

NHibernate als ORM oplossing

Weg met de SQL Queries

Wat is ORM? ORM staat in dit geval voor Object Relational Mapping, niet te verwarren met Object Role Modeling. ORM vertaalt een objectmodel naar een databasemodel en verzorgt de koppeling tussen deze twee modellen.

NHibernate is een afgeleide van Hibernate de bekendste OR-Mapper in de Java-wereld. In dit artikel behandelen we mapping, CRUD-operaties en de Criteria API.

NHibernate als ORM oplossing

Door gebruik te maken van NHibernate hoeven we ons geen zorgen meer te maken over SQL, deze wordt namelijk volledig gegenereerd. De standaard operaties voor het aanmaken, lezen, wijzigen en verwijderen zijn zonder queries uit te voeren. Ingewikkelde queries voor het inlezen van collecties kunnen worden uitgevoerd middels de Criteria API, welke we in dit artikel zullen behandelen.

Naast de Criteria API kan er ook gebruik gemaakt worden van Hibernate Query Language (HQL) en Query by Example voor het uitvoeren van complexe queries. Deze mogelijkheden zullen we verder niet behandelen in dit artikel.

Door gebruik te maken van NHibernate als OR-Mapper kan optimaal gebruikgemaakt worden van functionaliteit binnen SQL Server, aangezien NHibernate 1.2 nu ook Stored Procedures ondersteunt.

Virtual Book Shelf

In dit artikel gebruiken we als voorbeeld een applicatie welke kan dienen als de virtuele boekenplank voor een bedrijf. Met wat uitbreiding van de functionaliteit die we behandelen kunnen medewerkers inzien welke boeken het bedrijf heeft en welke medewerker het specifieke boek in bruikleen heeft.

Het objectmodel bestaat uit de objecten Boek en Medewerker welke in codevoorbeeld 1 en 2 zijn uitgewerkt. De standaard get / set properties zijn voor dit artikel niet van belang en hebben we dus ingekort. De relatie "Boek wordt geleend door Medewerker" is geïmplementeerd door Medewerker als eigenschap van Boek op te nemen. Deze relatie bekeken vanuit het Medewerker object, "Medewerker leent Boeken", hebben we gerealiseerd door een generic List van het object Boek op te nemen als eigenschap van Medewerker.

Deze gekozen implementatie wijkt niet af van de standaard wijze waarop deze relatie in een objectmodel wordt geïmplementeerd. Dit is een compleet andere aanpak dan het gebruik van het Dataset object, typed of untyped. Bij datasets werk je met een objectmodel dat erg lijkt op de benadering van de database. Bij het gebruik van TableAdapters worden nog vaak SQL queries aangepast zodat ze aan de eisen voldoen.

Het gebruik van NHibernate zorgt ervoor dat SQL queries tot het verleden behoren. De standaard CRUD operaties vereisen slechts een correcte mapping. Voor complexere queries kan gebruikgemaakt worden van de Criteria API. De mapping, CRUD operaties en de Criteria API behandelen we in de volgende paragrafen.

1	namespace QNH.VirtualBookShelf
---	--

```

2  {
3      [Class]
4      public class Boek
5      {
6          [Id(Name = "BoekId", Access = "field.camelcase-underscore")]
7          [Generator(1, Class = "native")]
8          private int _boekId;
9          private string _titel;
10         private string _auteur;
11         private string _isbn;
12         private Medewerker _lener;
13
14         [Property]
15         public virtual string Titel
16         {...}
17
18         [Property]
19         public virtual string Auteur
20         {...}
21
22         [Property]
23         public virtual string Isbn
24         {...}
25
26         [ManyToOne(Cascade = CascadeStyle.SaveUpdate, Column =
27         "LenerId")]
28         public virtual Medewerker Lener
29         {...}
30     }
31 }
32 }

```

Codevoorbeeld 1: Definitie van het boek object en de mapping naar de database.

```

1  using System.Collections.Generic;
2  using NHibernate.Mapping.Attributes;
3
4  namespace QNH.VirtualBookShelf
5  {
6      [Class(Lazy=false)]
7      public class Medewerker
8      {
9          [Id(Name = "MedewerkerId", Access = "field.camelcase-
10         underscore")]
11         [Generator(1, Class = "native")]
12         private int _medewerkerId;
13         private string _voornaam;
14         private string _achternaam;
15         private string _email;
16         private IList<Boek> _boeken = new List<Boek>();
17
18         [Property]
19         public virtual string Voornaam
20         {...}
21
22         [Property]
23         public virtual string Achternaam

```

```

24     {...}
25
26     [Property]
27     public virtual string Email
28     {...}
29
30     [Bag(Lazy = false)]
31     [Key(1, Column = "LenerId")]
32     [OneToMany(2, ClassType=typeof(Boek))]
33     public virtual IList<Boek> Boeken
34     {...}
35
36     public void BoekToevoegen(Boek boek)
37     {
38         boek.Lener = this;
39         Boeken.Add(boek);
40     }
41 }
42 }

```

Codevoorbeeld 2: Definitie van het medewerker object en de mapping naar de database.

Mappen naar de database

De relatie tussen het objectmodel en de database leggen we vast middels de mapping. De eerste stap is aangeven dat het hier om een object gaat dat mapt op een database tabel (zie codevoorbeeld 1). Zoals in het codevoorbeeld te zien is geven we niet aan wat de naam van de tabel is. Het opgeven van namen is ook voor kolommen niet altijd noodzakelijk. De intelligentie van NHibernate weet zelf op welke tabel en kolommen er wordt gemapt wanneer de namen overeenkomen.

In de mapping word per property in het objectmodel aangegeven bij welke kolom in de database deze hoort. Daarnaast bestaat er een speciale mapping voor id-velden. Voor id-velden kan aangegeven of het id in de database gegenereerd moet. Aangezien het id-veld in veel gevallen geen echt onderdeel van het objectmodel is, is het beter om deze te verbergen. Voor dit id-veld wordt geen property aangemaakt, maar wordt rechtstreeks op het veld gemapt.

Naast het mappen van de standaard properties is de echte toegevoegde waarde te vinden in de relaties tussen objecten. De relatie die we in de database wereld een één-op-veel noemen is ook aan te brengen in het objectmodel. Vanuit ons voorbeeld betekent dit dat iedere medewerker meerdere boeken kan lenen. Deze relatie kan mappen door aan te geven dat het om een lijst gaat, waarbij aangegeven is welke kolom als sleutel geldt. De laatste stap voor het mappen van deze relatie bestaat uit het aangeven van welk type object het een lijst is.

Bij een bi-directionele relatie kan de relatie van twee kanten benaderd worden. Om deze relatie ook in het objectmodel te laten werken dient bij het toevoegen van een boek aan een medewerker, ook de lener van het boek gezet te worden.

In het voorbeeld maken we gebruik van attributen voor het vastleggen van de mapping. Het is echter ook mogelijk gebruik te maken van XML-Files voor het vastleggen van de mapping deze mogelijkheid behandelen we niet in dit artikel.

Uitvoeren van CRUD operaties

De communicatie tussen een applicatie en een onderliggende database bestaat uit de zogenaamde CRUD operaties (Create, Read, Update en Delete). In SQL betekent dit dat er

insert, select, update en delete queries gemaakt moeten worden. Als eenmaal de mapping tussen de database en het objectmodel is ingesteld kunnen de CRUD-operaties aangeroepen worden.

Het aanmaken van een nieuwe medewerker bestaat uit de aanroep van het SaveOrUpdate commando met als argument de nieuwe medewerker. Dit commando bevat logica om te controleren of het om een nieuw of bestaand object gaat en zal op basis daarvan een insert of update query uitvoeren.

Voor het uitlezen van een object is de sleutel noodzakelijk. Meestal zal de sleutel in een id-veld opgeslagen worden. Als de sleutel bekend is kan het Load commando gebruikt worden. Echter wanneer de sleutel niet bekend is, kan er altijd gebruik gemaakt worden van de Criteria API voor het opzoeken van objecten.

Zoals al aangegeven is het commando voor het aanmaken gelijk aan het bijwerken. Echter wanneer het object is gebruikt buiten een NHibernate sessie om, weet NHibernate niets meer van het object. Voordat het SaveOrUpdate commando aangeroepen kan worden moet dan ook de status van de instantie worden verversd middels het Refresh commando.

Ook bij het verwijderen van een instantie moet eerst de status worden bijgewerkt, alvorens het Delete commando aan te roepen.

Create	<pre>using (ISession session = _sessionFactory.OpenSession()) { Medewerker medewerker = new Medewerker(); medewerker.Voornaam = "John"; medewerker.Achternaam = "Doe"; medewerker.Email = "john.doe@microsoft.com"; session.SaveOrUpdate(medewerker); }</pre>
Read	<pre>using (ISession session = _sessionFactory.OpenSession()) { Medewerker medewerker; medewerker = session.Load<Medewerker>(medewerkerId); session.Delete(medewerker); }</pre>
Update	<pre>using (ISession session = _sessionFactory.OpenSession()) { session.Refresh(medewerker); session.SaveOrUpdate(medewerker); }</pre>
Delete	<pre>using (ISession session = _sessionFactory.OpenSession()) { session.Refresh(medewerker); session.Delete(medewerker); }</pre>

Tabel 1: De create, read, update en delete operaties uitgewerkt voor het Medewerker object.

Eén van de voordelen van het gebruik van NHibernate is lazy loading. Wanneer we bijvoorbeeld alle medewerkers inladen en we geen gebruikmaken van de boeken die een medewerker in bruikleen heeft, willen we deze boeken niet onnodig inlezen. NHibernate heeft hier een oplossing voor, genaamd lazy loading. Standaard heeft NHibernate op iedere collectie lazy loading geïmplementeerd, in sommige gevallen is dit niet wenselijk en kan het handmatig worden uitgezet.

Queries met de Criteria API

De communicatie tussen een applicatie en de database bestaat grotendeels uit CRUD operaties. Echter het resterende gedeelte van de communicatie met de database bestaat veelal uit complexe joins met meerdere condities. NHibernate ondersteunt ook complexe joins en condities.

Echter zoals we tot nu toe gewend zijn, gebruiken we tijdens de communicatie met NHibernate geen SQL. NHibernate ondersteunt deze queries via twee interfaces. Er kan gebruikgemaakt worden van HQL een objectgeoriënteerde query taal die erg lijkt op SQL. Maar het opbouwen van een query met de Criteria API zorgt voor een echte objectgeoriënteerde query. De query in codevoorbeeld 3 zoekt alle medewerkers waarvan het e-mailadres eindigt op "@qnh.nl". NHibernate zet de query zelf om in een echte SQL query, zodat de query ook daadwerkelijk op de database wordt uitgevoerd, zodat de kracht van de database gebruikt wordt.

```
1 IList<Medewerker> medewerkersQNH;  
2 using (ISession session = _sessionFactory.OpenSession())  
3 {  
4     ICriteria criteria = session.CreateCriteria(typeof (Medewerker));  
5     criteria = criteria.Add(Expression.Like("Email", "%@qnh.nl"));  
6     medewerkersQNH = criteria.List<Medewerker>();  
7 }
```

Codevoorbeeld 3: Zoek medewerkers met een e-mail adres dat eindigt op @qnh.nl.

Conclusie

Met dit artikel hebben we NHibernate geïntroduceerd. Naast de mapping heeft dit artikel de CRUD-operaties en de Criteria API behandeld als alternatief voor sql-queries. NHibernate heeft nog veel andere features die we niet behandeld hebben, zoals transacties en inheritance. Voor meer informatie kan gekeken worden op www.nhibernate.org.

Referenties: www.nhibernate.org

Mark Monster is Software Engineer en Christian Siegers is Architect/Software Engineer bij QNH Solutions.