Maarten Balliauw is .NET Software Engineer bij Dolmen

Aan de slag met NHibernate

WERK MET OBJECTEN, NIET MET RECORDS!

Hoe groot of klein het project ook is, je hebt altijd met databases te maken. Iedereen weet dat een solide data-access-laag van groot belang is en we spenderen er daarom veel tijd aan. Hulp wordt aangeboden door NHibernate een open-source object-relational mapper. In dit artikel kijken we naar de mogelijkheid en gebruik van NHibernate.

Als .NET-ontwikkelaar werk je normaal gezien met ADO.NET om acties op je database uit te voeren. Data ophalen doe je vaak via een DataSet, dat niet meer is dan een representatie van de database zelf: ze bevat tabellen en relaties, in dezelfde structuur als jouw echte database. Nu objectgeoriënteerd programmeren meer en meer voorkomt, werk je vaak met objecten. Deze hebben niet dezelfde structuur als jouw RDBMS: een RDBMS werkt met relaties, een OO-taal werkt met referenties. Wil je gekoppelde objecten ophalen uit jouw database? Geen probleem, je moet wel de juiste foreign keys aanspreken. Verandert jouw databasestructuur? Dan is de kans groot dat je heel wat code mag gaan controleren en/of herschrijven. Een tweede verschil tussen relationele data en objecten is dat je aan objecten verschillende methodes kunt hangen. Elk object van een bepaalde klasse is verantwoordelijk voor zijn eigen acties en kan de waarde van zijn eigen attributen veranderen. Een record blijft een record en wil je er een bewerking op uitvoeren, dan moet dat steeds door een aparte functie of query gebeuren.

Object-relational mapping

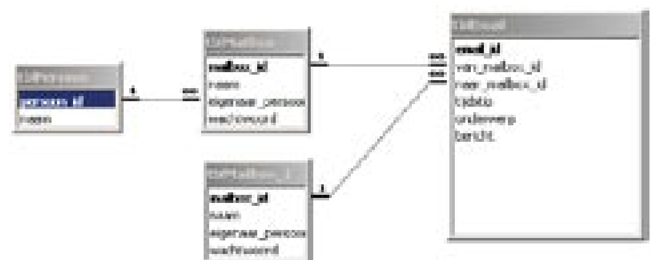
De meeste ontwikkelaars of projectleiders zijn het merendeel van hun tijd kwijt aan het opzetten van een stevige data-acces-laag. Hierin worden dikwijls handmatig records naar objecten omgezet en vice versa, het zogenaamde 'object-relational mapping'. Ook moet je rekening houden met de juiste datatypes, relaties en eventuele verfijningen. Zou het niet handig zijn als dat allemaal voor jou gedaan zou worden? NHibernate is de .NET-versie van een open-source Java-project: Hibernate. NHibernate verzorgt voor de ontwikkelaar de koppeling tussen object en relationele database en doet dit aan de hand van twee principes: reflectie en mapping-files. Reflectie wordt gebruikt om de objecten in jouw applicatie te kunnen aanmaken, bepaalde eigenschappen instellen, enzovoort. De mapping-files beschrijven welke velden van een bepaalde tabel gekoppeld zijn aan de eigenschappen van een bepaald object. In dit artikel bekijken hoe NHibernate precies werkt en dat doen we aan de hand van een eenvoudig voorbeeld: een programma voor het verzenden van interne berichten, waarvan je het begin in afbeelding 1 ziet. De bedoeling is dat je kunt inloggen op een mailbox met naam en wachtwoord en dat je dan de binnengekomen berichten kunt lezen.

Afbeelding 1. Inloggen op een mailbox

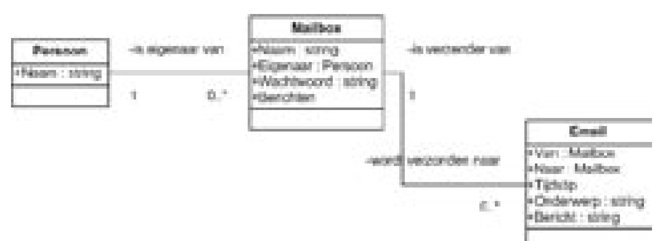
Voor wie de voorbeeldcode wil proberen, heb ik drie mailboxen aangemaakt. Na het starten kun je hierop aanmelden met de volgende combinaties van naam en wachtwoord: maarten/maarten, jan/jan, of peter/peter. De voorbeeldapplicatie wordt gemaakt in C# 2.0, met Visual Studio 2005. Van NHibernate gebruiken we versie 1.0.2.0, gecombineerd met de NHibernate JetDriver 1.0.2.0 voor het gebruik van Microsoft Access als onderliggende database. Er zijn drie classes, namelijk een Mailbox, met een Persoon als eigenaar, en een Mail, die wordt verstuurd van een Mailbox naar een andere Mailbox. Het datamodel vind je terug in afbeelding 2, het klassemodel in afbeelding 3.

Voor het bouwen van een applicatie met NHibernate, maken we allereerst onze projectstructuur, database en classes. Een Access-database is bijgevoegd. Ons project bestaat uit vier aparte lagen: de Windows Forms GUI (WinGUI), de business-laag (BL) en de data-abstractielaag (DAL). Ik voeg hier ook nog een domeinlaag (Domain) aan toe, deze zal onze klassen Persoon, mailbox en e-mail bevatten, en kan binnen de drie andere lagen worden gebruikt. Veel ontwikkelaars zijn gewend om op deze manier te werken. De projectstructuur vind je schematisch in afbeelding 4.

De klasse Mailbox maken we aan in de domeinlaag. Belangrijk is dat er een lege constructor wordt opgenomen, NHibernate



Afbeelding 2. Datamodel



Afbeelding 3. Klassemodel

```
/// <summary>
/// Empty class constructor
/// </summary>
public Mailbox() {}
Codevoorbeeld 1.

string m_Naam = "";
Persoon m_Eigenaar = null;
string m_Wachtwoord = "";
IList m_Berichten = null;

public string Naam {
   get { return this.m_Naam; }
   set { this.m_Naam = value; }
}
// ...
```

Codevoorbeeld 2.

```
1<?xml version="1.0" encoding="utf-8" ?>
2 <hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
3  <class name="Net_Mag.Domain.Mailbox, Net_Mag.Domain" table="tblMailbox">
4    <id name="Id" column="mailbox_id" type="int" unsaved-value="0">
5      <generator class="assigned" />
6    </id>
7    <property name="Naam" column="naam" type="string"/>
8    <many-to-one name="Eigenaar" class="eigenaar_persoon_id"
9    type="Net_Mag.Domain.Persoon, Net_Mag.Domain"/>
10   <property name="Wachtwoord" column="wachtwoord" type="string"/>
11   <bag name="Berichten">
12     <key column="naar_mailbox_id" />
13     <one-to-many class="Net_Mag.Domain.Email, Net_Mag.Domain" />
14   </bag>
15 </class>
16</hibernate-mapping>
```

Codevoorbeeld 3.

gebruikt deze immers om de klasse te instantiëren bij het ophalen van de gegevens uit de database (zie codevoorbeeld 1). In de klasse Mailbox hebben we een aantal private members, die ook als publieke eigenschap toegankelijk zijn. Zoals je ziet in codevoorbeeld 2, werk ik voor m_Berichten niet met Generics. Versie 1.0.2.0 van NHibernate laat het gebruik van Generics (helaas) niet toe zonder enige kunstgrepen toe te passen. Bij de links aan het einde van dit artikel vind je een website waar wordt uitgelegd hoe het wel kan. De klasse laat ik overerven van NHibernateBase, een zelfgeschreven klasse die de gemeenschappelijke members en eigenschappen bevat, waaronder het unieke ID in de database. Vervolgens schrijven we een Nhibernate-mapping. Dit is een XML-bestand, waarin we definiëren welke databasevelden overeenstemmen met welke objecteigenschappen. Ook primaire sleutels en relaties worden hierin aangegeven. Voor elk object dat we uit de database willen ophalen, is een mapping nodig. De mapping voor de klasse mailbox zie je in codevoorbeeld 3.

Hieruit wil ik enkele belangrijke lijnen benadrukken. Op lijn 3 wordt beschreven dat we de klasse mailbox, in de namespace Net_Mag.Domain, mappen op de databasetabel tblMailbox. Het is belangrijk om steeds de volledige klassenaam, inclusief namespace en assembly op te geven. Dit om problemen te vermijden. Op lijn 4 wordt de id van de tabel gemapt op de eigenschap Id, die vanuit NHibernateBase wordt overgeërfd. De generator is hier de AutoNummering uit de database. NHibernate heeft echter ook mechanismen om andere keygenerators aan te spreken, waaronder een eigen keygenerator die steeds de maximumwaarde van de ID met één verhoogt. Vervolgens wordt iedere eigenschap gekoppeld aan een kolom in de database, met vermelding van het type dat in de .NET-applicatie wordt gebruikt, zoals op lijn 7. Als een bepaalde eigenschap echter een relatie is met een andere klasse (binnen het eigen Domein), dan kun je deze onder andere als many-to-one beschrijven, zoals op lijn 8. Op lijn 11, zie je een zogenaamde bag. Een bag is het type dat NHibernate gebruikt om een collectie van andere objecten op te vullen. In jouw .NET-applicatie wordt hiervoor een IList gebruikt. De bag bestaat in dit geval uit e-mail-objecten die naar een bepaalde mailbox zijn gestuurd. Deze relatie wordt aangegeven als one-to-many. NHibernate zal intern zelf de juiste query's opstellen om gerelateerde objecten aan te maken.

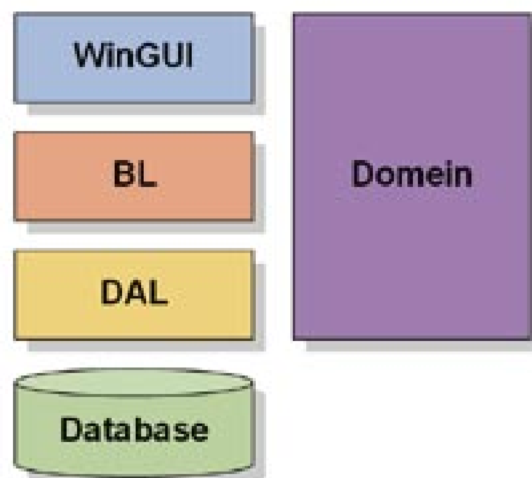
Een keukenhulpje

Het genereren van klassen en mappings kan ook gedaan worden met MyGeneration, een tool die jouw database gaat verkennen, en aan de hand van tabellen en relaties de juiste mappings aanmaakt. In dit artikel kies ik er voor om alles handmatig te doen. Een link naar de website van MyGeneration vind je terug aan het einde van dit artikel.

Tussenlagen

Nu we onze klassen en mappings hebben, kan worden begonnen met het inbouwen ervan in een applicatie. Ik stel voor om eerst de DAL en BL te maken. Iedere DAL-klasse erft van DALBase, een klasse waarin we NHibernate initialiseren en in een statische variabele opslaan. De code van DALBase vind je terug in codevoorbeeld 4.

Een afgeleide DAL-klasse zal dan methoden bevatten om via NHibernate bepaalde Domein-objecten uit de database op te halen. Een voorbeeld van een methode die een bepaalde mail ophaalt aan de hand van zijn unieke ID zie je in codevoorbeeld 5. Hierin zie je ook hoe een bepaalde klasse kan worden opgeslagen in de database. Binnen een DAL-klasse wordt steeds een sessie aangesproken, die we in de factory van DALBase statisch aanmaken. Deze sessie is een Nhibernate-object dat de verdere afhandeling van datamapping, opslag, lezen gaat afhandelen. In verband met 'lazy loading' (zie verderop in het artikel) zorgen we ervoor dat de sessie steeds is geopend. GetOne() haalt in dit geval een record op aan de hand van de ID-kolom die we in de NHibernate mapping-file hebben aangegeven. Nergens is een letter SQL-code te bespeuren, NHibernate regelt de data-abstractie intern. De business-laag kan nu ook eenvoudig gebruik maken van de DAL. Hierin kunnen we telkens een nieuw DAL-object instantiëren en er een methode van oproepen, waarvan het resultaat wordt doorgegeven aan de hogere laag (zie codevoorbeeld 6). Eventueel kan er nog extra validatiecode aan toegevoegd worden.



Afbeelding 4. Projectstructuur

```
using System;
using System.Collections.Generic;
using System.Text;
using NHibernate;
using NHibernate.Cfg;
using Net_Mag.Domain;

namespace Net_Mag.DAL
{
  public abstract class DALBase
  {
    #region Private members
    private static Configuration m_Configuration = null;
    private static ISessionFactory m_Factory = null;
    private static ISession m_Session = null;
    #endregion

    #region Constructor
    /// <summary>
    /// Default constructor
    /// </summary>
    public DALBase()
    {
      // Add mapping references (if needed)
      if (DALBase.m_Configuration == null)
      {
        DALBase.m_Configuration = new Configuration();
        DALBase.m_Configuration.AddAssembly("Net_Mag.DAL");

        // Create session factory
        DALBase.m_Factory =
      DALBase.m_Configuration.BuildSessionFactory();
      }
    }
    #endregion

    #region Public properties
    /// <summary>
    /// Property Factory
    /// </summary>
    public static ISessionFactory Factory
    {
      get { return DALBase.m_Factory; }
      set { DALBase.m_Factory = value; }
    }

    /// <summary>
    /// Property Session
    ///
    /// This will always return an open session
    /// </summary>
    public static ISession Session
    {
      get
      {
        // If needed, create a new session
        if (DALBase.m_Session == null)
        {
          DALBase.m_Session = DALBase.Factory.OpenSession();
        }

        // Return
        return DALBase.m_Session;
      }
      set
      {
        // Set local session
        DALBase.m_Session = value;

        // If the session is closed, open a new one
        // (needed for lazy loading support)
        if (!DALBase.m_Session.IsOpen)
        {
          DALBase.m_Session = DALBase.Factory.OpenSession();
        }
      }
    }
    #endregion
  }
}
```

Codevoorbeeld 4.

Aantwaarps? Zeeuws? Dialecten…

Na het bouwen van onze tussenlagen is er nog één ding dat je nodig hebt: enkele regels in jouw App.config-bestand, die aangeven welke vorm van SQL ('dialect'), en welke database moet worden gebruikt. Dit dialect is bijvoorbeeld verschillend voor SQL Server, Oracle, MySQL en PostgreSQL. Een App.config voor Jet SQL (= Microsoft Access) vind je terug in codevoorbeeld 7.
Hier zijn vier instellingen van groot belang: hibernate.connection.provider, hibernate.dialect en hibernate.connection.driver, die aangeven welke interne drivers gebruikt moeten worden door NHibernate, en hibernate.connection.connection_string, dat eigenlijk een connection-string is zoals je hem normaal zou gebruiken in een ADO.NET-applicatie.

Query's met voorwaarden

Wat ik nog niet heb besproken, is het ophalen van objecten uit de database aan de hand van criteria. Hiervoor kun je denken aan een soort WHERE-clausule. Zie ook codevoorbeeld 8, waarin we een uniek object (DISTINCT) ophalen, waarvan de eigenschappen Naam en Wachtwoord aan een bepaald criterium voldoen. De namespace NHibernate.Expression bevat zowat alle mogelijke criteriabouwstenen om een gedetailleerde query uit te voeren. Mocht dat niet voldoende zijn, dan kun je nog steeds HQL (Hibernate Query Language) gebruiken. En als je daarmee nog niet aan de juiste data komt, kun je terugvallen op normale SQL. Dit is echter af te raden, omdat je niet de databaseonafhankelijkheid die NHibernate biedt, wilt verliezen.

Lazy loading

In de voorbeeldcode heb ik niet gebruikgemaakt van 'lazy loading', maar het is een krachtig iets binnen NHibernate. Lazy loading is het principe dat bij het opvragen van een object, de gerelateerde

```
public Email GetOne(int pId) {
  Email returnValue = null;
  try {
    returnValue = (Email)DALBase.Session.Load(typeof(Email), pId);
    DALBase.Session.Flush();
  } catch (Exception ex) {
    DALBase.Session.Flush();
    throw ex;
  }
  return returnValue;
}

public void Save(Email pEmail) {
  try {
    DALBase.Session.SaveOrUpdate(pEmail);
    DALBase.Session.Flush();
  } catch (Exception ex) {
    DALBase.Session.Flush();
    throw ex;
  }
}
```

Codevoorbeeld 5.

```
public Email GetEmail(int pId) {
  DALEmail dal = new DALEmail();

  try {
    return dal.GetOne(pId);
  } catch (Exception ex) {
    throw ex;
  }
}
```

Codevoorbeeld 6.

objecten nog niet worden opgehaald uit de database. Vanaf het ogenblik dat je een van die objecten toch gebruikt binnen jouw applicatie, zal NHibernate op de achtergrond de gerelateerde objecten ophalen, alsof deze al waren ingeladen. Lazy loading is ideaal als je grote bags hebt, stel je in ons voorbeeld een mailbox voor met enkele duizenden berichten. Op de klassieke wijze haal je dan de mailbox op uit de database, tezamen met die duizenden berichten. En dat terwijl je enkel de naam van de mailbox wilde weten. Met lazy loading zal NHibernate deze berichten pas ophalen vanaf het moment dat je deze nodig hebt. Over prestaties gesproken! Als je gebruikmaakt van lazy loading is het wel belangrijk dat je steeds een open sessie hebt binnen Nhibernate. Doe je dit niet, dan krijg je een Exception als je probeert objecten lazy op te halen. De beste manier is om de sessie dan niet meer aan ieder DAL-object te koppelen, maar aan de huidige thread. In het geval van een Windows-applicatie kun je doen zoals ik heb gedaan: DALBase bevat steeds een open statische sessie die door NHibernate gebruikt kan worden. In ons voorbeeld kun je lazy loading zelf implementeren, door in de mapping van mailbox (Mailbox.hbm.xml) de tag <bag name="Berichten"> uit te breiden naar <bag name="Berichten" lazy="true">. Eenvoudig, niet?

Verder op stap

Uiteraard is NHibernate ondertussen enorm uitgebreid en zou het te ver gaan om alle mogelijkheden op te sommen. NHibernate ondersteunt ook transacties, levensloop van objecten (IsDirty, IsDeleted, …), sessiemanagement, cascading (het automatisch opslaan/verwijderen van gerelateerde objecten), enzovoort. Er bestaat ook een goed boek over (N)Hibernate, dat alle aspecten van Hibernate uitlegt. Dit is 'Hibernate in Action', toegespitst op de Java-variant, maar daarom niet minder informatief voor wie met NHibernate aan de slag wil. Ik ga er van uit dat jouw interesse in NHibernate is opgewekt en dat je de voordelen ervan inziet. (N)Hibernate is in ieder geval een van de meest uitgebreide open-source object-relational mappers op de markt, waar veel informatie, troubleshooting-tips en voorbeeldcode voor te vinden is.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="nhibernate"
      type="System.Configuration.NameValueSectionHandler, System,
          Version=1.0.5000.0,Culture=neutral,
          PublicKeyToken=b77a5c561934e089" />
  </configSections>

  <nhibernate>
    <add
      key="hibernate.connection.provider"
      value="NHibernate.Connection.DriverConnectionProvider" />
    <add
      key="hibernate.dialect"
      value="NHibernate.JetDriver.JetDialect, NHibernate.JetDriver" />
    <add
      key="hibernate.connection.driver_class"
      value="NHibernate.JetDriver.JetDriver, NHibernate.JetDriver" />
    <add
      key="hibernate.connection.connection_string"
      value="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=..\..\..\
            Data\nhibernate_net_mag.mdb" />
  </nhibernate>
</configuration>
```

Codevoorbeeld 7.

```
criteria = DALBase.Session.CreateCriteria(typeof(Mailbox));
criteria.Add(new EqExpression("Naam", pNaam));
criteria.Add(new EqExpression("Wachtwoord", pWachtwoord));

criteria.SetMaxResults(1);
returnValue = (Mailbox)criteria.UniqueResult();
```

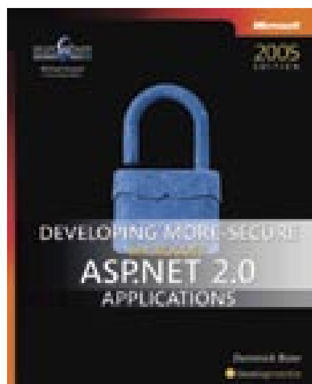
Codevoorbeeld 8.

Maarten Balliauw was twee jaar zelfstandig webontwikkelaar, maar is nu actief als .NET Software Engineer bij Dolmen (www.dolmen.be). Zijn interesses liggen vooral bij webapplicaties, zowel in ASP.NET (C#) als PHP. Je kunt contact met hem opnemen via maarten.balliauw@dolmen.be of via zijn blog op http://www.balliauw.be/maarten.

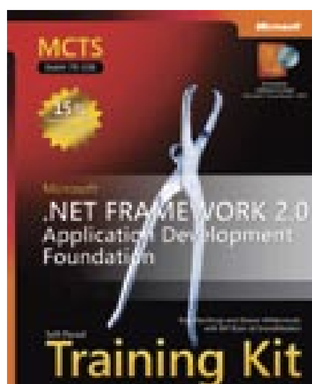
Referenties

Download NHibernate 1.0.2.0:
http://prdownloads.sourceforge.net/NHibernate/
NHibernate-1.0.2.0.zip?download De officiële website: http://www.NHibernate.org
MyGeneration, een generator voor klasses en mappings:
http://www.mygenerationsoftware.com
Generics binnen NHibernate 1.0.2.0:
http://www.ayende.com/projects/NHibernate-query-analyzer/generics.aspx
http://www.hibernate.org/362.html
http://www.hibernate.org/365.html
http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernate
http://www.theserverside.net/tt/articles/showarticle.tss?id=NHibernateP2
http://NHibernate.sourceforge.net/NHibernateEg
http://www.cuyahoga-project.org

(advertentie Microsoft Press)



Developing More-Secure Microsoft ASP.NET 2.0 Applications
ISBN: 9780735623316
Auteur: Dominick Baier
Pagina's: 480



MCTS Self-Paced Training Kit (Exam 70-536): Microsoft .NET Framework 2.0 - Application Development Foundation
Auteur: Tony Northrup and Shawn Wildermuth, with Bill Ryan of GrandMasters
ISBN: 9780735622777
Pagina's: 1088