </configSections>
    <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
        <session-factory>
            <property
name="connection.provider">NHibernate.Connection.DriverConnectionProvi
der, NHibernate</property>
            <property name="connection.connection_string">
                Server=127.0.0.1; Initial Catalog=thedatabase;
Integrated Security=SSPI
            </property>
                <property
name="dialect">NHibernate.Dialect.MsSql2000Dialect</property>
                <property
name="current_session_context_class">managed_web</property>
        </session-factory>
    </hibernate-configuration>
</configuration>
```

NHibernate relies on the ADO.NET data provider implementation of connection pooling.

You may define your own plugin strategy for obtaining ADO.NET connections by implementing the interface NHibernate.Connection.IConnectionProvider. You may select a custom implementation by setting connection.provider_class.

Optional configuration properties

There are a number of other properties that control the behaviour of NHibernate at runtime. All are optional and have reasonable default values.

System-level properties can only be set manually by setting static properties of NHibernate.Cfg.Environment class or be defined in the <hibernate-configuration>

section of the application configuration file. These properties cannot be set using Configuration.SetProperties.

Table 3.2. NHibernate Configuration Properties

Property name	Purpose
dialect	The classname of a NHibernate Dialect - enables certain platform dependent features. eg. full.classname.of.Dialect, assembly
default_schema	Qualify unqualified tablenames with the given schema/tablespace in generated SQL. eg. SCHEMA_NAME
use_outer_join	Enables outer join fetching. Deprecated, use max_fetch_depth. eg. true | false
max_fetch_depth	Set a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A 0 disables default outer join fetching. eg. recommended values between 0 and 3
use_reflection_optimizer	Enables use of a runtime-generated class to set or get properties of an entity or component instead of using runtime reflection (System-level property). The use of the reflection optimizer inflicts a certain startup cost on the application but should lead to better performance in the long run. You can not set this property in hibernate.cfg.xml or <hibernate-configuration> section of the application configuration file. eg. true | false
bytecode.provider	Specifies the bytecode provider to use to optimize the use of reflection in NHibernate. Use null to disable the optimization completely, lcg to use lightweight code generation, and codedom to use CodeDOM-based code generation. eg. null | lcg | codedom
cache.provider_class	The classname of a custom ICacheProvider. eg. classname.of.CacheProvider, assembly
cache.use_minimal_puts	Optimize second-level cache operation to minimize writes, at the cost of more frequent reads (useful for clustered caches). eg. true | false

cache.use_query_cache	Enable the query cache, individual queries still have to be set cacheable. eg. true | false
cache.query_cache_factory	The classname of a custom IQueryCacheFactory interface, defaults to the built-in StandardQueryCacheFactory. eg. classname.of.QueryCacheFactory, assembly
cache.region_prefix	A prefix to use for second-level cache region names. eg. prefix
query.substitutions	Mapping from tokens in NHibernate queries to SQL tokens (tokens might be function or literal names, for example). eg. hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC
show_sql	Write all SQL statements to console. eg. true | false
hbm2ddl.auto	Automatically export schema DDL to the database when the ISessionFactory is created. With create-drop, the database schema will be dropped when the ISessionFactory is closed explicitly. eg. create | create-drop
use_proxy_validator	Enables or disables validation of interfaces or classes specified as proxies. Enabled by default. eg. true | false
transaction.factory_class	The classname of a custom ITransactionFactory implementation, defaults to the built-in AdoNetTransactionFactory. eg. classname.of.TransactionFactory, assembly

SQL Dialects

You should always set the hibernate.dialect property to the correct NHibernate.Dialect.Dialect subclass for your database. This is not strictly essential unless you wish to use native or sequence primary key generation or pessimistic locking (with, eg. ISession.Lock() or IQuery.SetLockMode()). However, if you specify a dialect, NHibernate will use sensible defaults for some of the other properties listed above, saving you the effort of specifying them manually.

Table 3.3. NHibernate SQL Dialects (hibernate.dialect)

DB2 NHibernate.Dialect.DB2Dialect

RDBMS	Dialect	Remarks

DB2 for iSeries (OS/400)	NHibernate.Dialect.DB2400Dialect	
Ingres	NHibernate.Dialect.IngresDialect	
PostgreSQL	NHibernate.Dialect.PostgreSQLDialect	
PostgreSQL 8.1	NHibernate.Dialect.PostgreSQL81Dialect	This dialect supports FOR UPDATE NOWAIT available in PostgreSQL 8.1.
PostgreSQL 8.2	NHibernate.Dialect.PostgreSQL82Dialect	This dialect supports IF EXISTS keyword in DROP TABLE and DROP SEQUENCE available in PostgreSQL 8.2.
MySQL 3 or 4	NHibernate.Dialect.MySQLDialect	
MySQL 5	NHibernate.Dialect.MySQL5Dialect	
Oracle (any version)	NHibernate.Dialect.OracleDialect	
Oracle 9/10g	NHibernate.Dialect.Oracle9Dialect	
Sybase Adaptive Server Enterprise	NHibernate.Dialect.SybaseDialect	
Sybase Adaptive Server Anywhere	NHibernate.Dialect.SybaseAnywhereDialect	
Microsoft SQL Server 2000	NHibernate.Dialect.MsSql2000Dialect	
Microsoft SQL Server 2005	NHibernate.Dialect.MsSql2005Dialect	
Microsoft SQL Server 2005 Everywhere Edition	NHibernate.Dialect.MsSqlCeDialect	
Microsoft SQL Server 7	NHibernate.Dialect.MsSql7Dialect	

Firebird	NHibernate.Dialect.FirebirdDialect	Set driver_class to NHibernate.Driver.FirebirdClientDriver for Firebird ADO.NET provider 2.0.
SQLite	NHibernate.Dialect.SQLiteDialect	Set driver_class to NHibernate.Driver.SQLite20Driver for System.Data.SQLite provider for .NET 2.0.
Ingres 3.0	NHibernate.Dialect.IngresDialect	

Additional dialects may be available in the NHibernate Contrib project (see Part I, “NHibernateContrib Documentation”). At the time of writing this

Outer Join Fetching

If your database supports ANSI or Oracle style outer joins, outer join fetching might increase performance by limiting the number of round trips to and from the database (at the cost of possibly more work performed by the database itself). Outer join fetching allows a graph of objects connected by many-to-one, one-to-many or one-to-one associations to be retrieved in a single SQL SELECT.

By default, the fetched graph when loading an objects ends at leaf objects, collections, objects with proxies, or where circularities occur.

For a particular association, fetching may be configured (and the default behaviour overridden) by setting the fetch attribute in the XML mapping.

Outer join fetching may be disabled globally by setting the property hibernate.max_fetch_depth to 0. A setting of 1 or higher enables outer join fetching for one-to-one and many-to-one associations which have been mapped with fetch="join".

See Section 16.1, “Fetching strategies” for more information.

In NHibernate 1.0, outer-join attribute could be used to achieve a similar effect. This attribute is now deprecated in favor of fetch.

Custom ICacheProvider

You may integrate a process-level (or clustered) second-level cache system by implementing the interface NHibernate.Cache.ICacheProvider. You may select the custom implementation by setting cache.provider_class. See the Section 16.2, “The Second Level Cache” for more details.

Query Language Substitution

You may define new NHibernate query tokens using hibernate.query.substitutions. For example:

```
query.substitutions true=1, false=0
```

would cause the tokens true and false to be translated to integer literals in the generated SQL.

```
query.substitutions toLowercase=LOWER
```

would allow you to rename the SQL LOWER function.

Logging

NHibernate logs various events using Apache log4net.

You may download log4net from http://logging.apache.org/log4net/. To use log4net you will need a log4net configuration section in the application configuration file. An example of the configuration section is distributed with NHibernate in the src/NHibernate.Test project.

We strongly recommend that you familiarize yourself with NHibernate's log messages. A lot of work has been put into making the NHibernate log as detailed as possible, without making it unreadable. It is an essential troubleshooting device. Also don't forget to enable SQL logging as described above (show_sql), it is your first step when looking for performance problems.

Implementing an INamingStrategy

The interface NHibernate.Cfg.INamingStrategy allows you to specify a "naming standard" for database objects and schema elements.

You may provide rules for automatically generating database identifiers from .NET identifiers or for processing "logical" column and table names given in the mapping file into "physical" table and column names. This feature helps reduce the verbosity of the mapping document, eliminating repetitive noise (TBL_ prefixes, for example). The default strategy used by NHibernate is quite minimal.

You may specify a different strategy by calling Configuration.SetNamingStrategy() before adding mappings:

```
ISessionFactory sf = new Configuration()
   .SetNamingStrategy(ImprovedNamingStrategy.Instance)
   .AddFile("Item.hbm.xml")
   .AddFile("Bid.hbm.xml")
   .BuildSessionFactory();
```

NHibernate.Cfg.ImprovedNamingStrategy is a built-in strategy that might be a useful starting point for some applications.

XML Configuration File

An alternative approach is to specify a full configuration in a file named hibernate.cfg.xml. This file can be used as a replacement for the <nhibernate;> or <hibernate-configuration> sections of the application configuration file.

The XML configuration file is by default expected to be in your application directory. Here is an example:

<?xml version='1.0' encoding='utf-8'?> <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">

```
   <session-factory>
       <property
name="connection.provider">NHibernate.Connection.DriverConnectionProvi
der</property>
       <property
name="connection.driver_class">NHibernate.Driver.SqlClientDriver</prop
erty>
       <property
name="connection.connection_string">Server=localhost;initial
catalog=nhibernate;User Id=;Password=</property>
       <property name="show_sql">false</property>
       <property
name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
       <property name="use_outer_join">true</property>

       <mapping resource="NHibernate.Auction.Item.hbm.xml"
assembly="NHibernate.Auction" />
       <mapping resource="NHibernate.Auction.Bid.hbm.xml"
assembly="NHibernate.Auction" />
   </session-factory>
```

</hibernate-configuration>

Configuring NHibernate is then as simple as

```
ISessionFactory sf = new
Configuration().Configure().BuildSessionFactory();
```

You can pick a different XML configuration file using

```
ISessionFactory sf = new Configuration()
   .Configure("/path/to/config.cfg.xml")
   .BuildSessionFactory();
```

Capítulo 4. Classes Persistentes

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Capítulo 4. Classes Persistentes

Classes persistentes são classes em uma aplicação que implementam as entidades do negócio (Exemplo: Cliente e Pedido numa aplicação de E-comerce). As classes persistentes têm, como o nome sugere, instâncias transientes e também instâncias persistentes gravadas em banco de dados.

O NHibernate trabalha melhor se essas classes seguirem regras simples, também conhecidas como modelo de programação "Plain Old CLR Object" (POCO).

Um exemplo simples de POCO

Muitas aplicações .NET requerem uma classe persistente para representar felinos.

```
using System;
using Iesi.Collections;

namespace Eg
{
   public class Cat
   {
       private long id; // identifier
       private string name;
       private DateTime birthdate;
       private Cat mate;
       private ISet kittens
       private Color color;
       private char sex;
       private float weight;

       public virtual long Id
       {
           get { return id; }
           set { id = value; }
       }

       public virtual string Name
```

```
        {
            get { return name; }
            set { name = value; }
        }

        public virtual Cat Mate
        {
            get { return mate; }
            set { mate = value; }
        }

        public virtual DateTime Birthdate
        {
            get { return birthdate; }
            set { birthdate = value; }
        }

        public virtual float Weight
        {
            get { return weight; }
            set { weight = value; }
        }

        public virtual Color Color
        {
            get { return color; }
            set { color = value; }
        }

        public virtual ISet Kittens
        {
            get { return kittens; }
            set { kittens = value; }
        }

        // AddKitten not needed by NHibernate
        public virtual void AddKitten(Cat kitten)
        {
            kittens.Add(kitten);
        }

        public virtual char Sex
        {
            get { return sex; }
            set { sex = value; }
        }
    }
}
```

Há quatro regras principais para seguir aqui:

Declare accessores e modificadores para os campos persistentes

"Cat" declara métodos acessores para todos os seus campos persistentes. Muitas outras ferramentas ORM persistem diretamente variáveis de instância. Nós acreditamos que é bem melhor desacoplar esses detalhes de implementação do mecanismo de persistência. O NHibernate persiste propriedades usando os seus métodos: "get" e "set".

As propriedades não precisam ser declaradas como públicas - O NHibernate pode persistir uma propriedade com as seguintes visibilidades: internal, protected, protected internal ou private.

Implemente um construtor "default"

"Cat" tem um construtor "default" implícito (sem argumentos). Todas as classes persistentes devem ter um construtor "default" (que pode ser não publico) de forma que o NHibernate possa instanciá-los usando Activator.CreateInstance().

Disponibilize uma propriedade identificadora (Opcional)

"Cat" tem uma propriedade chamada Id. Esta propriedade guarda a coluna que é a chave primária de uma tabela no banco de dados. A propriedade poderia ter sido chamada de qualque coisa, e o seu tipo poderia ser qualquer tipo primitivo, string ou System.DateTime. (Se a sua tabela debanco de dados legado possuir uma chave composta, você pode até usar uma classe customizada com propriedades representando esses tipos - veja a seção de identificadores compostos abaixo.)

A propriedade identificadra é opcional. Voce pode deixar que o NHibernate rastreie os identificadores de objetos internamente. Entretanto, para muitas aplicações a sua utilização é ainda boa (e muito popular) decisão de "design".

E mais, algumas funcionalidades estão disponíveis somente para classes que declarem uma propriedade identificadora:

- Cascaded updates (veja "Lifecycle Objects")
- ISession.SaveOrUpdate()

Nós recomendamos que você declare propriedades identificadoras em classes persistentes.

Prefira classes não seladas e métodos virtuais (opcional)

Uma característica central do NHibernate, "proxies", depende da classe persistente ter sido declarada como não selada e todos os seus métodos públicos e eventos terem sido declarados como virtuais. Outra possibilidade é para o caso da classe implementar uma interface que declara todos os seus membros como públicos.

Você pode persistir classes seladas que não implementem uma interface e não tenham membros virtuais com o NHibernate, mas você não será capaz de usar os "proxies" - o que irá limitar as suas opções de ajuste de performance.

Implementando herança

A subclasse deve também observar a primeira e segunda regra. Ela herda a propriedade identificadora de "Cat".

using System;

```
namespace Eg
{
   public class DomesticCat : Cat
   {
       private string name;
       public virtual string Name
       {
           get { return name; }
           set { name = value; }
       }
   }
}
```

Implementnado Equals() e GetHashCode()

Voce deve sobreescrever os métodos "Equals()" e "GetHashCode()" se voce quiser misturar objetos de classes persistentes (Em um ISet, por exemplo).

Isso apenas se aplica caso esses objetos forem carregados em ISessions diferentes, uma vez que o NHibernate só garante a identidade (a == b, implementação default do Equals()) dentro de uma única iSession!

Mesmo que os dois objetos sejam a mesma linha do banco de dados (Eles tenham o mesmo valor de chave primária como identificador), nós não podemos garantir que eles sejam a mesma instância de objeto fora de um contexto particular de ISession.

A maneira mais óbvia seria implementar "Equals()/GetHashCode()" comparando o valor do identificador nos dois objetos. Caso eles fossem o mesmo, os dois deveriam ser a mesma linha do banco de dados, eles seriam, dessa forma, iguais (Caso eles fossem adicionados em um ISet, nós iríamos ter apenas um elemento no ISet). Infelizmente, nós não podemos utilizar essa abordagem. O NHibernate só irá assinalar valores aos identificadores que já foram persistidos, um novo objeto criado não irá ter um valor no identificador! Nós recomendamos implementar Equals() e GetHashCode segundo chaves de igualdade referentes ao negócio.

Igualdade de chave de negócio significa que o método Equals() deve comparar apenas as propriedades que formem a chave do negócio, a chave que identificaria a nossa instância no mundo real (uma chave candidata natural):

public class Cat {

```
   ...
   public override bool Equals(object other)
   {
       if (this == other) return true;

       Cat cat = other as Cat;
       if (cat == null) return false; // null or not a cat
       if (Name != cat.Name) return false;
       if (!Birthday.Equals(cat.Birthday)) return false;
       return true;
   }
   public override int GetHashCode()
   {
       unchecked
```

```
        {
            int result;
            result = Name.GetHashCode();
            result = 29 * result + Birthday.GetHashCode();
            return result;
        }
    }

}
```

Tenha em mente que a nossa chave candidata (neste caso a composição do nome e data de aniversário) tem que ser válida para uma operação particular de comparação (talvez até mesmo para um simples caso de uso). Nós não precisamos do critéio de estabilidade que normalmente aplicamos a uma chave primária de verdade!

Lifecycle Callbacks

Optionally, a persistent class might implement the interface ILifecycle which provides some callbacks that allow the persistent object to perform necessary initialization/cleanup after save or load and before deletion or update.

The NHibernate IInterceptor offers a less intrusive alternative, however.

public interface ILifecycle { (1)

```
        LifecycleVeto OnSave(ISession s);
(2)
        LifecycleVeto OnUpdate(ISession s);
(3)
        LifecycleVeto OnDelete(ISession s);
(4)
        void OnLoad(ISession s, object id);

}
```

(1)

OnSave - called just before the object is saved or inserted (2)

OnUpdate - called just before an object is updated (when the object is passed to ISession.Update()) (3)

OnDelete - called just before an object is deleted (4)

OnLoad - called just after an object is loaded

OnSave(), OnDelete() and OnUpdate() may be used to cascade saves and deletions of dependent objects. This is an alternative to declaring cascaded operations in the mapping file. OnLoad() may be used to initialize transient properties of the object from its persistent state. It may not be used to load dependent objects since the ISession interface may not be invoked from inside this method. A further intended usage of

OnLoad(), OnSave() and OnUpdate() is to store a reference to the current ISession for later use.

Note that OnUpdate() is not called every time the object's persistent state is updated. It is called only when a transient object is passed to ISession.Update().

If OnSave(), OnUpdate() or OnDelete() return LifecycleVeto.Veto, the operation is silently vetoed. If a CallbackException is thrown, the operation is vetoed and the exception is passed back to the application.

Note that OnSave() is called after an identifier is assigned to the object, except when native key generation is used.

IValidatable callback

If the persistent class needs to check invariants before its state is persisted, it may implement the following interface:

public interface IValidatable {

```
        void Validate();
```

}

The object should throw a ValidationFailure if an invariant was violated. An instance of Validatable should not change its state from inside Validate().

Unlike the callback methods of the ILifecycle interface, Validate() might be called at unpredictable times. The application should not rely upon calls to Validate() for business functionality.

Chapter 5. Basic O/R Mapping

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Capítulo 5. Mapeamento Básico de O/R

Mapping declaration

Object/relational mappings are defined in an XML document. The mapping document is designed to be readable and hand-editable. The mapping language is object-centric, meaning that mappings are constructed around persistent class declarations, not table declarations.

Note that, even though many NHibernate users choose to define XML mappings by hand, a number of tools exist to generate the mapping document, including

NHibernate.Mapping.Attributes library and various template-based code generators (CodeSmith, MyGeneration).

Let's kick off with an example mapping:

<?xml version="1.0"?> <hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Eg"

```
namespace="Eg">
    <class name="Cat" table="CATS" discriminator-value="C">
            <id name="Id" column="uid" type="Int64">
                    <generator class="hilo"/>
            </id>
            <discriminator column="subclass" type="Char"/>
            <property name="BirthDate" type="Date"/>
            <property name="Color" not-null="true"/>
            <property name="Sex" not-null="true" update="false"/>
            <property name="Weight"/>
            <many-to-one name="Mate" column="mate_id"/>
            <set name="Kittens">
                    <key column="mother_id"/>
                    <one-to-many class="Cat"/>
            </set>
            <subclass name="DomesticCat" discriminator-value="D">
                    <property name="Name" type="String"/>
            </subclass>
    </class>
    <class name="Dog">
    </class>
```

</hibernate-mapping>

We will now discuss the content of the mapping document. We will only describe the document elements and attributes that are used by NHibernate at runtime. The mapping document also contains some extra optional attributes and elements that affect the database schemas exported by the schema export tool. (For example the not-null attribute.)

XML Namespace

All XML mappings should declare the XML namespace shown. The actual schema definition may be found in the src\nhibernate-mapping.xsd file in the NHibernate distribution.

Tip: to enable IntelliSense for mapping and configuration files, copy the appropriate .xsd files to <VS 2005 installation directory>\Xml\Schemas .

hibernate-mapping

This element has several optional attributes. The schema attribute specifies that tables referred to by this mapping belong to the named schema. If specified, tablenames will be qualified by the given schema name. If missing, tablenames will be unqualified. The default-cascade attribute specifies what cascade style should be assumed for properties

and collections which do not specify a cascade attribute. The auto-import attribute lets us use unqualified class names in the query language, by default. The assembly and namespace attributes specify the assembly where persistent classes are located and the namespace they are declared in.

<hibernate-mapping (1)

```
        schema="schemaName"                         (2)
        default-cascade="none|save-update"          (3)
        auto-import="true|false"                    (4)
        assembly="Eg"                               (5)
        namespace="Eg"
/>
```

(1)

schema (optional): The name of a database schema. (2)

default-cascade (optional - defaults to none): A default cascade style. (3)

auto-import (optional - defaults to true): Specifies whether we can use unqualified class names (of classes in this mapping) in the query language. (4)(5)

assembly and namespace(optional): Specify assembly and namespace to assume for unqualified class names in the mapping document.

If you are not using assembly and namespace attributes, you have to specify fully-qualified class names, including the name of the assembly that classes are declared in.

If you have two persistent classes with the same (unqualified) name, you should set auto-import="false". NHibernate will throw an exception if you attempt to assign two classes to the same "imported" name.

class

You may declare a persistent class using the class element:

<class (1)

```
        name="ClassName"                            (2)
        table="tableName"                           (3)
        discriminator-value="discriminator_value"   (4)
        mutable="true|false"                        (5)
        schema="owner"                              (6)
        proxy="ProxyInterface"                      (7)
        dynamic-update="true|false"                 (8)
        dynamic-insert="true|false"                 (9)
        select-before-update="true|false"           (10)
        polymorphism="implicit|explicit"            (11)
        where="arbitrary sql where condition"       (12)
        persister="PersisterClass"                  (13)
        batch-size="N"                              (14)
        optimistic-lock="none|version|dirty|all"    (15)
        lazy="true|false"                           (16)
```

```
        abstract="true|false"
```

/>

(1)

name: The fully qualified .NET class name of the persistent class (or interface), including its assembly name. (2)

table(optional - defaults to the unqualified class name): The name of its database table. (3)

discriminator-value (optional - defaults to the class name): A value that distiguishes individual subclasses, used for polymorphic behaviour. Acceptable values include null and not null. (4)

mutable (optional, defaults to true): Specifies that instances of the class are (not) mutable. (5)

schema (optional): Override the schema name specified by the root <hibernate-mapping> element. (6)

proxy (optional): Specifies an interface to use for lazy initializing proxies. You may specify the name of the class itself. (7)

dynamic-update (optional, defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed. (8)

dynamic-insert (optional, defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null. (9)

select-before-update (optional, defaults to false): Specifies that NHibernate should never perform an SQL UPDATE unless it is certain that an object is actually modified. In certain cases (actually, only when a transient object has been associated with a new session using update()), this means that NHibernate will perform an extra SQL SELECT to determine if an UPDATE is actually required. (10)

polymorphism (optional, defaults to implicit): Determines whether implicit or explicit query polymorphism is used. (11)

where (optional) specify an arbitrary SQL WHERE condition to be used when retrieving objects of this class (12)

persister (optional): Specifies a custom IClassPersister. (13)

batch-size (optional, defaults to 1) specify a "batch size" for fetching instances of this class by identifier. (14)

optimistic-lock (optional, defaults to version): Determines the optimistic locking strategy. (15)

lazy (optional): Lazy fetching may be completely disabled by setting lazy="false". (16)

abstract (optional): Used to mark abstract superclasses in <union-subclass> hierarchies.

It is perfectly acceptable for the named persistent class to be an interface. You would then declare implementing classes of that interface using the <subclass> element. You may persist any inner class. You should specify the class name using the standard form ie. Eg.Foo+Bar, Eg. Due to an HQL parser limitation inner classes can not be used in queries in NHibernate 1.0.

Immutable classes, mutable="false", may not be updated or deleted by the application. This allows NHibernate to make some minor performance optimizations.

The optional proxy attribute enables lazy initialization of persistent instances of the class. NHibernate will initially return proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked. See "Proxies for Lazy Initialization" below.

Implicit polymorphism means that instances of the class will be returned by a query that names any superclass or implemented interface or the class and that instances of any subclass of the class will be returned by a query that names the class itself. Explicit polymorphism means that class instances will be returned only be queries that explicitly name that class and that queries that name the class will return only instances of subclasses mapped inside this <class> declaration as a <subclass> or <joined-subclass>. For most purposes the default, polymorphism="implicit", is appropriate. Explicit polymorphism is useful when two different classes are mapped to the same table (this allows a "lightweight" class that contains a subset of the table columns).

The persister attribute lets you customize the persistence strategy used for the class. You may, for example, specify your own subclass of NHibernate.Persister.EntityPersister or you might even provide a completely new implementation of the interface NHibernate.Persister.IClassPersister that implements persistence via, for example, stored procedure calls, serialization to flat files or LDAP. See NHibernate.DomainModel.CustomPersister for a simple example (of "persistence" to a Hashtable).

Note that the dynamic-update and dynamic-insert settings are not inherited by subclasses and so may also be specified on the <subclass> or <joined-subclass> elements. These settings may increase performance in some cases, but might actually decrease performance in others. Use judiciously.

Use of select-before-update will usually decrease performance. It is very useful to prevent a database update trigger being called unnecessarily.

If you enable dynamic-update, you will have a choice of optimistic locking strategies:

```
*
  version check the version/timestamp columns
*
  all check all columns
*
```

```
    dirty check the changed columns
*
    none do not use optimistic locking
```

We very strongly recommend that you use version/timestamp columns for optimistic locking with NHibernate. This is the optimal strategy with respect to performance and is the only strategy that correctly handles modifications made outside of the session (ie. when ISession.Update() is used). Keep in mind that a version or timestamp property should never be null, no matter what unsaved-value strategy, or an instance will be detected as transient.

Beginning with NHibernate 1.2.0, version numbers start with 1, not 0 as in previous versions. This was done to allow using 0 as unsaved-value for the version property.

id

Mapped classes must declare the primary key column of the database table. Most classes will also have a property holding the unique identifier of an instance. The <id> element defines the mapping from that property to the primary key column.

<id (1)

```
        name="PropertyName"                      (2)
        type="typename"                          (3)
        column="column_name"                     (4)
        unsaved-value="any|none|null|id_value"   (5)
        access="field|property|nosetter|ClassName">
        <generator class="generatorClass"/>
```

</id>

(1)

name (optional): The name of the identifier property. (2)

type (optional): A name that indicates the NHibernate type. (3)

column (optional - defaults to the property name): The name of the primary key column. (4)

unsaved-value (optional - defaults to a "sensible" value): An identifier property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session. (5)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value.

If the name attribute is missing, it is assumed that the class has no identifier property.

The unsaved-value attribute is almost never needed in NHibernate 1.0.

There is an alternative <composite-id> declaration to allow access to legacy data with composite keys. We strongly discourage its use for anything else. 5.1.4.1. generator

The required <generator> child element names a .NET class used to generate unique identifiers for instances of the persistent class. If any parameters are required to configure or initialize the generator instance, they are passed using the <param> element.

<id name="Id" type="Int64" column="uid" unsaved-value="0">

```
<generator class="NHibernate.Id.TableHiLoGenerator">
        <param name="table">uid_table</param>
        <param name="column">next_hi_value_column</param>
</generator>
```

</id>

All generators implement the interface NHibernate.Id.IIdentifierGenerator. This is a very simple interface; some applications may choose to provide their own specialized implementations. However, NHibernate provides a range of built-in implementations. There are shortcut names for the built-in generators:

increment

generates identifiers of any integral type that are unique only when no other process is inserting data into the same table. Do not use in a cluster.

identity

supports identity columns in DB2, MySQL, MS SQL Server and Sybase. The identifier returned by the database is converted to the property type using Convert.ChangeType. Any integral property type is thus supported.

sequence

uses a sequence in DB2, PostgreSQL, Oracle or a generator in Firebird. The identifier returned by the database is converted to the property type using Convert.ChangeType. Any integral property type is thus supported.

hilo

uses a hi/lo algorithm to efficiently generate identifiers of any integral type, given a table and column (by default hibernate_unique_key and next_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database. Do not use this generator with a user-supplied connection.

You can use the "where" parameter to specify the row to use in a table. This is useful if you want to use a single tabel for your identifiers, with different rows for each table.

seqhilo

uses a hi/lo algorithm to efficiently generate identifiers of any integral type, given a named database sequence.

uuid.hex

uses System.Guid and its ToString(string format) method to generate identifiers of type string. The length of the string returned depends on the configured format.

uuid.string

uses a new System.Guid to create a byte[] that is converted to a string.

guid

uses a new System.Guid as the identifier.

guid.comb

uses the algorithm to generate a new System.Guid described by Jimmy Nilsson in the article http://www.informit.com/articles/article.asp?p=25862.

native

picks identity, sequence or hilo depending upon the capabilities of the underlying database.

assigned

lets the application to assign an identifier to the object before Save() is called.

foreign

uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.

5.1.4.2. Hi/Lo Algorithm

The hilo and seqhilo generators provide two alternate implementations of the hi/lo algorithm, a favorite approach to identifier generation. The first implementation requires a "special" database table to hold the next available "hi" value. The second uses an Oracle-style sequence (where supported).

<id name="Id" type="Int64" column="cat_id">

```
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
```

```
        </generator>
```

</id>

<id name="Id" type="Int64" column="cat_id">

```
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
```

</id>

Unfortunately, you can't use hilo when supplying your own IDbConnection to NHibernate. NHibernate must be able to fetch the "hi" value in a new transaction.

UUID Hex Algorithm

<id name="Id" type="String" column="cat_id">

```
        <generator class="uuid.hex">
            <param name="format">format_value</param>
            <param name="seperator">seperator_value</param>
        </generator>
```

</id>

The UUID is generated by calling Guid.NewGuid().ToString(format). The valid values for format are described in the MSDN documentation. The default seperator is - and should rarely be modified. The format determines if the configured seperator can replace the default seperator used by the format.

UUID String Algorithm

The UUID is generated by calling Guid.NewGuid().ToByteArray() and then converting the byte[] into a char[]. The char[] is returned as a String consisting of 16 characters.

GUID Algorithms

The guid identifier is generated by calling Guid.NewGuid(). To address some of the performance concerns with using Guids as primary keys, foreign keys, and as part of indexes with MS SQL the guid.comb can be used. The benefit of using the guid.comb with other databases that support GUIDs has not been measured.

Identity columns and Sequences

For databases which support identity columns (DB2, MySQL, Sybase, MS SQL), you may use identity key generation. For databases that support sequences (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) you may use sequence style key generation. Both these strategies require two SQL queries to insert a new object.

<id name="Id" type="Int64" column="uid">

```
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
```

</id>

<id name="Id" type="Int64" column="uid" unsaved-value="0">

```
        <generator class="identity"/>
```

</id>

For cross-platform development, the native strategy will choose from the identity, sequence and hilo strategies, dependent upon the capabilities of the underlying database.

Assigned Identifiers

If you want the application to assign identifiers (as opposed to having NHibernate generate them), you may use the assigned generator. This special generator will use the identifier value already assigned to the object's identifier property. Be very careful when using this feature to assign keys with business meaning (almost always a terrible design decision).

Due to its inherent nature, entities that use this generator cannot be saved via the ISession's SaveOrUpdate() method. Instead you have to explicitly specify to NHibernate if the object should be saved or updated by calling either the Save() or Update() method of the ISession.

composite-id

<composite-id

```
        name="PropertyName"
        class="ClassName"
        unsaved-value="any|none"
        access="field|property|nosetter|ClassName">
        <key-property name="PropertyName" type="typename"
column="column_name"/>
        <key-many-to-one name="PropertyName class="ClassName"
column="column_name"/>
        ......
```

</composite-id>

For a table with a composite key, you may map multiple properties of the class as identifier properties. The <composite-id> element accepts <key-property> property mappings and <key-many-to-one> mappings as child elements.

<composite-id>

```
        <key-property name="MedicareNumber"/>
        <key-property name="Dependent"/>
```

</composite-id>

Your persistent class must override Equals() and GetHashCode() to implement composite identifier equality. It must also be Serializable.

Unfortunately, this approach to composite identifiers means that a persistent object is its own identifier. There is no convenient "handle" other than the object itself. You must instantiate an instance of the persistent class itself and populate its identifier properties before you can load() the persistent state associated with a composite key. We will describe a much more convenient approach where the composite identifier is implemented as a seperate class in Section 7.4, “Components as composite identifiers”. The attributes described below apply only to this alternative approach:

```
   *
     name (optional, required for this approach): A property of
component type that holds the composite identifier (see next section).
   *
     access (optional - defaults to property): The strategy NHibernate
should use for accessing the property value.
   *
     class (optional - defaults to the property type determined by
reflection): The component class used as a composite identifier (see
next section).
```

5.1.6. discriminator

The <discriminator> element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy and declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. A restricted set of types may be used: String, Char, Int32, Byte, Short, Boolean, YesNo, TrueFalse.

<discriminator

```
        column="discriminator_column"  (1)
        type="discriminator_type"      (2)
        force="true|false"             (3)
        insert="true|false"            (4)
        formula="arbitrary SQL expressi(5)on"
```

/>

(1)

column (optional - defaults to class) the name of the discriminator column. (2)

type (optional - defaults to String) a name that indicates the NHibernate type (3)

force (optional - defaults to false) "force" NHibernate to specify allowed discriminator values even when retrieving all instances of the root class. (4)

insert (optional - defaults to true) set this to false if your discriminator column is also part of a mapped composite identifier. (5)

formula (optional) an arbitrary SQL expression that is executed when a type has to be evaluated. Allows content-based discrimination.

Actual values of the discriminator column are specified by the discriminator-value attribute of the <class> and <subclass> elements.

The force attribute is (only) useful if the table contains rows with "extra" discriminator values that are not mapped to a persistent class. This will not usually be the case.

Using the formula attribute you can declare an arbitrary SQL expression that will be used to evaluate the type of a row:

<discriminator

```
formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
type="Int32"/>
```

version (optional)

The <version> element is optional and indicates that the table contains versioned data. This is particularly useful if you plan to use long transactions (see below).

<version

```
column="version_column"                              (1)
name="PropertyName"                                  (2)
type="typename"                                      (3)
access="field|property|nosetter|ClassName"           (4)
unsaved-value="null|negative|undefined|value"        (5)
generated="never|always"                             (6)
```

/>

(1)

column (optional - defaults to the property name): The name of the column holding the version number. (2)

name: The name of a property of the persistent class. (3)

type (optional - defaults to Int32): The type of the version number. (4)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (5)

unsaved-value (optional - defaults to a "sensible" value): A version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from

transient instances that were saved or loaded in a previous session. (undefined specifies that the identifier property value should be used.) (6)

generated (optional - defaults to never): Specifies that this version property value is actually generated by the database. See the discussion of Section 5.5, “Generated Properties”.

Version numbers may be of type Int64, Int32, Int16, Ticks, Timestamp, or TimeSpan (or their nullable counterparts in .NET 2.0). 5.1.8. timestamp (optional)

The optional <timestamp> element indicates that the table contains timestamped data. This is intended as an alternative to versioning. Timestamps are by nature a less safe implementation of optimistic locking. However, sometimes the application might use the timestamps in other ways.

<timestamp

```
column="timestamp_column"           (1)
name="PropertyName"                 (2)
access="field|property|nosetter|Clas(3)sName"
unsaved-value="null|undefined|value"(4)
generated="never|always"            (5)
```

/>

(1)

column (optional - defaults to the property name): The name of a column holding the timestamp. (2)

name: The name of a property of .NET type DateTime of the persistent class. (3)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (4)

unsaved-value (optional - defaults to null): A timestamp property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session. (undefined specifies that the identifier property value should be used.) (5)

generated (optional - defaults to never): Specifies that this timestamp property value is actually generated by the database. See the discussion of Section 5.5, “Generated Properties”.

Note that <timestamp> is equivalent to <version type="timestamp">.

property

The <property> element declares a persistent property of the class.

<property

```
      name="propertyName"                 (1)
      column="column_name"                (2)
      type="typename"                     (3)
      update="true|false"                 (4)
      insert="true|false"                 (4)
      formula="arbitrary SQL expression"  (5)
      access="field|property|ClassName"   (6)
      optimistic-lock="true|false"        (7)
      generated="never|insert|always"     (8)
```

/>

(1)

name: the name of the property of your class. (2)

column (optional - defaults to the property name): the name of the mapped database table column. (3)

type (optional): a name that indicates the NHibernate type. (4)

update, insert (optional - defaults to true) : specifies that the mapped columns should be included in SQL UPDATE and/or INSERT statements. Setting both to false allows a pure "derived" property whose value is initialized from some other property that maps to the same column(s) or by a trigger or other application. (5)

formula (optional): an SQL expression that defines the value for a computed property. Computed properties do not have a column mapping of their own. (6)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (7)

optimistic-lock (optional - defaults to true): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty. (8)

generated (optional - defaults to never): Specifies that this property value is actually generated by the database. See the discussion of Section 5.5, “Generated Properties”.

typename could be:

```
  1.
     The name of a NHibernate basic type (eg. Int32, String, Char,
DateTime, Timestamp, Single, Byte[], Object, ...).
  2.
     The name of a .NET type with a default basic type (eg.
System.Int16, System.Single, System.Char, System.String,
System.DateTime, System.Byte[], ...).
  3.
     The name of an enumeration type (eg. Eg.Color, Eg).
  4.
     The name of a serializable .NET type.
  5.
     The class name of a custom type (eg. Illflow.Type.MyCustomType).
```

Note that you have to specify full assembly-qualified names for all except basic NHibernate types (unless you set assembly and/or namespace attributes of the <hibernate-mapping> element).

NHibernate supports .NET 2.0 Nullable types. These types are mostly treated the same as plain non-Nullable types internally. For example, a property of type Nullable<Int32> can be mapped using type="Int32" or type="System.Int32".

If you do not specify a type, NHibernate will use reflection upon the named property to take a guess at the correct NHibernate type. NHibernate will try to interpret the name of the return class of the property getter using rules 2, 3, 4 in that order. However, this is not always enough. In certain cases you will still need the type attribute. (For example, to distinguish between NHibernateUtil.DateTime and NHibernateUtil.Timestamp, or to specify a custom type.)

The access attribute lets you control how NHibernate will access the value of the property at runtime. The value of the access attribute should be text formatted as access-strategy.naming-strategy. The .naming-strategy is not always required.

Table 5.1. Access Strategies Access Strategy Name Description property

The default implementation. NHibernate uses the get/set accessors of the property. No naming strategy should be used with this access strategy because the value of the name attribute is the name of the property. field

NHibernate will access the field directly. NHibernate uses the value of the name attribute as the name of the field. This can be used when a property's getter and setter contain extra actions that you don't want to occur when NHibernate is populating or reading the object. If you want the name of the property and not the field to be what the consumers of your API use with HQL, then a naming strategy is needed. nosetter

NHibernate will access the field directly when setting the value and will use the Property when getting the value. This can be used when a property only exposes a get accessor because the consumers of your API can't change the value directly. A naming strategy is required because NHibernate uses the value of the name attribute as the property name and needs to be told what the name of the field is. ClassName

If NHibernate's built in access strategies are not what is needed for your situation then you can build your own by implementing the interface NHibernate.Property.IPropertyAccessor. The value of the access attribute should be an assembly-qualified name that can be loaded with Activator.CreateInstance(string assemblyQualifiedName).

Table 5.2. Naming Strategies Naming Strategy Name Description camelcase

The name attribute is converted to camel case to find the field. <property name="Foo" ... > uses the field foo. camelcase-underscore

The name attribute is converted to camel case and prefixed with an underscore to find the field. <property name="Foo" ... > uses the field _foo. lowercase

The name attribute is converted to lower case to find the Field. <property name="FooBar" ... > uses the field foobar. lowercase-underscore

The name attribute is converted to lower case and prefixed with an underscore to find the Field. <property name="FooBar" ... > uses the field _foobar. pascalcase-underscore

The name attribute is prefixed with an underscore to find the field. <property name="Foo" ... > uses the field _Foo. pascalcase-m

The name attribute is prefixed with the character m to find the field. <property name="Foo" ... > uses the field mFoo. pascalcase-m-underscore

The name attribute is prefixed with the character m and an underscore to find the field. <property name="Foo" ... > uses the field m_Foo.

many-to-one

An ordinary association to another persistent class is declared using a many-to-one element. The relational model is a many-to-one association. (It's really just an object reference.)

<many-to-one

```
name="PropertyName"                                (1)
column="column_name"                               (2)
class="ClassName"                                  (3)
cascade="all|none|save-update|delete"              (4)
fetch="join|select"                                (5)
update="true|false"                                (6)
insert="true|false"                                (6)
property-ref="PropertyNameFromAssociatedClass"     (7)
access="field|property|nosetter|ClassName"         (8)
unique="true|false"                                (9)
optimistic-lock="true|false"                       (10)
not-found="ignore|exception"                       (11)
```

/>

(1)

name: The name of the property. (2)

column (optional): The name of the column. (3)

class (optional - defaults to the property type determined by reflection): The name of the associated class. (4)

cascade (optional): Specifies which operations should be cascaded from the parent object to the associated object. (5)

fetch (optional - defaults to select): Chooses between outer-join fetching or sequential select fetching. (6)

update, insert (optional - defaults to true) specifies that the mapped columns should be included in SQL UPDATE and/or INSERT statements. Setting both to false allows a pure "derived" association whose value is initialized from some other property that maps to the same colum(s) or by a trigger or other application. (7)

property-ref: (optional) The name of a property of the associated class that is joined to this foreign key. If not specified, the primary key of the associated class is used. (8)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (9)

unique (optional): Enable the DDL generation of a unique constraint for the foreign-key column. (10)

optimistic-lock (optional - defaults to true): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, dertermines if a version increment should occur when this property is dirty. (11)

not-found (optional - defaults to exception): Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association.

The cascade attribute permits the following values: all, save-update, delete, none. Setting a value other than none will propagate certain operations to the associated (child) object. See "Lifecycle Objects" below.

The fetch attribute accepts two different values:

```
*
  join Fetch the association using an outer join
*
  select Fetch the association using a separate query
```

A typical many-to-one declaration looks as simple as

<many-to-one name="product" class="Product" column="PRODUCT_ID"/>

The property-ref attribute should only be used for mapping legacy data where a foreign key refers to a unique key of the associated table other than the primary key. This is an ugly relational model. For example, suppose the Product class had a unique serial number, that is not the primary key. (The unique attribute controls NHibernate's DDL generation with the SchemaExport tool.)

<property name="serialNumber" unique="true" type="string"
column="SERIAL_NUMBER"/>

Then the mapping for OrderItem might use:

<many-to-one name="product" property-ref="serialNumber"
column="PRODUCT_SERIAL_NUMBER"/>

This is certainly not encouraged, however.

one-to-one

A one-to-one association to another persistent class is declared using a one-to-one element.

<one-to-one

```
name="PropertyName"                                (1)
class="ClassName"                                  (2)
cascade="all|none|save-update|delete"              (3)
constrained="true|false"                           (4)
fetch="join|select"                                (5)
property-ref="PropertyNameFromAssociatedClass"     (6)
access="field|property|nosetter|ClassName"         (7)
```

/>

(1)

name: The name of the property. (2)

class (optional - defaults to the property type determined by reflection): The name of the associated class. (3)

cascade (optional) specifies which operations should be cascaded from the parent object to the associated object. (4)

constrained (optional) specifies that a foreign key constraint on the primary key of the mapped table references the table of the associated class. This option affects the order in which Save() and Delete() are cascaded (and is also used by the schema export tool). (5)

fetch (optional - defaults to select): Chooses between outer-join fetching or sequential select fetching. (6)

property-ref: (optional) The name of a property of the associated class that is joined to the primary key of this class. If not specified, the primary key of the associated class is used. (7)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value.

There are two varieties of one-to-one association:

* primary key associations
* unique foreign key associations

Primary key associations don't need an extra table column; if two rows are related by the association then the two table rows share the same primary key value. So if you want two objects to be related by a primary key association, you must make sure that they are assigned the same identifier value!

For a primary key association, add the following mappings to Employee and Person, respectively.

<one-to-one name="Person" class="Person"/>

<one-to-one name="Employee" class="Employee" constrained="true"/>

Now we must ensure that the primary keys of related rows in the PERSON and EMPLOYEE tables are equal. We use a special NHibernate identifier generation strategy called foreign:

<class name="Person" table="PERSON">

```
<id name="Id" column="PERSON_ID">
    <generator class="foreign">
        <param name="property">Employee</param>
    </generator>
</id>
...
<one-to-one name="Employee"
    class="Employee"
    constrained="true"/>
```

</class>

A newly saved instance of Person is then assigned the same primar key value as the Employee instance refered with the Employee property of that Person.

Alternatively, a foreign key with a unique constraint, from Employee to Person, may be expressed as:

<many-to-one name="Person" class="Person" column="PERSON_ID" unique="true"/>

And this association may be made bidirectional by adding the following to the Person mapping:

<one-to-one name="Employee" class="Employee" property-ref="Person"/>

component, dynamic-component

The <component> element maps properties of a child object to columns of the table of a parent class. Components may, in turn, declare their own properties, components or collections. See "Components" below.

<component

```
    name="PropertyName"                                 (1)
    class="ClassName"                                   (2)
    insert="true|false"                                 (3)
    upate="true|false"                                  (4)
    access="field|property|nosetter|ClassName"          (5)
    optimistic-lock="true|false"                        (6)
```

>

```
        <property ...../>
        <many-to-one .... />
        ........
```

</component>

(1)

name: The name of the property. (2)

class (optional - defaults to the property type determined by reflection): The name of the component (child) class. (3)

insert: Do the mapped columns appear in SQL INSERTs? (4)

update: Do the mapped columns appear in SQL UPDATEs? (5)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (6)

optimistic-lock (optional - defaults to true): Specifies that updates to this component do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.

The child <property> tags map properties of the child class to table columns.

The <component> element allows a <parent> subelement that maps a property of the component class as a reference back to the containing entity.

The <dynamic-component> element allows an IDictionary to be mapped as a component, where the property names refer to keys of the dictionary.

subclass

Finally, polymorphic persistence requires the declaration of each subclass of the root persistent class. For the (recommended) table-per-class-hierarchy mapping strategy, the <subclass> declaration is used.

<subclass

```
        name="ClassName"                                (1)
        discriminator-value="discriminator_value"       (2)
        proxy="ProxyInterface"                          (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false">
        <property .... />
        .....
```

</subclass>

(1)

name: The fully qualified .NET class name of the subclass, including its assembly name. (2)

discriminator-value (optional - defaults to the class name): A value that distiguishes individual subclasses. (3)

proxy (optional): Specifies a class or interface to use for lazy initializing proxies. (4)

lazy (optional, defaults to true): Setting lazy="false" disables the use of lazy fetching.

Each subclass should declare its own persistent properties and subclasses. <version> and <id> properties are assumed to be inherited from the root class. Each subclass in a hierarchy must define a unique discriminator-value. If none is specified, the fully qualified .NET class name is used.

For information about inheritance mappings, see Chapter 8, Inheritance Mapping.

joined-subclass

Alternatively, a subclass that is persisted to its own table (table-per-subclass mapping strategy) is declared using a <joined-subclass> element.

<joined-subclass

```
name="ClassName"                    (1)
proxy="ProxyInterface"              (2)
lazy="true|false"                   (3)
dynamic-update="true|false"
dynamic-insert="true|false">
<key .... >
<property .... />
.....
```

</joined-subclass>

(1)

name: The fully qualified class name of the subclass. (2)

proxy (optional): Specifies a class or interface to use for lazy initializing proxies. (3)

lazy (optional): Setting lazy="true" is a shortcut equalivalent to specifying the name of the class itself as the proxy interface.

No discriminator column is required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier using the <key> element. The mapping at the start of the chapter would be re-written as:

<?xml version="1.0"?> <hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Eg"

```
   namespace="Eg">
       <class name="Cat" table="CATS">
               <id name="Id" column="uid" type="Int64">
                       <generator class="hilo"/>
               </id>
               <property name="BirthDate" type="Date"/>
               <property name="Color" not-null="true"/>
               <property name="Sex" not-null="true"/>
               <property name="Weight"/>
               <many-to-one name="Mate"/>
               <set name="Kittens">
                       <key column="MOTHER"/>
                       <one-to-many class="Cat"/>
               </set>
               <joined-subclass name="DomesticCat"
table="DOMESTIC_CATS">
                   <key column="CAT"/>
                       <property name="Name" type="String"/>
               </joined-subclass>
       </class>
       <class name="Dog">
       </class>
```

</hibernate-mapping>

For information about inheritance mappings, see Chapter 8, Inheritance Mapping.

union-subclass

A third option is to map only the concrete classes of an inheritance hierarchy to tables, (the table-per-concrete-class strategy) where each table defines all persistent state of the class, including inherited state. In Hibernate, it is not absolutely necessary to explicitly map such inheritance hierarchies. You can simply map each class with a separate <class> declaration. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the <union-subclass> mapping.

<union-subclass

```
       name="ClassName"                    (1)
       table="tablename"                   (2)
       proxy="ProxyInterface"              (3)
       lazy="true|false"                   (4)
       dynamic-update="true|false"
       dynamic-insert="true|false"
       schema="schema"
       catalog="catalog"
       extends="SuperclassName"
       abstract="true|false"
       persister="ClassName"
       subselect="SQL expression"
       entity-name="EntityName"
       node="element-name">
       <property .... />
```

. . . . .

</union-subclass>

(1)

name: The fully qualified class name of the subclass. (2)

table: The name of the subclass table. (3)

proxy (optional): Specifies a class or interface to use for lazy initializing proxies. (4)

lazy (optional, defaults to true): Setting lazy="false" disables the use of lazy fetching.

No discriminator column or key column is required for this mapping strategy.

For information about inheritance mappings, see Chapter 8, Inheritance Mapping.

join

Using the <join> element, it is possible to map properties of one class to several tables, when there's a 1-to-1 relationship between the tables.

<join

```
table="tablename"                          (1)
schema="owner"                             (2)
fetch="join|select"                        (3)
inverse="true|false"                       (4)
optional="true|false">                     (5)
<key ... />
<property ... />
...
```

</join>

(1)

table: The name of the joined table. (2)

schema (optional): Override the schema name specified by the root <hibernate-mapping> element. (3)

fetch (optional - defaults to join): If set to join, the default, Hibernate will use an inner join to retrieve a <join> defined by a class or its superclasses and an outer join for a <join> defined by a subclass. If set to select then Hibernate will use a sequential select for a <join> defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a <join> defined by the class and its superclasses. (4)

inverse (optional - defaults to false): If enabled, Hibernate will not try to insert or update the properties defined by this join. (5)

optional (optional - defaults to false): If enabled, Hibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.

For example, the address information for a person can be mapped to a separate table (while preserving value type semantics for all properties):

<class name="Person"

```
table="PERSON">
<id name="id" column="PERSON_ID">...</id>
<join table="ADDRESS">
    <key column="ADDRESS_ID"/>
    <property name="address"/>
    <property name="zip"/>
    <property name="country"/>
</join>
...
```

This feature is often only useful for legacy data models, we recommend fewer tables than classes and a fine-grained domain model. However, it is useful for switching between inheritance mapping strategies in a single hierarchy, as explained later.

map, set, list, bag

Collections are discussed later.

import

Suppose your application has two persistent classes with the same name, and you don't want to specify the fully qualified name in NHibernate queries. Classes may be "imported" explicitly, rather than relying upon auto-import="true". You may even import classes and interfaces that are not explicitly mapped.

<import class="System.Object" rename="Universe"/>

<import

```
class="ClassName"              (1)
rename="ShortName"             (2)
```

/>

(1)

class: The fully qualified class name of any .NET class, including its assembly name. (2)

rename (optional - defaults to the unqualified class name): A name that may be used in the query language.

NHibernate Types

Entities and values

To understand the behaviour of various .NET language-level objects with respect to the persistence service, we need to classify them into two groups:

An entity exists independently of any other objects holding references to the entity. Contrast this with the usual Java model where an unreferenced object is garbage collected. Entities must be explicitly saved and deleted (except that saves and deletions may be cascaded from a parent entity to its children). This is different from the ODMG model of object persistence by reachability - and corresponds more closely to how application objects are usually used in large systems. Entities support circular and shared references. They may also be versioned.

An entity's persistent state consists of references to other entities and instances of value types. Values are primitives, collections, components and certain immutable objects. Unlike entities, values (in particular collections and components) are persisted and deleted by reachability. Since value objects (and primitives) are persisted and deleted along with their containing entity they may not be independently versioned. Values have no independent identity, so they cannot be shared by two entities or collections.

All NHibernate types except collections support null semantics if the .NET type is nullable (i.e. not derived from System.ValueType).

Up until now, we've been using the term "persistent class" to refer to entities. We will continue to do that. Strictly speaking, however, not all user-defined classes with persistent state are entities. A component is a user defined class with value semantics.

Basic value types

The basic types may be roughly categorized into three groups - System.ValueType types, System.Object types, and System.Object types for large objects. Just like the .NET Types, columns for System.ValueType types can not store null values and System.Object types can store null values.

Table 5.3. System.ValueType Mapping Types NHibernate Type .NET Type Database Type Remarks AnsiChar System.Char DbType.AnsiStringFixedLength - 1 char Boolean System.Boolean DbType.Boolean Default when no type attribute specified. Byte System.Byte DbType.Byte Default when no type attribute specified. Char System.Char DbType.StringFixedLength - 1 char Default when no type attribute specified. DateTime System.DateTime DbType.DateTime - ignores the milliseconds Default when no type attribute specified. Decimal System.Decimal DbType.Decimal Default when no type attribute specified. Double System.Double DbType.Double Default when no type attribute specified. Guid System.Guid DbType.Guid Default when no type attribute specified. Int16 System.Int16 DbType.Int16 Default when no type attribute specified. Int32 System.Int32 DbType.Int32 Default when no type

attribute specified. Int64 System.Int64 DbType.Int64 Default when no type attribute specified. PersistentEnum A System.Enum The DbType for the underlying value. Do not specify type="PersistentEnum" in the mapping. Instead specify the Assembly Qualified Name of the Enum or let NHibernate use Reflection to "guess" the Type. The UnderlyingType of the Enum is used to determine the correct DbType. Single System.Single DbType.Single Default when no type attribute specified. Ticks System.DateTime DbType.Int64 type="Ticks" must be specified. TimeSpan System.TimeSpan DbType.Int64 Default when no type attribute specified. Timestamp System.DateTime DbType.DateTime - as specific as database supports. type="Timestamp" must be specified. TrueFalse System.Boolean DbType.AnsiStringFixedLength - 1 char either 'T' or 'F' type="TrueFalse" must be specified. YesNo System.Boolean DbType.AnsiStringFixedLength - 1 char either 'Y' or 'N' type="YesNo" must be specified.

Table 5.4. System.Object Mapping Types NHibernate Type .NET Type Database Type Remarks AnsiString System.String DbType.AnsiString type="AnsiString" must be specified. CultureInfo System.Globalization.CultureInfo DbType.String - 5 chars for culture Default when no type attribute specified. Binary System.Byte[] DbType.Binary Default when no type attribute specified. Type System.Type DbType.String holding Assembly Qualified Name. Default when no type attribute specified. String System.String DbType.String Default when no type attribute specified.

Table 5.5. Large Object Mapping Types NHibernate Type .NET Type Database Type Remarks StringClob System.String DbType.String type="StringClob" must be specified. Entire field is read into memory. BinaryBlob System.Byte[] DbType.Binary type="BinaryBlob" must be specified. Entire field is read into memory. Serializable Any System.Object that is marked with SerializableAttribute. DbType.Binary type="Serializable" should be specified. This is the fallback type if no NHibernate Type can be found for the Property.

NHibernate supports some additional type names for compatibility with Hibernate (useful for those coming over from Hibernate or using some of the tools to generate hbm.xml files). A type="integer" or type="int" will map to an Int32 NHibernate type, type="short" to an Int16 NHibernateType. To see all of the conversions you can view the source of static constructor of the class NHibernate.Type.TypeFactory.

Custom value types

It is relatively easy for developers to create their own value types. For example, you might want to persist properties of type Int64 to VARCHAR columns. NHibernate does not provide a built-in type for this. But custom types are not limited to mapping a property (or collection element) to a single table column. So, for example, you might have a property Name { get; set; } of type String that is persisted to the columns FIRST_NAME, INITIAL, SURNAME.

To implement a custom type, implement either NHibernate.UserTypes.IUserType or NHibernate.UserTypes.ICompositeUserType and declare properties using the fully qualified name of the type. Check out NHibernate.DomainModel.DoubleStringType to see the kind of things that are possible.

<property name="TwoStrings" type="NHibernate.DomainModel.DoubleStringType, NHibernate.DomainModel">

```
   <column name="first_string"/>
   <column name="second_string"/>
```

</property>

Notice the use of <column> tags to map a property to multiple columns.

The ICompositeUserType, IEnhancedUserType, INullableUserType, IUserCollectionType, and IUserVersionType interfaces provide support for more specialized uses.

You may even supply parameters to an IUserType in the mapping file. To do this, your IUserType must implement the NHibernate.UserTypes.IParameterizedType interface. To supply parameters to your custom type, you can use the <type> element in your mapping files.

```
<property name="priority">
   <type name="MyCompany.UserTypes.DefaultValueIntegerType">
       <param name="default">0</param>
   </type>
</property>
```

The IUserType can now retrieve the value for the parameter named default from the IDictionary object passed to it.

Even though NHibernate's rich range of built-in types and support for components means you will very rarely need to use a custom type, it is nevertheless considered good form to use custom types for (non-entity) classes that occur frequently in your application. For example, a MonetaryAmount class is a good candidate for an ICompositeUserType, even though it could easily be mapped as a component. One motivation for this is abstraction. With a custom type, your mapping documents would be future-proofed against possible changes in your way of representing monetary values.

Any type mappings

There is one further type of property mapping. The <any> mapping element defines a polymorphic association to classes from multiple tables. This type of mapping always requires more than one column. The first column holds the type of the associated entity. The remaining columns hold the identifier. It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations. You should use this only in very special cases (eg. audit logs, user session data, etc).

The meta-type attribute lets the application specify a custom type that maps database column values to persistent classes which have identifier properties of the type specified by id-type. You must specify the mapping from values of the meta-type to class names.

```
<any name="being" id-type="Int64" meta-type="string">
   <meta-value value="TBL_ANIMAL" class="Animal"/>
   <meta-value value="TBL_HUMAN" class="Human"/>
   <meta-value value="TBL_ALIEN" class="Alien"/>
   <column name="table_name"/>
   <column name="id"/>
</any>
```

NHibernate also support meta-type="class". In this case no meta-value are required because the meta-value will be the entity-name (persistentClass.FullName).

```
<any name="being" id-type="Int64" meta-type="class">
   <column name="table_name"/>
   <column name="id"/>
 </any>
```

When you use meta-type="class" to set a parameter in a query you must use:

```
SetParameter("paramName", typeof(YourClass).FullName,
NHibernateUtil.ClassMetaType)
```

Mapping section.

<any

```
       name="propertyName"                    (1)
       id-type="idtypename"                   (2)
       meta-type="metatypename"               (3)
       cascade="cascade_style"                (4)
       access="field|property|ClassName"      (5)
       optimistic-lock="true|false"           (6)
```

>

```
       <meta-value ... />
       <meta-value ... />
       .....
       <column .... />
       <column .... />
       .....
```

</any>

(1)

name: the property name. (2)

id-type: the identifier type. (3)

meta-type (optional - defaults to string): Any type that is allowed for a discriminator mapping or "class". (4)

cascade (optional- defaults to none): the cascade style. (5)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (6)

optimistic-lock (optional - defaults to true): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, define if a version increment should occur if this property is dirty.

SQL quoted identifiers

You may force NHibernate to quote an identifier in the generated SQL by enclosing the table or column name in backticks in the mapping document. NHibernate will use the correct quotation style for the SQL Dialect (usually double quotes, but brackets for SQL Server and backticks for MySQL).

<class name="LineItem" table="`Line Item`">

```
   <id name="Id" column="`Item Id`"/><generator
class="assigned"/></id>
   <property name="ItemNumber" column="`Item #`"/>
   ...
```

</class>

Modular mapping files

It is possible to define subclass and joined-subclass mappings in seperate mapping documents, directly beneath hibernate-mapping. This allows you to extend a class hierachy just by adding a new mapping file. You must specify an extends attribute in the subclass mapping, naming a previously mapped superclass. Use of this feature makes the ordering of the mapping documents important!

<hibernate-mapping>

```
       <subclass name="Eg.Subclass.DomesticCat, Eg"
           extends="Eg.Cat, Eg" discriminator-value="D">
            <property name="name" type="string"/>
       </subclass>
```

</hibernate-mapping>

Generated Properties

Generated properties are properties which have their values generated by the database. Typically, NHibernate applications needed to Refresh objects which contain any properties for which the database was generating values. Marking properties as generated, however, lets the application delegate this responsibility to NHibernate. Essentially, whenever NHibernate issues an SQL INSERT or UPDATE for an entity which has defined generated properties, it immediately issues a select afterwards to retrieve the generated values.

Properties marked as generated must additionally be non-insertable and non-updateable. Only Section 5.1.7, “version (optional)”, Section 5.1.8, “timestamp (optional)”, and Section 5.1.9, “property” can be marked as generated.

never (the default) - means that the given property value is not generated within the database.

insert - states that the given property value is generated on insert, but is not regenerated on subsequent updates. Things like created-date would fall into this category. Note that even though Section 5.1.7, “version (optional)” and Section 5.1.8, “timestamp (optional)” properties can be marked as generated, this option is not available there...

always - states that the property value is generated both on insert and on update.

Auxiliary Database Objects

Allows CREATE and DROP of arbitrary database objects, in conjunction with NHibernate's schema evolution tools, to provide the ability to fully define a user schema within the NHibernate mapping files. Although designed specifically for creating and dropping things like triggers or stored procedures, really any SQL command that can be run via a IDbCommand.ExecuteNonQuery() method is valid here (ALTERs, INSERTS, etc). There are essentially two modes for defining auxiliary database objects.

The first mode is to explicitly list the CREATE and DROP commands out in the mapping file:

<nhibernate-mapping>

```
...
<database-object>
    <create>CREATE TRIGGER my_trigger ...</create>
    <drop>DROP TRIGGER my_trigger</drop>
</database-object>
```

</nhibernate-mapping>

The second mode is to supply a custom class which knows how to construct the CREATE and DROP commands. This custom class must implement the NHibernate.Mapping.IAuxiliaryDatabaseObject interface.

<hibernate-mapping>

```
...
<database-object>
    <definition class="MyTriggerDefinition, MyAssembly"/>
</database-object>
```

</hibernate-mapping>

You may also specify parameters to be passed to the database object:

<hibernate-mapping>

```
...
<database-object>
    <definition class="MyTriggerDefinition, MyAssembly">
        <param name="parameterName">parameterValue</param>
    </definition>
</database-object>
```

</hibernate-mapping>

NHibernate will call IAuxiliaryDatabaseObject.SetParameterValues passing it a dictionary of parameter names and values.

Additionally, these database objects can be optionally scoped such that they only apply when certain dialects are used.

<hibernate-mapping>

```
...
<database-object>
    <definition class="MyTriggerDefinition"/>
    <dialect-scope name="NHibernate.Dialect.Oracle9Dialect"/>
    <dialect-scope name="NHibernate.Dialect.OracleDialect"/>
</database-object>
```

</hibernate-mapping>

Chapter 6. Collection Mapping

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Chapter 6. Collection Mapping

Persistent Collections

NHibernate requires that persistent collection-valued fields be declared as an interface type, for example:

public class Product {

```
private string serialNumber;
private ISet parts = new HashedSet();

public ISet Parts
{
    get { return parts; }
    set { parts = value; }
}
public string SerialNumber
{
    get { return serialNumber; }
    set { serialNumber = value; }
}
```

}

The actual interface might be Iesi.Collections.ISet, System.Collections.ICollection, System.Collections.IList, System.Collections.IDictionary, System.Collections.Generic.ICollection<T>, System.Collections.Generic.IList<T>, System.Collections.Generic.IDictionary<K, V>, Iesi.Collections.Generic.ISet<T> or ... anything you like! (Where "anything you like" means you will have to write an implementation of NHibernate.UserType.IUserCollectionType.)

Notice how we initialized the instance variable with an instance of HashedSet. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent - by calling Save(), for example - NHibernate will actually replace the HashedSet with an instance of NHibernate's own implementation of ISet. Watch out for errors like this:

Cat cat = new DomesticCat(); Cat kitten = new DomesticCat(); .... ISet kittens = new HashedSet(); kittens.Add(kitten); cat.Kittens = kittens; session.Save(cat); kittens = cat.Kittens; //Okay, kittens collection is an ISet HashedSet hs = (HashedSet) cat.Kittens; //Error!

Collection instances have the usual behavior of value types. They are automatically persisted when referenced by a persistent object and automatically deleted when unreferenced. If a collection is passed from one persistent object to another, its elements might be moved from one table to another. Two entities may not share a reference to the same collection instance. Due to the underlying relational model, collection-valued properties do not support null value semantics; NHibernate does not distinguish between a null collection reference and an empty collection.

You shouldn't have to worry much about any of this. Just use NHibernate's collections the same way you use ordinary .NET collections, but make sure you understand the semantics of bidirectional associations (discussed later) before using them.

Collection instances are distinguished in the database by a foreign key to the owning entity. This foreign key is referred to as the collection key . The collection key is mapped by the <key> element.

Collections may contain almost any other NHibernate type, including all basic types, custom types, entity types and components. This is an important definition: An object in a collection can either be handled with "pass by value" semantics (it therefore fully depends on the collection owner) or it can be a reference to another entity with an own lifecycle. Collections may not contain other collections. The contained type is referred to as the collection element type. Collection elements are mapped by <element>, <composite-element>, <one-to-many>, <many-to-many> or <many-to-any>. The first two map elements with value semantics, the other three are used to map entity associations.

All collection types except ISet and bag have an index column - a column that maps to an array or IList index or IDictionary key. The index of an IDictionary may be of any basic type, an entity type or even a composite type (it may not be a collection). The index of an array or list is always of type Int32. Indexes are mapped using <index>, <index-many-to-many>, <composite-index> or <index-many-to-any>.

There are quite a range of mappings that can be generated for collections, covering many common relational models. We suggest you experiment with the schema generation tool to get a feeling for how various mapping declarations translate to database tables.

Mapping a Collection

Collections are declared by the <set>, <list>, <map>, <bag>, <array> and <primitive-array> elements. <map> is representative:

<map

```
name="propertyName"                                          (1)
table="table_name"                                           (2)
schema="schema_name"                                         (3)
lazy="true|false"                                            (4)
inverse="true|false"                                         (5)
cascade="all|none|save-update|delete|all-delete-orphan"      (6)
sort="unsorted|natural|comparatorClass"                      (7)
order-by="column_name asc|desc"                              (8)
where="arbitrary sql where condition"                        (9)
fetch="select|join"                                          (10)
batch-size="N"                                               (11)
access="field|property|ClassName"                            (12)
optimistic-lock="true|false"                                 (13)
generic="true|false"                                         (14)
```

>

```
<key .... />
<index .... />
<element .... />
```

</map>

(1)

name the collection property name (2)

table (optional - defaults to property name) the name of the collection table (not used for one-to-many associations) (3)

schema (optional) the name of a table schema to override the schema declared on the root element (4)

lazy (optional - defaults to true) may be used to disable lazy fetching and specify that the association is always eagerly fetched. (5)

inverse (optional - defaults to false) mark this collection as the "inverse" end of a bidirectional association (6)

cascade (optional - defaults to none) enable operations to cascade to child entities (7)

sort (optional) specify a sorted collection with natural sort order, or a given comparator class (8)

order-by (optional) specify a table column (or columns) that define the iteration order of the IDictionary, ISet or bag, together with an optional asc or desc (9)

where (optional) specify an arbitrary SQL WHERE condition to be used when retrieving or removing the collection (useful if the collection should contain only a subset of the available data) (10)

fetch (optional) Choose between outer-join fetching and fetching by sequential select. (11)

batch-size (optional, defaults to 1) specify a "batch size" for lazily fetching instances of this collection. (12)

access (optional - defaults to property): The strategy NHibernate should use for accessing the property value. (13)

optimistic-lock (optional - defaults to true): Species that changes to the state of the collection results in increment of the owning entity's version. (For one to many associations, it is often reasonable to disable this setting.) (14)

generic (optional): Choose between generic and non-generic collection interface. If this option is not specified, NHibernate will use reflection to choose the interface.

The mapping of an IList or array requires a seperate table column holding the array or list index (the i in foo[i]). If your relational model doesn't have an index column, e.g. if you're working with legacy data, use an unordered ISet instead. This seems to put people off who assume that IList should just be a more convenient way of accessing an

unordered collection. NHibernate collections strictly obey the actual semantics attached to the ISet, IList and IDictionary interfaces. IList elements don't just spontaneously rearrange themselves!

On the other hand, people who planned to use the IList to emulate bag semantics have a legitimate grievance here. A bag is an unordered, unindexed collection which may contain the same element multiple times. The .NET collections framework lacks an IBag interface, hence you have to emulate it with an IList. NHibernate lets you map properties of type IList or ICollection with the <bag> element. Note that bag semantics are not really part of the ICollection contract and they actually conflict with the semantics of the IList contract (however, you can sort the bag arbitrarily, discussed later in this chapter).

Note: Large NHibernate bags mapped with inverse="false" are inefficient and should be avoided; NHibernate can't create, delete or update rows individually, because there is no key that may be used to identify an individual row.

Collections of Values and Many-To-Many Associations

A collection table is required for any collection of values and any collection of references to other entities mapped as a many-to-many association (the natural semantics for a .NET collection). The table requires (foreign) key column(s), element column(s) and possibly index column(s).

The foreign key from the collection table to the table of the owning class is declared using a <key> element.

<key column="column_name"/>

(1)

column (required): The name of the foreign key column.

For indexed collections like maps and lists, we require an <index> element. For lists, this column contains sequential integers numbered from zero. Make sure that your index really starts from zero if you have to deal with legacy data. For maps, the column may contain any values of any NHibernate type.

<index

```
column="column_name"                (1)
type="typename"                     (2)
```

/>

(1)

column (required): The name of the column holding the collection index values. (2)

type (optional, defaults to Int32): The type of the collection index.

Alternatively, a map may be indexed by objects of entity type. We use the <index-many-to-many> element.

<index-many-to-many

```
        column="column_name"                (1)
        class="ClassName"                   (2)
```

/>

(1)

column (required): The name of the foreign key column for the collection index values. (2)

class (required): The entity class used as the collection index.

For a collection of values, we use the <element> tag.

<element

```
        column="column_name"                (1)
        type="typename"                     (2)
```

/>

(1)

column (required): The name of the column holding the collection element values. (2)

type (required): The type of the collection element.

A collection of entities with its own table corresponds to the relational notion of many-to-many association. A many to many association is the most natural mapping of a .NET collection but is not usually the best relational model.

<many-to-many

```
        column="column_name"                                (1)
        class="ClassName"                                   (2)
        fetch="join|select"                                 (3)
        not-found="ignore|exception"                        (4)
    />
```

(1)

column (required): The name of the element foreign key column. (2)

class (required): The name of the associated class. (3)

fetch (optional, defaults to join): enables outer-join or sequential select fetching for this association. This is a special case; for full eager fetching (in a single SELECT) of an

entity and its many-to-many relationships to other entities, you would enable join fetching not only of the collection itself, but also with this attribute on the <many-to-many> nested element. (4)

not-found (optional - defaults to exception): Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association.

Some examples, first, a set of strings:

<set name="Names" table="NAMES">

```
<key column="GROUPID"/>
<element column="NAME" type="String"/>
```

</set>

A bag containing integers (with an iteration order determined by the order-by attribute):

<bag name="Sizes" table="SIZES" order-by="SIZE ASC">

```
<key column="OWNER"/>
<element column="SIZE" type="Int32"/>
```

</bag>

An array of entities - in this case, a many to many association (note that the entities are lifecycle objects, cascade="all"):

<array name="Foos" table="BAR_FOOS" cascade="all">

```
<key column="BAR_ID"/>
<index column="I"/>
<many-to-many column="FOO_ID" class="Eg.Foo, Eg"/>
```

</array>

A map from string indices to dates:

<map name="Holidays" table="holidays" schema="dbo" order-by="hol_name asc">

```
<key column="id"/>
<index column="hol_name" type="String"/>
<element column="hol_date" type="Date"/>
```

</map>

A list of components (discussed in the next chapter):

<list name="CarComponents" table="car_components">

```
<key column="car_id"/>
<index column="posn"/>
```

```
   <composite-element class="Eg.Car.CarComponent">
           <property name="Price" type="float"/>
           <property name="Type" type="Eg.Car.ComponentType, Eg"/>
           <property name="SerialNumber" column="serial_no"
type="String"/>
   </composite-element>
```

</list>

One-To-Many Associations

A one to many association links the tables of two classes directly, with no intervening collection table. (This implements a one-to-many relational model.) This relational model loses some of the semantics of .NET collections:

```
   *
     No null values may be contained in a dictionary, set or list
   *
     An instance of the contained entity class may not belong to more
than one instance of the collection
   *
     An instance of the contained entity class may not appear at more
than one value of the collection index
```

An association from Foo to Bar requires the addition of a key column and possibly an index column to the table of the contained entity class, Bar. These columns are mapped using the <key> and <index> elements described above.

The <one-to-many> tag indicates a one to many association.

<one-to-many

```
       class="ClassName"                                    (1)
       not-found="ignore|exception"                         (2)
   />
```

(1)

class (required): The name of the associated class. (2)

not-found (optional - defaults to exception): Specifies how foreign keys that reference missing rows will be handled: ignore will treat a missing row as a null association.

Example:

<set name="Bars">

```
   <key column="foo_id"/>
   <one-to-many class="Eg.Bar, Eg"/>
```

</set>

Notice that the <one-to-many> element does not need to declare any columns. Nor is it necessary to specify the table name anywhere.

Very Important Note: If the <key> column of a <one-to-many> association is declared NOT NULL, NHibernate may cause constraint violations when it creates or updates the association. To prevent this problem, you must use a bidirectional association with the many valued end (the set or bag) marked as inverse="true". See the discussion of bidirectional associations later in this chapter. 6.5. Lazy Initialization

Collections (other than arrays) may be lazily initialized, meaning they load their state from the database only when the application needs to access it. Initialization happens transparently to the user so the application would not normally need to worry about this (in fact, transparent lazy initialization is the main reason why NHibernate needs its own collection implementations). However, if the application tries something like this:

s = sessions.OpenSession(); ITransaction tx = sessions.BeginTransaction(); User u = (User) s.Find("from User u where u.Name=?", userName, NHibernateUtil.String)[0]; IDictionary permissions = u.Permissions; tx.Commit(); s.Close();

int accessLevel = (int) permissions["accounts"]; // Error!

It could be in for a nasty surprise. Since the permissions collection was not initialized when the ISession was committed, the collection will never be able to load its state. The fix is to move the line that reads from the collection to just before the commit. (There are other more advanced ways to solve this problem, however.)

Alternatively, use a non-lazy collection. Since lazy initialization can lead to bugs like that above, non-laziness is the default. However, it is intended that lazy initialization be used for almost all collections, especially for collections of entities (for reasons of efficiency).

Exceptions that occur while lazily initializing a collection are wrapped in a LazyInitializationException.

Declare a lazy collection using the optional lazy attribute:

<set name="Names" table="NAMES" lazy="true">

```
<key column="group_id"/>
<element column="NAME" type="String"/>
```

</set>

In some application architectures, particularly where the code that accesses data using NHibernate, and the code that uses it are in different application layers, it can be a problem to ensure that the ISession is open when a collection is initialized. There are two basic ways to deal with this issue:

*
  In a web-based application, an event handler can be used to close the ISession only at the very end of a user request, once the

```
rendering of the view is complete. Of course, this places heavy
demands upon the correctness of the exception handling of your
application infrastructure. It is vitally important that the ISession
is closed and the transaction ended before returning to the user, even
when an exception occurs during rendering of the view. The event
handler has to be able to access the ISession for this approach. We
recommend that the current ISession is stored in the HttpContext.Items
collection (see chapter 1, Section 1.4, “Playing with cats”, for an
example implementation).
   *
     In an application with a seperate business tier, the business
logic must "prepare" all collections that will be needed by the web
tier before returning. This means that the business tier should load
all the data and return all the data already initialized to the
presentation/web tier that is required for a particular use case.
Usually, the application calls NHibernateUtil.Initialize() for each
collection that will be needed in the web tier (this call must occur
before the session is closed) or retrieves the collection eagerly
using a NHibernate query with a FETCH clause.
   *
     You may also attach a previously loaded object to a new ISession
with Update() or Lock() before accessing unitialized collections (or
other proxies). NHibernate can not do this automatically, as it would
introduce ad hoc transaction semantics!
```

You can use the Filter() method of the NHibernate ISession API to get the size of a collection without initializing it:

ICollection countColl = s.Filter(collection, "select count(*)"); IEnumerator countEn = countColl.GetEnumerator(); countEn.MoveNext(); int count = (int) countEn.Current;

Filter() or CreateFilter() are also used to efficiently retrieve subsets of a collection without needing to initialize the whole collection. 6.6. Sorted Collections

NHibernate supports collections implemented by System.Collections.SortedList and Iesi.Collections.SortedSet. You must specify a comparer in the mapping file:

<set name="Aliases" table="person_aliases" sort="natural">

```
   <key column="person"/>
   <element column="name" type="String"/>
```

</set>

<map name="Holidays" sort="My.Custom.HolidayComparer, MyAssembly" lazy="true">

```
   <key column="year_id"/>
   <index column="hol_name" type="String"/>
   <element column="hol_date" type="Date"/>
```

</map>

Allowed values of the sort attribute are unsorted, natural and the name of a class implementing System.Collections.IComparer.

If you want the database itself to order the collection elements use the order-by attribute of set, bag or map mappings. This performs the ordering in the SQL query, not in memory.

Setting the order-by attribute tells NHibernate to use ListDictionary or ListSet class internally for dictionaries and sets, maintaining the order of the elements. Note that lookup operations on these collections are very slow if they contain more than a few elements.

<set name="Aliases" table="person_aliases" order-by="name asc">

```
<key column="person"/>
<element column="name" type="String"/>
```

</set>

<map name="Holidays" order-by="hol_date, hol_name" lazy="true">

```
<key column="year_id"/>
<index column="hol_name" type="String"/>
<element column="hol_date type="Date"/>
```

</map>

Note that the value of the order-by attribute is an SQL ordering, not a HQL ordering!

Associations may even be sorted by some arbitrary criteria at runtime using a Filter().

sortedUsers = s.Filter(group.Users, "order by this.Name");

Using an <idbag>

If you've fully embraced our view that composite keys are a bad thing and that entities should have synthetic identifiers (surrogate keys), then you might find it a bit odd that the many to many associations and collections of values that we've shown so far all map to tables with composite keys! Now, this point is quite arguable; a pure association table doesn't seem to benefit much from a surrogate key (though a collection of composite values might). Nevertheless, NHibernate provides a feature that allows you to map many to many associations and collections of values to a table with a surrogate key.

The <idbag> element lets you map a List (or Collection) with bag semantics.

<idbag name="Lovers" table="LOVERS" lazy="true">

```
<collection-id column="ID" type="Int64">
    <generator class="hilo"/>
</collection-id>
<key column="PERSON1"/>
<many-to-many column="PERSON2" class="Eg.Person" fetch="join"/>
```

</idbag>

As you can see, an <idbag> has a synthetic id generator, just like an entity class! A different surrogate key is assigned to each collection row. NHibernate does not provide any mechanism to discover the surrogate key value of a particular row, however.

Note that the update performance of an <idbag> is much better than a regular <bag>! NHibernate can locate individual rows efficiently and update or delete them individually, just like a list, map or set.

As of version 2.0, the native identifier generation strategy is supported for <idbag> collection identifiers. 6.8. Bidirectional Associations

A bidirectional association allows navigation from both "ends" of the association. Two kinds of bidirectional association are supported:

one-to-many

```
   set or bag valued at one end, single-valued at the other
```

many-to-many

```
   set or bag valued at both ends
```

Please note that NHibernate does not support bidirectional one-to-many associations with an indexed collection (list, map or array) as the "many" end, you have to use a set or bag mapping.

You may specify a bidirectional many-to-many association simply by mapping two many-to-many associations to the same database table and declaring one end as inverse (which one is your choice). Here's an example of a bidirectional many-to-many association from a class back to itself (each category can have many items and each item can be in many categories):

<class name="NHibernate.Auction.Category, NHibernate.Auction">

```
   <id name="Id" column="ID"/>
   ...
   <bag name="Items" table="CATEGORY_ITEM" lazy="true">
       <key column="CATEGORY_ID"/>
       <many-to-many class="NHibernate.Auction.Item,
NHibernate.Auction" column="ITEM_ID"/>
   </bag>
```

</class>

<class name="NHibernate.Auction.Item, NHibernate.Auction">

```
   <id name="id" column="ID"/>
   ...
   <bag name="categories" table="CATEGORY_ITEM" inverse="true"
lazy="true">
       <key column="ITEM_ID"/>
       <many-to-many class="NHibernate.Auction.Category,
NHibernate.Auction" column="CATEGORY_ID"/>
```

```
    </bag>
```

</class>

Changes made only to the inverse end of the association are not persisted. This means that NHibernate has two representations in memory for every bidirectional association, one link from A to B and another link from B to A. This is easier to understand if you think about the .NET object model and how we create a many-to-many relationship in C#:

category.Items.Add(item); // The category now "knows" about the relationship item.Categories.Add(category); // The item now "knows" about the relationship

session.Update(item); // No effect, nothing will be saved! session.Update(category); // The relationship will be saved

The non-inverse side is used to save the in-memory representation to the database. We would get an unneccessary INSERT/UPDATE and probably even a foreign key violation if both would trigger changes! The same is of course also true for bidirectional one-to-many associations.

You may map a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end inverse="true".

<class name="Eg.Parent, Eg">

```
    <id name="Id" column="id"/>
    ....
    <set name="Children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="Eg.Child, Eg"/>
    </set>
```

</class>

<class name="Eg.Child, Eg">

```
    <id name="Id" column="id"/>
    ....
    <many-to-one name="Parent" class="Eg.Parent, Eg"
column="parent_id"/>
```

</class>

Mapping one end of an association with inverse="true" doesn't affect the operation of cascades, both are different concepts! 6.9. Ternary Associations

There are two possible approaches to mapping a ternary association. One approach is to use composite elements (discussed below). Another is to use an IDictionary with an association as its index:

<map name="Contracts" lazy="true">

```
<key column="employer_id"/>
<index-many-to-many column="employee_id" class="Employee"/>
<one-to-many column="contract_id" class="Contract"/>
```

</map>

<map name="Connections" lazy="true">

```
<key column="node1_id"/>
<index-many-to-many column="node2_id" class="Node"/>
<many-to-many column="connection_id" class="Connection"/>
```

</map>

Heterogeneous Associations

The <many-to-any> and <index-many-to-any> elements provide for true heterogeneous associations. These mapping elements work in the same way as the <any> element - and should also be used rarely, if ever. 6.11. Collection examples

The previous sections are pretty confusing. So lets look at an example. This class:

using System; using System.Collections;

namespace Eg

```
public class Parent
{
    private long id;
    private ISet children;

    public long Id
    {
        get { return id; }
        set { id = value; }
    }

    private ISet Children
    {
        get { return children; }
        set { children = value; }
    }

    ....
    ....
}
```

}

has a collection of Eg.Child instances. If each child has at most one parent, the most natural mapping is a one-to-many association:

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

```
    assembly="Eg" namespace="Eg">
    <class name="Parent">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <set name="Children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>
    <class name="Child">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <property name="Name"/>
    </class>
```

</hibernate-mapping>

This maps to the following table definitions:

create table parent (Id bigint not null primary key) create table child (Id bigint not null primary key, Name varchar(255), parent_id bigint) alter table child add constraint childfk0 (parent_id) references parent

If the parent is required, use a bidirectional one-to-many association:

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

```
    assembly="Eg" namespace="Eg">
    <class name="Parent">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <set name="Children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>
    <class name="Child">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <property name="Name"/>
        <many-to-one name="parent" class="Parent" column="parent_id"
not-null="true"/>
    </class>
```

</hibernate-mapping>

Notice the NOT NULL constraint:

create table parent (Id bigint not null primary key) create table child (Id bigint not null

```
                    primary key,
                    Name varchar(255),
                    parent_id bigint not null )
```

alter table child add constraint childfk0 (parent_id) references parent

On the other hand, if a child might have multiple parents, a many-to-many association is appropriate:

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

```
    assembly="Eg" namespace="Eg">
    <class name="Parent">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <set name="Children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>
    <class name="eg.Child">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <property name="Name"/>
    </class>
```

</hibernate-mapping>

Table definitions:

create table parent (Id bigint not null primary key) create table child (Id bigint not null primary key, name varchar(255)) create table childset (parent_id bigint not null,

```
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
```

alter table childset add constraint childsetfk0 (parent_id) references parent alter table childset add constraint childsetfk1 (child_id) references child

Chapter 7. Component Mapping

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Chapter 7. Component Mapping

The notion of a component is re-used in several different contexts, for different purposes, throughout NHibernate. ===Dependent objects

A component is a contained object that is persisted as a value type, not an entity. The term "component" refers to the object-oriented notion of composition (not to architecture-level components). For example, you might model a person like this:

public class Person {

```
private DateTime birthday;
private Name name;
private string key;

public string Key
{
    get { return key; }
    set { key = value; }
}

public DateTime Birthday
{
    get { return birthday; }
    set { birthday = value; }
}
public Name Name
{
    get { return name; }
    set { name = value; }
}
......
......
```

}

public class Name {

```
char initial;
string first;
string last;

public string First
{
```

```
        get { return first; }
        set { first = value; }
    }

    public string Last
    {
        get { return last; }
        set { last = value; }
    }
    public char Initial
    {
        get { return initial; }
        set { initial = value; }
    }

}
```

Now Name may be persisted as a component of Person. Notice that Name defines getter and setter methods for its persistent properties, but doesn't need to declare any interfaces or identifier properties.

Our NHibernate mapping would look like:

<class name="Eg.Person, Eg" table="person">

```
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="Birthday" type="date"/>
    <component name="Name" class="Eg.Name, Eg">
        <property name="Initial"/>
        <property name="First"/>
        <property name="Last"/>
    </component>
```

</class>

The person table would have the columns pid, Birthday, Initial, First and Last.

Like all value types, components do not support shared references. The null value semantics of a component are ad hoc. When reloading the containing object, NHibernate will assume that if all component columns are null, then the entire component is null. This should be okay for most purposes.

The properties of a component may be of any NHibernate type (collections, many-to-one associations, other components, etc). Nested components should not be considered an exotic usage. NHibernate is intended to support a very fine-grained object model.

The <component> element allows a <parent> subelement that maps a property of the component class as a reference back to the containing entity.

<class name="Eg.Person, Eg" table="person">

```
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
```

```
    </id>
    <property name="Birthday" type="date"/>
    <component name="Name" class="Eg.Name, Eg">
        <parent name="NamedPerson"/>
        <property name="Initial"/>
        <property name="First"/>
        <property name="Last"/>
    </component>
```

</class>

Collections of dependent objects

Collections of components are supported (eg. an array of type Name). Declare your component collection by replacing the <element> tag with a <composite-element> tag.

<set name="SomeNames" table="some_names" lazy="true">

```
    <key column="id"/>
    <composite-element class="Eg.Name, Eg">
        <property name="Initial"/>
        <property name="First"/>
        <property name="Last"/>
    </composite-element>
```

</set>

Note: if you define an ISet of composite elements, it is very important to implement Equals() and GetHashCode() correctly.

Composite elements may contain components but not collections. If your composite element itself contains components, use the <nested-composite-element> tag. This is a pretty exotic case - a collection of components which themselves have components. By this stage you should be asking yourself if a one-to-many association is more appropriate. Try remodelling the composite element as an entity - but note that even though the object model is the same, the relational model and persistence semantics are still slightly different.

Please note that a composite element mapping doesn't support null-able properties if you're using a <set>. NHibernate has to use each columns value to identify a record when deleting objects (there is no separate primary key column in the composite element table), which is not possible with null values. You have to either use only not-null properties in a composite-element or choose a <list>, <map>, <bag> or <idbag>.

A special case of a composite element is a composite element with a nested <many-to-one> element. A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class. The following is a many-to-many association from Order to Item where PurchaseDate, Price and Quantity are properties of the association:

<class name="Order" .... >

```
    ....
```

```
    <set name="PurchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="Purchase">
            <property name="PurchaseDate"/>
            <property name="Price"/>
            <property name="Quantity"/>
            <many-to-one name="Item" class="Item"/>
        </composite-element>
    </set>
```

</class>

Even ternary (or quaternary, etc) associations are possible:

<class name="Order" .... >

```
    ....
    <set name="PurchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="OrderLine">
            <many-to-one name="PurchaseDetails class="Purchase"/>
            <many-to-one name="Item" class="Item"/>
        </composite-element>
    </set>
```

</class>

Composite elements may appear in queries using the same syntax as associations to other entities.

Components as IDictionary indices

The <composite-index> element lets you map a component class as the key of an IDictionary. Make sure you override GetHashCode() and Equals() correctly on the component class.

Components as composite identifiers

You may use a component as an identifier of an entity class. Your component class must satisfy certain requirements:

```
    *
      It must be Serializable.
    *
      It must re-implement Equals() and GetHashCode(), consistently
with the database's notion of composite key equality.
```

You can't use an IIdentifierGenerator to generate composite keys. Instead the application must assign its own identifiers.

Since a composite identifier must be assigned to the object before saving it, we can't use unsaved-value of the identifier to distinguish between newly instantiated instances and instances saved in a previous session.

You may instead implement IInterceptor.IsUnsaved() if you wish to use SaveOrUpdate() or cascading save / update. As an alternative, you may also set the unsaved-value attribute on a <version> (or <timestamp>) element to specify a value that indicates a new transient instance. In this case, the version of the entity is used instead of the (assigned) identifier and you don't have to implement IInterceptor.IsUnsaved() yourself.

Use the <composite-id> tag (same attributes and elements as <component>) in place of <id> for the declaration of a composite identifier class:

<class name="Foo" table="FOOS">

```
<composite-id name="CompId" class="FooCompositeID">
    <key-property name="String"/>
    <key-property name="Short"/>
    <key-property name="Date" column="date_" type="Date"/>
</composite-id>
<property name="Name"/>
....
```

</class>

Now, any foreign keys into the table FOOS are also composite. You must declare this in your mappings for other classes. An association to Foo would be declared like this:

<many-to-one name="Foo" class="Foo">

```
<column name="foo_string"/>
<column name="foo_short"/>
<column name="foo_date"/>
```

</many-to-one>

This new <column> tag is also used by multi-column custom types. Actually it is an alternative to the column attribute everywhere. A collection with elements of type Foo would use:

<set name="Foos">

```
<key column="owner_id"/>
<many-to-many class="Foo">
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-many>
```

</set>

On the other hand, <one-to-many>, as usual, declares no columns.

If Foo itself contains collections, they will also need a composite foreign key.

<class name="Foo">

```
....
....
<set name="Dates" lazy="true">
    <key>
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </key>
    <element column="foo_date" type="Date"/>
</set>
```

</class>

Dynamic components

You may even map a property of type IDictionary:

<dynamic-component name="UserAttributes">

```
<property name="Foo" column="FOO"/>
<property name="Bar" column="BAR"/>
<many-to-one name="Baz" class="Baz" column="BAZ"/>
```

</dynamic-component>

The semantics of a <dynamic-component> mapping are identical to <component>. The advantage of this kind of mapping is the ability to determine the actual properties of the component at deployment time, just by editing the mapping document. (Runtime manipulation of the mapping document is also possible, using a DOM parser.)

Chapter 8. Inheritance Mapping

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Chapter 8. Inheritance Mapping

The Three Strategies

NHibernate supports the three basic inheritance mapping strategies.

- table per class hierarchy
- table per subclass
- table per concrete class

In addition, NHibernate supports a fourth, slightly different kind of polymorphism:

- implicit polymorphism

It is possible to use different mapping strategies for different branches of the same inheritance hierarchy, and then make use of implicit polymorphism to achieve polymorphism across the whole hierarchy. However, Hibernate does not support mixing <subclass>, and <joined-subclass> and <union-subclass> mappings under the same root <class> element. It is possible to mix together the table per hierarchy and table per subclass strategies, under the the same <class> element, by combining the <subclass> and <join> elements (see below).

It is possible to define subclass, union-subclass, and joined-subclass mappings in separate mapping documents, directly beneath hibernate-mapping. This allows you to extend a class hierachy just by adding a new mapping file. You must specify an extends attribute in the subclass mapping, naming a previously mapped superclass.

```
<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-
value="D">
        <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>
```

Table per class hierarchy

Suppose we have an interface IPayment, with implementors CreditCardPayment, CashPayment, ChequePayment. The table-per-hierarchy mapping would look like:

<class name="IPayment" table="PAYMENT">

```
<id name="Id" type="Int64" column="PAYMENT_ID">
    <generator class="native"/>
</id>
<discriminator column="PAYMENT_TYPE" type="String"/>
<property name="Amount" column="AMOUNT"/>
...
<subclass name="CreditCardPayment" discriminator-value="CREDIT">
    ...
</subclass>
<subclass name="CashPayment" discriminator-value="CASH">
    ...
</subclass>
<subclass name="ChequePayment" discriminator-value="CHEQUE">
    ...
</subclass>
```

</class>

Exactly one table is required. There is one big limitation of this mapping strategy: columns declared by the subclasses may not have NOT NULL constraints.

Table per subclass

A table-per-subclass mapping would look like:

<class name="IPayment" table="PAYMENT">

```
<id name="Id" type="Int64" column="PAYMENT_ID">
    <generator class="native"/>
</id>
<property name="Amount" column="AMOUNT"/>
...
<joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <key column="PAYMENT_ID"/>
    ...
</joined-subclass>
<joined-subclass name="CashPayment" table="CASH_PAYMENT">
    <key column="PAYMENT_ID"/>
    ...
</joined-subclass>
<joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    <key column="PAYMENT_ID"/>
    ...
</joined-subclass>
```

</class>

Four tables are required. The three subclass tables have primary key associations to the superclass table (so the relational model is actually a one-to-one association).

Table per subclass, using a discriminator

Note that NHibernate's implementation of table-per-subclass requires no discriminator column. Other object/relational mappers use a different implementation of table-per-subclass which requires a type discriminator column in the superclass table. The approach taken by NHibernate is much more difficult to implement but arguably more correct from a relational point of view. If you would like to use a discriminator column with the table per subclass strategy, you may combine the use of <subclass> and <join>, as follow:

<class name="Payment" table="PAYMENT">

```
<id name="Id" type="Int64" column="PAYMENT_ID">
    <generator class="native"/>
</id>
<discriminator column="PAYMENT_TYPE" type="string"/>
<property name="Amount" column="AMOUNT"/>
...
<subclass name="CreditCardPayment" discriminator-value="CREDIT">
    <join table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="CreditCardType" column="CCTYPE"/>
        ...
    </join>
</subclass>
<subclass name="CashPayment" discriminator-value="CASH">
    <join table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </join>
</subclass>
<subclass name="ChequePayment" discriminator-value="CHEQUE">
    <join table="CHEQUE_PAYMENT" fetch="select">
        <key column="PAYMENT_ID"/>
        ...
    </join>
</subclass>
```

</class>

The optional fetch="select" declaration tells NHibernate not to fetch the ChequePayment subclass data using an outer join when querying the superclass.

Mixing table per class hierarchy with table per subclass

You may even mix the table per hierarchy and table per subclass strategies using this approach:

<class name="Payment" table="PAYMENT">

```
<id name="Id" type="Int64" column="PAYMENT_ID">
    <generator class="native"/>
```

```
</id>
<discriminator column="PAYMENT_TYPE" type="string"/>
<property name="Amount" column="AMOUNT"/>
...
<subclass name="CreditCardPayment" discriminator-value="CREDIT">
    <join table="CREDIT_PAYMENT">
        <property name="CreditCardType" column="CCTYPE"/>
        ...
    </join>
</subclass>
<subclass name="CashPayment" discriminator-value="CASH">
    ...
</subclass>
<subclass name="ChequePayment" discriminator-value="CHEQUE">
    ...
</subclass>
```

</class>

For any of these mapping strategies, a polymorphic association to IPayment is mapped using <many-to-one>.

<many-to-one name="Payment" column="PAYMENT" class="IPayment"/>

Table per concrete class

There are two ways we could go about mapping the table per concrete class strategy. The first is to use <union-subclass>.

<class name="Payment">

```
<id name="Id" type="Int64" column="PAYMENT_ID">
    <generator class="sequence"/>
</id>
<property name="Amount" column="AMOUNT"/>
...
<union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
    <property name="CreditCardType" column="CCTYPE"/>
    ...
</union-subclass>
<union-subclass name="CashPayment" table="CASH_PAYMENT">
    ...
</union-subclass>
<union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
    ...
</union-subclass>
```

</class>

Three tables are involved for the subclasses. Each table defines columns for all properties of the class, including inherited properties.

The limitation of this approach is that if a property is mapped on the superclass, the column name must be the same on all subclass tables. (We might relax this in a future release of Hibernate.) The identity generator strategy is not allowed in union subclass

inheritance, indeed the primary key seed has to be shared accross all unioned subclasses of a hierarchy.

If your superclass is abstract, map it with abstract="true". Of course, if it is not abstract, an additional table (defaults to PAYMENT in the example above) is needed to hold instances of the superclass.

Table per concrete class, using implicit polymorphism

An alternative approach is to make use of implicit polymorphism:

<class name="CreditCardPayment" table="CREDIT_PAYMENT">

```
<id name="Id" type="Int64" column="CREDIT_PAYMENT_ID">
    <generator class="native"/>
</id>
<property name="Amount" column="CREDIT_AMOUNT"/>
...
```

</class>

<class name="CashPayment" table="CASH_PAYMENT">

```
<id name="Id" type="Int64" column="CASH_PAYMENT_ID">
    <generator class="native"/>
</id>
<property name="Amount" column="CASH_AMOUNT"/>
...
```

</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">

```
<id name="Id" type="Int64" column="CHEQUE_PAYMENT_ID">
    <generator class="native"/>
</id>
<property name="Amount" column="CHEQUE_AMOUNT"/>
...
```

</class>

Notice that nowhere do we mention the IPayment interface explicitly. Also notice that properties of IPayment are mapped in each of the subclasses. If you want to avoid duplication, consider using XML entities (e.g. [<!ENTITY allproperties SYSTEM "allproperties.xml">] in the DOCTYPE declartion and &allproperties; in the mapping).

The disadvantage of this approach is that NHibernate does not generate SQL UNIONs when performing polymorphic queries.

For this mapping strategy, a polymorphic association to IPayment is usually mapped using <any>.

<any name="Payment" meta-type="string" id-type="Int64">

```
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
```

</any>

Mixing implicit polymorphism with other inheritance mappings

There is one further thing to notice about this mapping. Since the subclasses are each mapped in their own <class> element (and since IPayment is just an interface), each of the subclasses could easily be part of another table-per-class or table-per-subclass inheritance hierarchy! (And you can still use polymorphic queries against the IPayment interface.)

<class name="CreditCardPayment" table="CREDIT_PAYMENT">

```
    <id name="Id" type="Int64" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="String"/>
    <property name="Amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
```

</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">

```
    <id name="Id" type="Int64" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="Amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="Amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
```

</class>

Once again, we don't mention IPayment explicitly. If we execute a query against the IPayment interface - for example, from IPayment - NHibernate automatically returns instances of CreditCardPayment (and its subclasses, since they also implement IPayment), CashPayment and ChequePayment but not instances of NonelectronicTransaction.

Limitations

There are certain limitations to the "implicit polymorphism" approach to the table per concrete-class mapping strategy. There are somewhat less restrictive limitations to <union-subclass> mappings.

The following table shows the limitations of table per concrete-class mappings, and of implicit polymorphism, in NHibernate.

Table 8.1. Features of inheritance mappings Inheritance strategy Polymorphic many-to-one Polymorphic one-to-one Polymorphic one-to-many Polymorphic many-to-many Polymorphic load()/get() Polymorphic queries Polymorphic joins Outer join fetching table per class-hierarchy <many-to-one> <one-to-one> <one-to-many> <many-to-many> s.Get(typeof(IPayment), id) from IPayment p from Order o join o.Payment p supported table per subclass <many-to-one> <one-to-one> <one-to-many> <many-to-many> s.Get(typeof(IPayment), id) from IPayment p from Order o join o.Payment p supported table per concrete-class (union-subclass) <many-to-one> <one-to-one> <one-to-many> (for inverse="true" only) <many-to-many> s.Get(typeof(IPayment), id) from IPayment p from Order o join o.Payment p supported table per concrete class (implicit polymorphism) <any> not supported not supported <many-to-any> use a query from IPayment p not supported not supported

Chapter 9. Manipulating Persistent Data

Tabela de conteúdo

[esconder]

Chapter 9. Manipulating Persistent Data

Creating a persistent object

An object (entity instance) is either transient or persistent with respect to a particular ISession. Newly instantiated objects are, of course, transient. The session offers services for saving (ie. persisting) transient instances:

DomesticCat fritz = new DomesticCat(); fritz.Color = Color.Ginger; fritz.Sex = 'M'; fritz.Name = "Fritz"; long generatedId = (long) sess.Save(fritz);

DomesticCat pk = new DomesticCat(); pk.Color = Color.Tabby; pk.Sex = 'F'; pk.Name = "PK"; pk.Kittens = new HashSet(); pk.AddKitten(fritz); sess.Save(pk, 1234L);

The single-argument Save() generates and assigns a unique identifier to fritz. The two-argument form attempts to persist pk using the given identifier. We generally discourage the use of the two-argument form since it may be used to create primary keys with business meaning.

Associated objects may be made persistent in any order you like unless you have a NOT NULL constraint upon a foreign key column. There is never a risk of violating foreign

key constraints. However, you might violate a NOT NULL constraint if you Save() the objects in the wrong order.

Loading an object

The Load() methods of ISession give you a way to retrieve a persistent instance if you already know its identifier. One version takes a class object and will load the state into a newly instantiated object. The second version allows you to supply an instance into which the state will be loaded. The form which takes an instance is only useful in special circumstances (DIY instance pooling etc.)

Cat fritz = (Cat) sess.Load(typeof(Cat), generatedId);

long pkId = 1234; DomesticCat pk = (DomesticCat) sess.Load(typeof(Cat), pkId);

Cat cat = new DomesticCat(); // load pk's state into cat sess.Load(cat, pkId); ISet kittens = cat.Kittens;

Note that Load() will throw an unrecoverable exception if there is no matching database row. If the class is mapped with a proxy, Load() returns an object that is an uninitialized proxy and does not actually hit the database until you invoke a method of the object. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database.

If you are not certain that a matching row exists, you should use the Get() method, which hits the database immediately and returns null if there is no matching row.

Cat cat = (Cat) sess.Get(typeof(Cat), id); if (cat==null) {

```
cat = new Cat();
sess.Save(cat, id);
```

} return cat;

You may also load an objects using an SQL SELECT ... FOR UPDATE. See the next section for a discussion of NHibernate LockModes.

Cat cat = (Cat) sess.Get(typeof(Cat), id, LockMode.Upgrade);

Note that any associated instances or contained collections are not selected FOR UPDATE.

It is possible to re-load an object and all its collections at any time, using the Refresh() method. This is useful when database triggers are used to initialize some of the properties of the object.

sess.Save(cat); sess.Flush(); //force the SQL INSERT sess.Refresh(cat); //re-read the state (after the trigger executes)

An important question usually appears at this point: How much does Hibernate load from the database and how many SQL SELECTs will it use? This depends on the fetching strategy and is explained in Section 16.1, “Fetching strategies”.

Querying

If you don't know the identifier(s) of the object(s) you are looking for, use the Find() methods of ISession. NHibernate supports a simple but powerful object oriented query language.

IList cats = sess.Find(

```
"from Cat as cat where cat.Birthdate = ?",
date,
NHibernateUtil.Date
```

);

IList mates = sess.Find(

```
"select mate from Cat as cat join cat.Mate as mate " +
"where cat.name = ?",
name,
NHibernateUtil.String
```

);

IList cats = sess.Find("from Cat as cat where cat.Mate.Birthdate is null");

IList moreCats = sess.Find(

```
"from Cat as cat where " +
"cat.Name = 'Fritz' or cat.id = ? or cat.id = ?",
new object[] { id1, id2 },
new IType[] { NHibernateUtil.Int64, NHibernateUtil.Int64 }
```

);

IList mates = sess.Find(

```
"from Cat as cat where cat.Mate = ?",
izi,
NHibernateUtil.Entity(typeof(Cat))
```

);

IList problems = sess.Find(

```
"from GoldFish as fish " +
"where fish.Birthday > fish.Deceased or fish.Birthday is null"
```

);

The second argument to Find() accepts an object or array of objects. The third argument accepts a NHibernate type or array of NHibernate types. These given types are used to bind the given objects to the ? query placeholders (which map to input parameters of an ADO.NET IDbCommand). Just as in ADO.NET, you should use this binding mechanism in preference to string manipulation.

The NHibernateUtil class defines a number of static methods and constants, providing access to most of the built-in types, as instances of NHibernate.Type.IType.

If you expect your query to return a very large number of objects, but you don't expect to use them all, you might get better performance from the Enumerable() methods, which return a System.Collections.IEnumerable. The iterator will load objects on demand, using the identifiers returned by an initial SQL query (n+1 selects total).

// fetch ids IEnumerable en = sess.Enumerable("from eg.Qux q order by q.Likeliness"); foreach (Qux qux in en) {

```
// something we couldnt express in the query
if ( qux.CalculateComplicatedAlgorithm() ) {
    // dont need to process the rest
    break;
}
```

}

The Enumerable() method also performs better if you expect that many of the objects are already loaded and cached by the session, or if the query results contain the same objects many times. (When no data is cached or repeated, Find() is almost always faster.) Heres an example of a query that should be called using Enumerable():

IEnumerable en = sess.Enumerable(

```
"select customer, product " +
"from Customer customer, " +
"Product product " +
"join customer.Purchases purchase " +
"where product = purchase.Product"
```

);

Calling the previous query using Find() would return a very large ADO.NET result set containing the same data many times.

NHibernate queries sometimes return tuples of objects, in which case each tuple is returned as an array:

IEnumerable foosAndBars = sess.Enumerable(

```
"select foo, bar from Foo foo, Bar bar " +
"where bar.Date = foo.Date"
```

); foreach (object[] tuple in foosAndBars) {

```
   Foo foo = tuple[0]; Bar bar = tuple[1];
   ....
```

}

Scalar queries

Queries may specify a property of a class in the select clause. They may even call SQL aggregate functions. Properties or aggregates are considered "scalar" results.

IEnumerable results = sess.Enumerable(

```
       "select cat.Color, min(cat.Birthdate), count(cat) from Cat cat
" +
       "group by cat.Color"
```

); foreach (object[] row in results) {

```
   Color type = (Color) row[0];
   DateTime oldest = (DateTime) row[1];
   int count = (int) row[2];
   .....
```

}

IEnumerable en = sess.Enumerable(

```
   "select cat.Type, cat.Birthdate, cat.Name from DomesticCat cat"
```

);

IList list = sess.Find(

```
   "select cat, cat.Mate.Name from DomesticCat cat"
```

);

9.3.2. The IQuery interface

If you need to specify bounds upon your result set (the maximum number of rows you want to retrieve and / or the first row you want to retrieve) you should obtain an instance of NHibernate.IQuery:

IQuery q = sess.CreateQuery("from DomesticCat cat"); q.SetFirstResult(20); q.SetMaxResults(10); IList cats = q.List();

You may even define a named query in the mapping document. (Remember to use a CDATA section if your query contains characters that could be interpreted as markup.)

<query name="Eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[

```
   from Eg.DomesticCat as cat
```

```
        where cat.Name = ?
        and cat.Weight > ?
```

]]></query>

IQuery q = sess.GetNamedQuery("Eg.DomesticCat.by.name.and.minimum.weight"); q.SetString(0, name); q.SetInt32(1, minWeight); IList cats = q.List();

The query interface supports the use of named parameters. Named parameters are identifiers of the form :name in the query string. There are methods on IQuery for binding values to named or positional parameters. NHibernate numbers parameters from zero. The advantages of named parameters are:

```
   *
     named parameters are insensitive to the order they occur in the
query string
   *
     they may occur multiple times in the same query
   *
     they are self-documenting
```

//named parameter (preferred) IQuery q = sess.CreateQuery("from DomesticCat cat where cat.Name = :name"); q.SetString("name", "Fritz"); IEnumerable cats = q.Enumerable();

//positional parameter IQuery q = sess.CreateQuery("from DomesticCat cat where cat.Name = ?"); q.SetString(0, "Izi"); IEnumerable cats = q.Enumerable();

//named parameter list IList names = new ArrayList(); names.Add("Izi"); names.Add("Fritz"); IQuery q = sess.CreateQuery("from DomesticCat cat where cat.Name in (:namesList)"); q.SetParameterList("namesList", names); IList cats = q.List();

Filtering collections

A collection filter is a special type of query that may be applied to a persistent collection or array. The query string may refer to this, meaning the current collection element.

ICollection blackKittens = session.Filter(

```
   pk.Kittens, "where this.Color = ?", Color.Black,
NHibernateUtil.Enum(typeof(Color))
```

);

The returned collection is considered a bag.

Observe that filters do not require a from clause (though they may have one if required). Filters are not limited to returning the collection elements themselves.

ICollection blackKittenMates = session.Filter(

```
   pk.Kittens, "select this.Mate where this.Color = Eg.Color.Black"
```

);

Criteria queries

HQL is extremely powerful but some people prefer to build queries dynamically, using an object oriented API, rather than embedding strings in their .NET code. For these people, NHibernate provides an intuitive ICriteria query API.

ICriteria crit = session.CreateCriteria(typeof(Cat)); crit.Add(Expression.Eq("color", Eg.Color.Black)); crit.SetMaxResults(10); IList cats = crit.List();

If you are uncomfortable with SQL-like syntax, this is perhaps the easiest way to get started with NHibernate. This API is also more extensible than HQL. Applications might provide their own implementations of the ICriterion interface.

Queries in native SQL

You may express a query in SQL, using CreateSQLQuery(). You must enclose SQL aliases in braces.

IList cats = session.CreateSQLQuery(

```
   "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
   "cat",
   typeof(Cat)
```

).List();

IList cats = session.CreateSQLQuery(

```
   "SELECT {cat}.ID AS {cat.Id}, {cat}.SEX AS {cat.Sex}, " +
          "{cat}.MATE AS {cat.Mate}, {cat}.SUBCLASS AS {cat.class},
... " +
   "FROM CAT {cat} WHERE ROWNUM<10",
   "cat",
   typeof(Cat)
```

).List()

SQL queries may contain named and positional parameters, just like NHibernate queries.

Updating objects

Updating in the same ISession

Transactional persistent instances (ie. objects loaded, saved, created or queried by the ISession) may be manipulated by the application and any changes to persistent state will be persisted when the ISession is flushed (discussed later in this chapter). So the most

straightforward way to update the state of an object is to Load() it, and then manipulate it directly, while the ISession is open:

DomesticCat cat = (DomesticCat) sess.Load(typeof(Cat), 69L); cat.Name = "PK"; sess.Flush(); // changes to cat are automatically detected and persisted

Sometimes this programming model is inefficient since it would require both an SQL SELECT (to load an object) and an SQL UPDATE (to persist its updated state) in the same session. Therefore NHibernate offers an alternate approach.

Updating detached objects

Many applications need to retrieve an object in one transaction, send it to the UI layer for manipulation, then save the changes in a new transaction. (Applications that use this kind of approach in a high-concurrency environment usually use versioned data to ensure transaction isolation.) This approach requires a slightly different programming model to the one described in the last section. NHibernate supports this model by providing the method Session.Update().

// in the first session Cat cat = (Cat) firstSession.Load(typeof(Cat), catId); Cat potentialMate = new Cat(); firstSession.Save(potentialMate);

// in a higher tier of the application cat.Mate = potentialMate;

// later, in a new session secondSession.Update(cat); // update cat secondSession.Update(mate); // update mate

If the Cat with identifier catId had already been loaded by secondSession when the application tried to update it, an exception would have been thrown.

The application should individually Update() transient instances reachable from the given transient instance if and only if it wants their state also updated. (Except for lifecycle objects, discussed later.)

Hibernate users have requested a general purpose method that either saves a transient instance by generating a new identifier or update the persistent state associated with its current identifier. The SaveOrUpdate() method now implements this functionality.

NHibernate distinguishes "new" (unsaved) instances from "existing" (saved or loaded in a previous session) instances by the value of their identifier (or version, or timestamp) property. The unsaved-value attribute of the <id> (or <version>, or <timestamp>) mapping specifies which values should be interpreted as representing a "new" instance.

<id name="Id" type="Int64" column="uid" unsaved-value="0">

```
<generator class="hilo"/>
```

</id>

The allowed values of unsaved-value are:

```
   *
     any - always save
   *
     none - always update
   *
     null - save when identifier is null
   *
     valid identifier value - save when identifier is null or the
given value
   *
     undefined - if set for version or timestamp, then identifier
check is used
```

If unsaved-value is not specified for a class, NHibernate will attempt to guess it by creating an instance of the class using the no-argument constructor and reading the property value from the instance.

// in the first session Cat cat = (Cat) firstSession.Load(typeof(Cat), catID);

// in a higher tier of the application Cat mate = new Cat(); cat.Mate = mate;

// later, in a new session secondSession.SaveOrUpdate(cat); // update existing state (cat has a non-null id) secondSession.SaveOrUpdate(mate); // save the new instance (mate has a null id)

The usage and semantics of SaveOrUpdate() seems to be confusing for new users. Firstly, so long as you are not trying to use instances from one session in another new session, you should not need to use Update() or SaveOrUpdate(). Some whole applications will never use either of these methods.

Usually Update() or SaveOrUpdate() are used in the following scenario:

```
   *
     the application loads an object in the first session
   *
     the object is passed up to the UI tier
   *
     some modifications are made to the object
   *
     the object is passed back down to the business logic tier
   *
     the application persists these modifications by calling Update()
in a second session
```

SaveOrUpdate() does the following:

```
   *
     if the object is already persistent in this session, do nothing
   *
     if the object has no identifier property, Save() it
   *
     if the object's identifier matches the criteria specified by
unsaved-value, Save() it
   *
```

```
    if the object is versioned (version or timestamp), then the
version will take precedence to identifier check, unless the versions
unsaved-value="undefined" (default value)
  *
    if another object associated with the session has the same
identifier, throw an exception
```

The last case can be avoided by using SaveOrUpdateCopy(Object o). This method copies the state of the given object onto the persistent object with the same identifier. If there is no persistent instance currently associated with the session, it will be loaded. The method returns the persistent instance. If the given instance is unsaved or does not exist in the database, NHibernate will save it and return it as a newly persistent instance. Otherwise, the given instance does not become associated with the session. In most applications with detached objects, you need both methods, SaveOrUpdate() and SaveOrUpdateCopy().

Reattaching detached objects

The Lock() method allows the application to reassociate an unmodified object with a new session.

//just reassociate: sess.Lock(fritz, LockMode.None); //do a version check, then reassociate: sess.Lock(izi, LockMode.Read); //do a version check, using SELECT ... FOR UPDATE, then reassociate: sess.Lock(pk, LockMode.Upgrade);

Deleting persistent objects

ISession.Delete() will remove an object's state from the database. Of course, your application might still hold a reference to it. So it's best to think of Delete() as making a persistent instance transient.

sess.Delete(cat);

You may also delete many objects at once by passing a NHibernate query string to Delete().

You may now delete objects in any order you like, without risk of foreign key constraint violations. Of course, it is still possible to violate a NOT NULL constraint on a foreign key column by deleting objects in the wrong order.

Flush

From time to time the ISession will execute the SQL statements needed to synchronize the ADO.NET connection's state with the state of objects held in memory. This process, flush, occurs by default at the following points

```
  *
    from some invocations of Find() or Enumerable()
  *
    from NHibernate.ITransaction.Commit()
  *
    from ISession.Flush()
```

The SQL statements are issued in the following order

1. all entity insertions, in the same order the corresponding objects were saved using ISession.Save()
2. all entity updates
3. all collection deletions
4. all collection element deletions, updates and insertions
5. all collection insertions
6. all entity deletions, in the same order the corresponding objects were deleted using ISession.Delete()

(An exception is that objects using native ID generation are inserted when they are saved.)

Except when you explicity Flush(), there are absolutely no guarantees about when the Session executes the ADO.NET calls, only the order in which they are executed. However, NHibernate does guarantee that the ISession.Find(..) methods will never return stale data; nor will they return the wrong data.

It is possible to change the default behavior so that flush occurs less frequently. The FlushMode class defines three different modes: only flush at commit time (and only when the NHibernate ITransaction API is used), flush automatically using the explained routine, or never flush unless Flush() is called explicitly. The last mode is useful for long running units of work, where an ISession is kept open and disconnected for a long time (see Section 10.4, “Optimistic concurrency control”).

sess = sf.OpenSession(); ITransaction tx = sess.BeginTransaction(); sess.FlushMode = FlushMode.Commit; //allow queries to return stale state Cat izi = (Cat) sess.Load(typeof(Cat), id); izi.Name = "iznizi"; // execute some queries.... sess.Find("from Cat as cat left outer join cat.Kittens kitten"); //change to izi is not flushed! ... tx.Commit(); //flush occurs

Ending a Session

Ending a session involves four distinct phases:

* flush the session
* commit the transaction
* close the session
* handle exceptions

Flushing the Session

If you happen to be using the ITransaction API, you don't need to worry about this step. It will be performed implicitly when the transaction is committed. Otherwise you should call ISession.Flush() to ensure that all changes are synchronized with the database.

Committing the database transaction

If you are using the NHibernate ITransaction API, this looks like:

tx.Commit(); // flush the session and commit the transaction

If you are managing ADO.NET transactions yourself you should manually Commit() the ADO.NET transaction.

sess.Flush(); currentTransaction.Commit();

If you decide not to commit your changes:

tx.Rollback(); // rollback the transaction

or:

currentTransaction.Rollback();

If you rollback the transaction you should immediately close and discard the current session to ensure that NHibernate's internal state is consistent.

Closing the ISession

A call to ISession.Close() marks the end of a session. The main implication of Close() is that the ADO.NET connection will be relinquished by the session.

tx.Commit(); sess.Close();

sess.Flush(); currentTransaction.Commit(); sess.Close();

If you provided your own connection, Close() returns a reference to it, so you can manually close it or return it to the pool. Otherwise Close() returns it to the pool.

Exception handling

NHibernate use might lead to exceptions, usually HibernateException. This exception can have a nested inner exception (the root cause), use the InnerException property to access it.

If the ISession throws an exception you should immediately rollback the transaction, call ISession.Close() and discard the ISession instance. Certain methods of ISession will not leave the session in a consistent state.

For exceptions thrown by the data provider while interacting with the database, NHibernate will wrap the error in an instance of ADOException. The underlying exception is accessible by calling ADOException.InnerException.

The following exception handling idiom shows the typical case in NHibernate applications:

using (ISession sess = factory.OpenSession()) using (ITransaction tx = sess.BeginTransaction()) {

```
    // do some work
    ...
    tx.Commit();
```

}

Or, when manually managing ADO.NET transactions:

ISession sess = factory.openSession(); try {

```
    // do some work
    ...
    sess.Flush();
    currentTransaction.Commit();
```

} catch (Exception e) {

```
    currentTransaction.Rollback();
    throw;
```

} finally {

```
    sess.Close();
```

}

Lifecyles and object graphs

To save or update all objects in a graph of associated objects, you must either

```
    *
      Save(), SaveOrUpdate() or Update() each individual object OR
    *
      map associated objects using cascade="all" or cascade="save-
update".
```

Likewise, to delete all objects in a graph, either

```
    *
      Delete() each individual object OR
    *
      map associated objects using cascade="all", cascade="all-delete-
orphan" or cascade="delete".
```

Recommendation:

```
   *
     If the child object's lifespan is bounded by the lifespan of the
of the parent object make it a lifecycle object by specifying
cascade="all".
   *
     Otherwise, Save() and Delete() it explicitly from application
code. If you really want to save yourself some extra typing, use
cascade="save-update" and explicit Delete().
```

Mapping an association (many-to-one, or collection) with cascade="all" marks the association as a parent/child style relationship where save/update/deletion of the parent results in save/update/deletion of the child(ren). Futhermore, a mere reference to a child from a persistent parent will result in save / update of the child. The metaphor is incomplete, however. A child which becomes unreferenced by its parent is not automatically deleted, except in the case of a <one-to-many> association mapped with cascade="all-delete-orphan". The precise semantics of cascading operations are as follows:

```
   *
     If a parent is saved, all children are passed to SaveOrUpdate()
   *
     If a parent is passed to Update() or SaveOrUpdate(), all children
are passed to SaveOrUpdate()
   *
     If a transient child becomes referenced by a persistent parent,
it is passed to SaveOrUpdate()
   *
     If a parent is deleted, all children are passed to Delete()
   *
     If a transient child is dereferenced by a persistent parent,
nothing special happens (the application should explicitly delete the
child if necessary) unless cascade="all-delete-orphan", in which case
the "orphaned" child is deleted.
```

NHibernate does not fully implement "persistence by reachability", which would imply (inefficient) persistent garbage collection. However, due to popular demand, NHibernate does support the notion of entities becoming persistent when referenced by another persistent object. Associations marked cascade="save-update" behave in this way. If you wish to use this approach throughout your application, it's easier to specify the default-cascade attribute of the <hibernate-mapping> element.

Interceptors

The IInterceptor interface provides callbacks from the session to the application allowing the application to inspect and / or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. One possible use for this is to track auditing information. For example, the following IInterceptor automatically sets the CreateTimestamp when an IAuditable is created and updates the LastUpdateTimestamp property when an IAuditable is updated.

using System; using NHibernate.Type;

namespace NHibernate.Test {

```
[Serializable]
public class AuditInterceptor : IInterceptor
{

    private int updates;
    private int creates;

    public void OnDelete(object entity,
                         object id,
                         object[] state,
                         string[] propertyNames,
                         IType[] types)
    {
        // do nothing
    }

    public boolean OnFlushDirty(object entity,
                                object id,
                                object[] currentState,
                                object[] previousState,
                                string[] propertyNames,
                                IType[] types) {

        if ( entity is IAuditable )
        {
            updates++;
            for ( int i=0; i < propertyNames.Length; i++ )
            {
                if ( "LastUpdateTimestamp" == propertyNames[i] )
                {
                    currentState[i] = DateTime.Now;
                    return true;
                }
            }
        }
        return false;
    }

    public boolean OnLoad(object entity,
                          object id,
                          object[] state,
                          string[] propertyNames,
                          IType[] types)
    {
        return false;
    }

    public boolean OnSave(object entity,
                          object id,
                          object[] state,
                          string[] propertyNames,
                          IType[] types)
    {
        if ( entity is IAuditable )
        {
            creates++;
            for ( int i=0; i<propertyNames.Length; i++ )
            {
                if ( "CreateTimestamp" == propertyNames[i] )
```

```
                    {
                        state[i] = DateTime.Now;
                        return true;
                    }
                }
            }
            return false;
        }

        public void PostFlush(ICollection entities)
        {
            Console.Out.WriteLine("Creations: {0}, Updates: {1}",
creates, updates);
        }

        public void PreFlush(ICollection entities) {
            updates=0;
            creates=0;
        }

        ......
        ......
    }

}
```

The interceptor would be specified when a session is created.

ISession session = sf.OpenSession(new AuditInterceptor());

You may also set an interceptor on a global level, using the Configuration:

new Configuration().SetInterceptor(new AuditInterceptor());

Metadata API

NHibernate requires a very rich meta-level model of all entity and value types. From time to time, this model is very useful to the application itself. For example, the application might use NHibernate's metadata to implement a "smart" deep-copy algorithm that understands which objects should be copied (eg. mutable value types) and which should not (eg. immutable value types and, possibly, associated entities).

NHibernate exposes metadata via the IClassMetadata and ICollectionMetadata interfaces and the IType hierarchy. Instances of the metadata interfaces may be obtained from the ISessionFactory.

```
Cat fritz = ......;
IClassMetadata catMeta = sessionfactory.GetClassMetadata(typeof(Cat));
long id = (long) catMeta.GetIdentifier(fritz);
object[] propertyValues = catMeta.GetPropertyValues(fritz);
string[] propertyNames = catMeta.PropertyNames;
IType[] propertyTypes = catMeta.PropertyTypes;

// get an IDictionary of all properties which are not collections or
associations
```

```
// TODO: what about components?

IDictionary namedValues = new Hashtable();
for ( int i=0; i<propertyNames.Length; i++ )
{
   if ( !propertyTypes[i].IsEntityType
&& !propertyTypes[i].IsCollectionType )

{

        namedValues[ propertyNames[i] ] = propertyValues[i];
   }
}
```

Chapter 10. Transactions And Concurrency

De NHibernate - Tradução da Documentação Oficial

Ir para: navegação, pesquisa

Tabela de conteúdo

[esconder]

Chapter 10. Transactions And Concurrency

NHibernate is not itself a database. It is a lightweight object-relational mapping tool. Transaction management is delegated to the underlying database connection. If the connection is enlisted with a distributed transaction, operations performed by the ISession are atomically part of the wider distributed transaction. NHibernate can be seen as a thin adapter to ADO.NET, adding object-oriented semantics.

Configurations, Sessions and Factories

An ISessionFactory is an expensive-to-create, threadsafe object intended to be shared by all application threads. An ISession is an inexpensive, non-threadsafe object that should be used once, for a single business process, and then discarded. For example, when using NHibernate in an ASP.NET application, pages could obtain an ISessionFactory using:

```
ISessionFactory sf = Global.SessionFactory;
```

Each call to a service method could create a new ISession, Flush() it, Commit() its transaction, Close() it and finally discard it. (The ISessionFactory may also be kept in a static Singleton helper variable.)

We use the NHibernate ITransaction API as discussed previously, a single Commit() of a NHibernate ITransaction flushes the state and commits any underlying database connection (with special handling of distributed transactions).

Ensure you understand the semantics of Flush(). Flushing synchronizes the persistent store with in-memory changes but not vice-versa. Note that for all NHibernate ADO.NET connections/transactions, the transaction isolation level for that connection applies to all operations executed by NHibernate!

The next few sections will discuss alternative approaches that utilize versioning to ensure transaction atomicity. These are considered "advanced" approaches to be used with care.

Threads and connections

You should observe the following practices when creating NHibernate Sessions:

- Never create more than one concurrent ISession or ITransaction instance per database connection.
- Be extremely careful when creating more than one ISession per database per transaction. The ISession itself keeps track of updates made to loaded objects, so a different ISession might see stale data.
- The ISession is not threadsafe! Never access the same ISession in two concurrent threads. An ISession is usually only a single unit-of-work!

Considering object identity

The application may concurrently access the same persistent state in two different units-of-work. However, an instance of a persistent class is never shared between two ISession instances. Hence there are two different notions of identity:

Database Identity

```
foo.Id.Equals( bar.Id )
```

CLR Identity

```
foo == bar
```

Then for objects attached to a particular Session, the two notions are equivalent. However, while the application might concurrently access the "same" (persistent identity) business object in two different sessions, the two instances will actually be "different" (CLR identity).

This approach leaves NHibernate and the database to worry about concurrency. The application never needs to synchronize on any business object, as long as it sticks to a single thread per ISession or object identity (within an ISession the application may safely use == to compare objects). 10.4. Optimistic concurrency control

Many business processes require a whole series of interactions with the user interleaved with database accesses. In web and enterprise applications it is not acceptable for a database transaction to span a user interaction.

Maintaining isolation of business processes becomes the partial responsibility of the application tier, hence we call this process a long running application transaction. A single application transaction usually spans several database transactions. It will be atomic if only one of these database transactions (the last one) stores the updated data, all others simply read data.

The only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. NHibernate provides for three possible approaches to writing application code that uses optimistic concurrency.

Long session with automatic versioning

A single ISession instance and its persistent instances are used for the whole application transaction.

The ISession uses optimistic locking with versioning to ensure that many database transactions appear to the application as a single logical application transaction. The ISession is disconnected from any underlying ADO.NET connection when waiting for user interaction. This approach is the most efficient in terms of database access. The application need not concern itself with version checking or with reattaching detached instances.

// foo is an instance loaded earlier by the Session session.Reconnect(); transaction = session.BeginTransaction(); foo.Property = "bar"; session.Flush(); transaction.Commit(); session.Disconnect();

The foo object still knows which ISession it was loaded it. As soon as the ISession has an ADO.NET connection, we commit the changes to the object.

This pattern is problematic if our ISession is too big to be stored during user think time, e.g. an HttpSession should be kept as small as possible. As the ISession is also the (mandatory) first-level cache and contains all loaded objects, we can propably use this strategy only for a few request/response cycles. This is indeed recommended, as the ISession will soon also have stale data.

Many sessions with automatic versioning

Each interaction with the persistent store occurs in a new ISession. However, the same persistent instances are reused for each interaction with the database. The application manipulates the state of detached instances originally loaded in another ISession and then "reassociates" them using ISession.Update() or ISession.SaveOrUpdate().

// foo is an instance loaded by a previous Session foo.Property = "bar"; session = factory.OpenSession(); transaction = session.BeginTransaction(); session.SaveOrUpdate(foo); session.Flush(); transaction.Commit(); session.Close();

You may also call Lock() instead of Update() and use LockMode.Read (performing a version check, bypassing all caches) if you are sure that the object has not been modified.

Customizing automatic versioning

You may disable NHibernate's automatic version increment for particular properties and collections by setting the optimistic-lock mapping attribute to false. NHibernate will then no longer increment versions if the property is dirty.

Legacy database schemas are often static and can't be modified. Or, other applications might also access the same database and don't know how to handle version numbers or even timestamps. In both cases, versioning can't rely on a particular column in a table. To force a version check without a version or timestamp property mapping, with a comparison of the state of all fields in a row, turn on optimistic-lock="all" in the <class> mapping. Note that this concepetually only works if NHibernate can compare the old and new state, i.e. if you use a single long ISession and not session-per-request-with-detached-objects.

Sometimes concurrent modification can be permitted as long as the changes that have been made don't overlap. If you set optimistic-lock="dirty" when mapping the <class>, NHibernate will only compare dirty fields during flush.

In both cases, with dedicated version/timestamp columns or with full/dirty field comparison, NHibernate uses a single UPDATE statement (with an appropriate WHERE clause) per entity to execute the version check and update the information. If you use transitive persistence to cascade reattachment to associated entities, NHibernate might execute uneccessary updates. This is usually not a problem, but on update triggers in the database might be executed even when no changes have been made to detached instances. You can customize this behavior by setting select-before-update="true" in the <class> mapping, forcing Hibernate to SELECT the instance to ensure that changes did actually occur, before updating the row.

Application version checking

Each interaction with the database occurs in a new ISession that reloads all persistent instances from the database before manipulating them. This approach forces the application to carry out its own version checking to ensure application transaction

isolation. (Of course, NHibernate will still update version numbers for you.) This approach is the least efficient in terms of database access.

// foo is an instance loaded by a previous Session session = factory.OpenSession(); transaction = session.BeginTransaction(); int oldVersion = foo.Version; session.Load(foo, foo.Key); if (oldVersion != foo.Version) throw new StaleObjectStateException(); foo.Property = "bar"; session.Flush(); transaction.Commit(); session.close();

Of course, if you are operating in a low-data-concurrency environment and don't require version checking, you may use this approach and just skip the version check.

Session disconnection

The first approach described above is to maintain a single ISession for a whole business process thats spans user think time. (For example, a servlet might keep an ISession in the user's HttpSession.) For performance reasons you should

1. commit the ITransaction and then
2. disconnect the ISession from the ADO.NET connection

before waiting for user activity. The method ISession.Disconnect() will disconnect the session from the ADO.NET connection and return the connection to the pool (unless you provided the connection).

ISession.Reconnect() obtains a new connection (or you may supply one) and restarts the session. After reconnection, to force a version check on data you aren't updating, you may call ISession.Lock() on any objects that might have been updated by another transaction. You don't need to lock any data that you are updating.

Heres an example:

```
ISessionFactory sessions;
IList fooList;
Bar bar;

ISession s = sessions.OpenSession();
ITransaction tx = null;

try
{
   tx = s.BeginTransaction());

   fooList = s.Find(
        "select foo from Eg.Foo foo where foo.Date = current date"
       // uses db2 date function
   );

   bar = new Bar();
   s.Save(bar);

   tx.Commit();
}
catch (Exception)
{
```

```
   if (tx != null) tx.Rollback();
   s.Close();
   throw;
}
s.Disconnect();
```

Later on:

```
s.Reconnect();

try
{
   tx = s.BeginTransaction();

   bar.FooTable = new HashMap();
   foreach (Foo foo in fooList)
   {
       s.Lock(foo, LockMode.Read);    //check that foo isn't stale
       bar.FooTable.Put( foo.Name, foo );
   }

   tx.Commit();
}
catch (Exception)
{
   if (tx != null) tx.Rollback();
   throw;
}
finally
{
   s.Close();
}
```

You can see from this how the relationship between ITransactions and ISessions is many-to-one, An ISession represents a conversation between the application and the database. The ITransaction breaks that conversation up into atomic units of work at the database level.

Pessimistic Locking

It is not intended that users spend much time worring about locking strategies. It's usually enough to specify an isolation level for the ADO.NET connections and then simply let the database do all the work. However, advanced users may sometimes wish to obtain exclusive pessimistic locks, or re-obtain locks at the start of a new transaction.

NHibernate will always use the locking mechanism of the database, never lock objects in memory!

The LockMode class defines the different lock levels that may be acquired by NHibernate. A lock is obtained by the following mechanisms:

- LockMode.Write is acquired automatically when NHibernate updates or inserts a row.
- LockMode.Upgrade may be acquired upon explicit user request using SELECT ... FOR UPDATE on databases which support that syntax.

- LockMode.UpgradeNoWait may be acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAIT under Oracle.
- LockMode.Read is acquired automatically when NHibernate reads data under Repeatable Read or Serializable isolation level. May be re-acquired by explicit user request.
- LockMode.None represents the absence of a lock. All objects switch to this lock mode at the end of an ITransaction. Objects associated with the session via a call to Update() or SaveOrUpdate() also start out in this lock mode.

The "explicit user request" is expressed in one of the following ways:

- A call to ISession.Load(), specifying a LockMode.
- A call to ISession.Lock().
- A call to IQuery.SetLockMode().

If ISession.Load() is called with Upgrade or UpgradeNoWait, and the requested object was not yet loaded by the session, the object is loaded using SELECT ... FOR UPDATE. If Load() is called for an object that is already loaded with a less restrictive lock than the one requested, NHibernate calls Lock() for that object.

ISession.Lock() performs a version number check if the specified lock mode is Read, Upgrade or UpgradeNoWait. (In the case of Upgrade or UpgradeNoWait, SELECT ... FOR UPDATE is used.)

If the database does not support the requested lock mode, NHibernate will use an appropriate alternate mode (instead of throwing an exception). This ensures that applications will be portable.

Connection Release Modes

The legacy (1.0.x) behavior of NHibernate in regards to ADO.NET connection management was that a ISession would obtain a connection when it was first needed and then hold unto that connection until the session was closed. NHibernate 1.2 introduced the notion of connection release modes to tell a session how to handle its ADO.NET connections. Note that the following discussion is pertinent only to connections provided through a configured IConnectionProvider; user-supplied connections are outside the breadth of this discussion. The different release modes are identified by the enumerated values of NHibernate.ConnectionReleaseMode:

```
   *
     OnClose - is essentially the legacy behavior described above. The
NHibernate session obtains a connection when it first needs to perform
some database access and holds unto that connection until the session
is closed.
   *
     AfterTransaction - says to release connections after a
NHibernate.ITransaction has completed.
```

The configuration parameter hibernate.connection.release_mode is used to specify which release mode to use. The possible values:

```
   *
```

```
     auto (the default) - equivalent to after_transaction in the
current release. It is rarely a good idea to change this default
behavior as failures due to the value of this setting tend to indicate
bugs and/or invalid assumptions in user code.
   *
     on_close - says to use ConnectionReleaseMode.OnClose. This
setting is left for backwards compatibility, but its use is highly
discouraged.
   *
     after_transaction - says to use
ConnectionReleaseMode.AfterTransaction. Note that with
ConnectionReleaseMode.AfterTransaction, if a session is considered to
be in auto-commit mode (i.e. no transaction was started) connections
will be released after every operation.
```

As of NHibernate 1.2.0, if your application manages transactions through .NET APIs such as System.Transactions library, ConnectionReleaseMode.AfterTransaction may cause NHibernate to open and close several connections during one transaction, leading to unnecessary overhead and transaction promotion from local to distributed. Specifying ConnectionReleaseMode.OnClose will revert to the legacy behavior and prevent this problem from occuring.