Entity Framework en Entity SQL

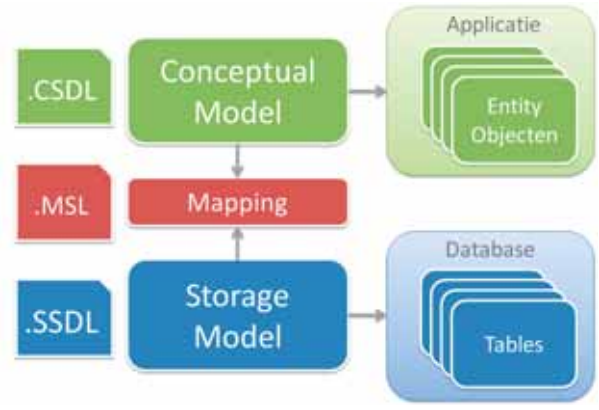
DATABASETOEGANG EN OBJECT RELATIONAL MAPPING MET HET ENTITY-MODEL.

Hoe bouw je een persistence-laag met ADO.NET 3.5? De auteur maakt in zijn voorbeeld gebruik van het Entity Framework. Daarbij ligt de mapping tussen de objectgeoriënteerde classes en het relationele databaseschema vast in het Entity Data Model. Als nieuwe query-taal wordt Entity SQL toegepast.

Doelstelling van de implementatie van het Entity Framework is om voor de ontwikkelaar abstractie te maken van de relationele structuur van de database en die uitsluitend te laten werken met objecten. Voor het bouwen van toepassingen is hierbij geen kennis meer nodig van de onderliggende opbouw van de relaties tussen de tabellen en eventuele stored-procedures. We kunnen werken met classes die aan elkaar gerelateerd zijn via de gekende objectgeoriënteerde technieken als inheritance and associations. Het framework doet at-runtime de vertaalslag naar de queries van hetgeen via deze classes gevraagd is en genereert ook updates, inserts en deletes voor de betreffende tabellen in de database. Om dit te doen moet de mapping tussen de classes en tables beschreven zijn en moet at-runtime dan ook de nodige metadata beschikbaar staan. De metadata is declaratief beschreven in een drietal XML-bestanden. De applicatie zal deze bestanden terugvinden via een in de connectionstring aanwezig path dat naar de eigenlijke database wijst. Zie codevoorbeeld 1.

Het Entity Model

Het belangrijkste concept in het Entity Framework is uiteraard het Entity Model. In een Entity Model wordt alle benodigde informatie voor de mapping tussen objecten en tables vastgelegd. Dit conceptuele model kunnen we vanuit Visual Studio visueel ontwerpen. Dit kan op twee manieren. Ofwel door het importeren van de databasestructuur, waarbij een deel van de mapping reeds aangemaakt zal worden (waarna je het model verder kan uitwerken naar een goed OO-model). Ofwel door eerst de objecten vast te leggen en ze daarna te mappen naar elementen in de database. Deze elementen kunnen bestaan uit tables, views of stored procedures. In Visual Studio krijgt zo'n model de extensie .edmx. In een .edmx file onderscheiden we drie verschillende lagen. Allereerst het store-model, een beschrijving hoe de relationele structuur van de database er uitziet. Hierin komen zaken als veldtypes, primary keys en foreign keys. Daarboven ligt de conceptuele beschrijving van de objecten en hun relaties in objectgeoriënteerde termen. Hier definiëren we zaken als overerving en associaties. Tussen deze twee lagen is de mapping-laag aanwezig. Deze bevat uiteindelijk de informatie die beschrijft hoe de twee werelden naar elkaar toe gemappet worden. Bij het compileren wordt het .edmx-bestand uitgesplitst in .csdl-, .msl-, .ssdl-bestanden, voor respectievelijk de conceptuele, mapping en storage metadata. De applicatie heeft deze drie bestanden nodig om de mapping correct uit te voeren.



AFBEELDING 1. DE ONDERDELEN VAN HET ENTITY MODEL (CONCEPTUAL, MAPPING EN STORAGE)

ObjectContext

De meeste benodigde classes om met het Entity Framework te werken bevinden zich in de System.Data.Objects namespace uit de System.Data.Entity dll. Hierin is ObjectContext meteen de belangrijkste class en zorgt onder andere voor het openen van een connectie naar de eigenlijke database. Bij de instantiatie van deze context kan een connectionstring (zie codevoorbeeld 1) meegegeven worden, waarin ook het path naar de mapping files is vastgelegd. Als deze connectiestring niet is meegegeven zal uiteraard in de .config file gezocht worden. De objectcontext heeft als belangrijkste taak het opvolgen van de wijzigingen in de objecten waarvan de code door het Entity Model is aangemaakt. Deze wijzigingen kunnen toevoegingen zijn, aanpassingen of het verwijderen van objecten. Na het uitvoeren van de SaveChange() method van de ObjectContext worden de nodige SQL statements aangemaakt en naar de database verzonden. De ObjectContext zelf heeft typisch een korte levensduur om zoveel mogelijk concurrency-problemen te vermijden.

```
connectionString="metadata=NorthwindEFModel.
csdl|NorthwindEFModel.ssdl|NorthwindEFModel.
msl;provider=System.Data.SqlClient;provider connection
string=&quot;Data
Source=.\SQLEXPRESS;AttachDbFilename=C:\DB\NorthwindEF.
mdf;Integrated Security=True;User Instance=True;MultipleActiv
eResultSets=True&quot;" providerName="System.Data.EntityClient"
```

CODEVOORBEELD 1. EEN VOOR DE OBJECTCONTEXT BENODIGDE CONNECTIONSTRING

Adv

```
NorthwindEFModel.NorthwindEntities NWCtx;
NWCtx = new NorthwindEFModel.NorthwindEntities();

//Selecting records with a LINQ query
var q = from p in NWCtx.Employees
     select p;

//Create an address
NorthwindEFModel.CommonAddress address;
address = new NorthwindEFModel.CommonAddress();
address.Address = "XXXX";
address.City = "XXX";
address.Country = "BE";
address.PostalCode = "XXXX";
address.Region = "XXXX";

//Create an employee with this address
NorthwindEFModel.Employee employee;
employee = NorthwindEFModel.CurrentEmployee.CreateCurrentEm-
ployee(1001, "Solo", "Han", address);
employee.Title = "Captain of Millennium Falcon";

//Add it to the collections
NWCtx.AddToEmployees(employee);

//Save the changes
NWCtx.SaveChanges();
```

CODEVOORBEELD 2. HET GEBRUIK VAN DE OBJECTCONTEXT, AANMAAK VAN OBJECTEN EN HET BEWAREN VAN DE WIJZIGINGEN

Entity SQL

Het framework gebruikt de taal Entity SQL om op een dynamische manier de opbouw van de queries ten opzichte van het Entity Model toe te laten en de mapping engine aan te sturen om er pure relationele commando's uit te genereren. De taal lijkt zeer sterk op transact SQL maar heeft toch een wat andere syntax. De in deze taal uitgedrukte queries bevinden zich als strings in de code en zijn dus, doordat deze geïnterpreteerd worden, voor de compiler niet zichtbaar. Hierdoor zijn syntaxfouten at runtime slecht te detecteren en dit kan leiden tot exceptions. De meerwaarde van deze taal zit hem in de support voor de objectgeoriënteerde aspecten, vastgelegd in het Entity Model. In Entity SQL kan bijvoorbeeld de conditie van een where clause bestaan uit een navigatie door de associaties van de classes heen. Ook bestaat de mogelijkheid om aan te geven welk type van objecten we als antwoord willen. Bovendien kunnen we voor de beschrijving van de resultaten namen van velden in classes mengen met namen van subclasses. Om zo een antwoord te krijgen dat bestaat uit twee recordsets in een parent-child-relatie, zonder expliciet de relatie ertussen in de query te beschrijven. Het Entity Model kent immers deze relatie. Zie codevoorbeeld 3.

Entity SQL beperkt ook een deel van de 'te vrije' functionaliteit in T-SQL. Zo is het bijvoorbeeld niet toegelaten gebruik te maken van de * wildcard (zoals in SELECT * FROM) om alle velden op te halen zonder deze allemaal expliciet te benoemen. Er moet vastliggen welke velden precies in de recordset als antwoord aanwezig dienen te zijn. Het Entity-model heeft immers alleen kennis van de velden die er in gedefinieerd zijn en kan dus niet overweg met een door de database bepaalde lijst van die velden.

```
SELECT c.CategoryName, c.Products FROM NorthwindEntities.Cate-
gories as c

SELECT p.ProductName FROM OFTYPE(NorthwindEntities.
Products,NorthwindEFModel.DiscontinuedProduct) AS p ORDER BY
p.ProductName

SELECT VALUE o FROM Orders AS o WHERE o.Customer.Address.Coun-
try = 'Mexico'
```

CODEVOORBEELD 3. ENKELE VOORBEELDEN VAN ENTITY SQL STATEMENTS

Query's uitvoeren

Door het Entity Framework zal de ontwikkelaar niet meer de SQL select statements als voorheen opbouwen, maar tegen een tussenlaag gaan spreken. Deze tussenlaag verwacht dus een query die eerder objectgeoriënteerd is dan relationeel. Er zijn drie manieren om gegevens via Entity Framework uit een database te gaan halen.

- *De EntityClient Provider en Entity SQL*

Hierin kent men het programmeermodel zoals in ADO.NET 2.0. Er is een Connection, een Command en een DataReader, meer specifiek een EntityConnection, een EntityCommand en een EntityDataReader. De manier van werken is identiek zoals voorheen. Het EntityCommand heeft een verwijzing nodig naar een EntityConnection en een CommandText die een query-opdracht (in Enity SQL uitgedrukt) bevat. Het Entity command is ook parametriseerbaar en zal een EntityDataReader als antwoord geven op de ExectureReader() method. Hierbij speelt de ObjectContext niet mee, de connectie is geopend via de EntityConnection en er zijn geen wijzigingen uitvoerbaar. De EntityDataReader is zoals zijn voorganger read-only. Zie codevoorbeeld 4 waarin een Enity SQL query wordt uitgevoerd die eigenlijk verscheidene tables aanspreekt en een geneste DataReader teruggeeft. Voor elk record uit een parent table kunnen dus verschillende records uit een child table aangesproken worden. Van belang is te weten dat ADO hierbij verwacht dat de database multiple active resultsets (MARS) ondersteunt. Dit moet je dus ook bij de connectionstring aangeven. Bij deze techniek zijn de resultaten niet op voorhand getypeerd, de velden kunnen via de datareader alleen met hun naam of volgnummer aangesproken worden, zoals ook bij de klassieke DataReader.

- *ObjectServices en Entity SQL*

Met Objectservices werken we ook met Entity SQL die dynamisch als string opgebouwd kan worden. Het uitvoeren ervan resulteert echter in een getypeerde collection van objects met de data. Deze objecten zijn instanties van door het Entity Model aangemaakte classes. Het resultaat is dus sterk getypeerd en men kan de inhoud van de velden direct aanspreken via members van de classes. Ook zal de ObjectContext nu in staat zijn de wijzigingen te detecteren en updates te genereren. Zie

```
string connectionString;
connectionString = @"Name=NorthwindEntities";

EntityConnection connection;
connection = new EntityConnection(connectionString);
connection.Open();

EntityCommand command;
command = new EntityCommand();
command.Connection = connection;

command.CommandText = "SELECT c.CategoryName, c.Products FROM
NorthwindEntities.Categories as c";

EntityDataReader dataReader;
dataReader = command.ExecuteReader(CommandBehavior.Sequenti-
alAccess);

while (dataReader.Read())
{
    //…
    System.Data.Common.DbDataReader nestedReader;
    nestedReader = dataReader.GetDataReader(1);
    while (nestedReader.Read())
    {
        //…
    }
}
```

CODEVOORBEELD 4. EEN ENTITYCOMMAND MET EEN GENESTE DATAREADER ALS RESULTAAT

```
NorthwindEFModel.NorthwindEntities NWCtx;
NWCtx = new NorthwindEFModel.NorthwindEntities();

string entitySQL = "SELECT VALUE o FROM Orders AS o WHERE
o.Customer.Address.Country = 'Mexico';";

ObjectQuery<NorthwindEFModel.Order> q = NWCtx.
CreateQuery<NorthwindEFModel.Order>(entitySQL).
Include("Customer");

foreach (NorthwindEFModel.Order item in q)
{
//…
}
```

CODEVOORBEELD 5. EEN QUERY DOOR MIDDEL VAN OBJECTSERVICES

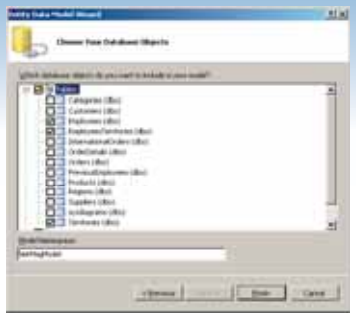
codevoorbeeld 5 waarin we via de context de opdracht geven een Entity SQL command uit te voeren en hierbij aangeven welke classes voor de resultaten gebruikt moeten worden.

- *LINQ to Entities*

De derde mogelijkheid is het gebruik van een LINQ query. Met LINQ legt men een query vast als een commando dat de compiler ziet en dus controle mogelijk is op typeringsfouten. De elementen die in de query als resultaat kunnen voorkomen en de mogelijke velden waarop de filter in de where clause zich kan baseren, zijn reeds aanwezig als classes en members van classes. De ondersteuning door middel van intellisense betekent hier een zeer belangrijke meerwaarde.

O/R mapping.

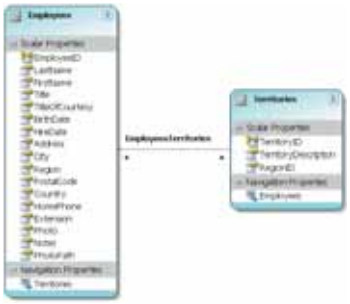
Object/Relational mapping zorgt dus voor de beschrijving door middel van metadata hoe classes naar tables gekoppeld zijn. Belangrijk te weten



AFBEELDING 2. HET IMPORTEREN VAN TABLES IN HET ENTITY MODEL

is dat je ook views en stored procedures in deze mapping kan toepassen. De kracht van O/R mapping wordt snel duidelijk als we in de database gebruikmaken van een meer-op-meer relatie. Een meer-op-meer relatie tussen twee tables bouw je effectief op door een derde table die de primary keys van de twee tables bevat. Deze tussentabel is eigenlijk niet relevant voor onze classes en O/R mapping zorgt hier dus voor de nodige abstractie. In het Entity Model zullen slechts twee entities bestaan met elk een collectie van referenties naar elkaar. Al bij het importeren van het databaseschema in het Entity Model worden de klassieke meer-op-meer relaties via de tussen-table herkend en automatisch omgevormd naar de OO-manier met collecties. Zie hiervoor de screenshots in afbeeldingen 2 en 3. Daarin worden drie tables uit de Northwind database geïmporteerd en komen er uiteindelijk twee entiteiten in het model terecht. Let op de *..*-relatie tussen de twee entiteiten die er automatisch bijkwam.

Maar O/R mapping kan natuurlijk verder gaan dan dit gedrag alleen. Er is een aantal typische mapping patterns tussen classes en tables die het Enity Framework ondersteunen. Deze patronen staan beschreven in het boek 'Patterns of Enterprise Application Architecture' van Martin Fowler. Deze patterns kunnen vanuit de XML-notatie in de .edmx-bestanden (of de .cssd-, .msl- en .ssdl-bestanden) of vanuit de designer opgezet worden. In de bètaversies van het framework heeft de designer nog een te beperkte functionaliteit om dit puur visueel te doen, dus moet er vanuit de XML gewerkt worden.

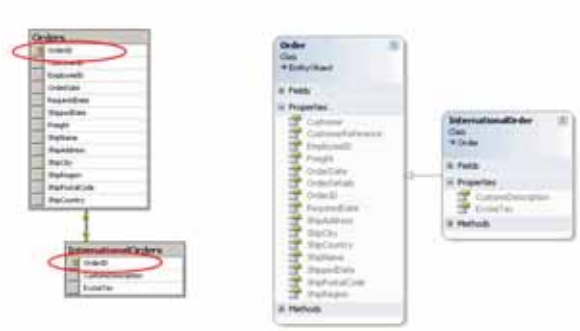


AFBEELDING 3. ENTITEITEN MET EEN M-N RELATIE

AFBEELDING 4. TABLE PER CONCRETE TYPE: EMPLOYEES – PREVIOUSEMPLOYEES

```
<!-- CSDL -->

<EntityType Name="CurrentEmployee" BaseType="NorthwindEFModel.
Employee">
</EntityType>
<EntityType Name="PreviousEmployee" BaseType="NorthwindEFModel.
Employee">
</EntityType>

<!-- MSL -->

<EntitySetMapping Name="Employees" >
 <EntityTypeMapping TypeName ="NorthwindEFModel.CurrentEm-
ployee">
  <MappingFragment StoreEntitySet ="Employees">
   ...
 </EntityTypeMapping>
 <EntityTypeMapping TypeName="NorthwindEFModel.PreviousEm-
ployee">
   <MappingFragment StoreEntitySet ="PreviousEmployees">
   ...
  </MappingFragment>
 </EntityTypeMapping>
<EntitySetMapping>
```

CODEVOORBEELD 6. TWEE ENTITIES ERVEN OVER VAN EEN GEMEENSCHAPPELIJK BASETYPE



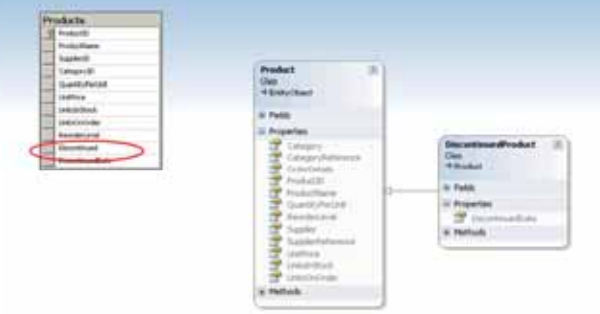
AFBEELDING 5. TABLE PER TYPE: ORDERS – INTERNATIONAL ORDERS

```
<!-- CSDL -->

<EntityType Name="InternationalOrder" BaseType ="NorthwindEF-
Model.Order">
</EntityType>

<!-- MSL -->

<EntitySetMapping Name="Orders">
 <EntityTypeMapping TypeName="IsTypeOf(NorthwindEFModel.Or-
der)">
  <MappingFragment StoreEntitySet="Orders">
   ...
  </MappingFragment>
 </EntityTypeMapping>
 <EntityTypeMapping TypeName ="NorthwindEFModel.Internationa-
lOrder">
  <MappingFragment StoreEntitySet="InternationalOrders">
   ...
  </MappingFragment>
 </EntityTypeMapping>
<EntitySetMapping>
```

CODEVOORBEELD 7. CODEFRAGMENT VOOR TABLE PER TYPE

Concrete Table Inheritance

Bij Concrete Table Inheritance (of Table Per Concrete Type) komt een table overeen met een concreet type. Daarbij worden gelijkaardige tables met gemeenschappelijke velden, dus elk naar een overerving van een abstracte class, met deze gemeenschappelijke velden gemappet. Typisch is hier een over de verschillende tables heen unieke primary key. Als voorbeeld nemen we een table met huidige werknemers en een tweede table met ex-werknemers. De reden om dit vanuit databasestandpunt in twee



AFBEELDING 6. TABLE PER HIERACHY: PRODUCTS – DISCONTINUED PRODUCTS

```
<!-- CSDL -->

<EntityType Name="Product">
</EntityType>

<EntityType Name="DiscontinuedProduct"
BaseType="NorthwindEFModel.Product">
</EntityType>

<!-- MSL -->

<EntitySetMapping Name="Products">
 <EntityTypeMapping TypeName="NorthwindEFModel.Product">
  <MappingFragment StoreEntitySet="Products">
   ...
   <Condition ColumnName="Discontinued" Value="false"/>
  </MappingFragment>
 </EntityTypeMapping>
 <EntityTypeMapping TypeName="NorthwindEFModel.Discontinued-
Product">
  <MappingFragment StoreEntitySet="Products">
   ...
   <Condition ColumnName="Discontinued" Value="true"/>
  </MappingFragment>
 </EntityTypeMapping>
</EntitySetMapping>
```

CODEVOORBEELD 8. TABLE PER HIERACHY, WAARBIJ DE CONDITIE VOOR DE DISCRIMINATOR IS OPGEGEVEN

tabellen uit te splitsen, is performantiewinst. Het gaat nu eenmaal sneller om alleen op de huidige werknemers te selecteren, terwijl je toch ook alle records van ex-werknemers wilt behouden.

Class Table Inheritance

Het pattern Class Table Inheritance (of Table Per Type) is vooral bruikbaar waar velden die logisch gezien bij elkaar horen toch verspreid zijn over meer tables. Misschien zijn deze tables historisch zo gegroeid en werden ze eerder ook door verschillende applicaties gebruikt. Het voorbeeld bestaat uit een table met orders en een tweede table met extra velden voor internationale orders. Een internationale order heeft naast alle velden van de orders table enkele extra velden. In classes wordt dit patroon geïmplementeerd door overerving.

Single Table Inheritance

Bij het pattern Single Table Inheritance (of Table Per Hierarchy) is het mogelijk een table naar diverse classes te mappen omdat in de table eigenlijk twee types van records kunnen voorkomen. Het veld discriminator geeft aan over welk type het eigenlijk gaat. Dit resulteert in classes die van elkaar overerven, waarbij de onderliggende class de extra velden bevat. In het voorbeeld zien we de tabel Products met een boolean-veld (discontinued) om aan te geven dat een product eigenlijk niet meer beschikbaar is. Een exta veld (DiscontinuedDate) is uitsluitend voor dit type producten van belang.
Uiteraard zijn dit slechts kleine fragmenten uit de mapping files ter verduidelijking van de patterns. Er is heel wat meer metadata aanwezig die

```
try
{
    ctx.SaveChanges();
}
catch (OptimisticConcurrencyException ex)
{

    List<object> failed = new List<object>();

    foreach (ObjectStateEntry entry in ex.StateEntries)
    {
        failed.Add(entry.Entity);
    }
    if (AskUserWhatToDo())
    {
        ctx.Refresh(RefreshMode.ClientWins, failed);
        ctx.SaveChanges();
    }
    else
    {
        ctxUser2.Refresh(RefreshMode.StoreWins, failed);
        ctxUser2.SaveChanges();
    }
}
```

CODEVOORBEELD 9. HOE HET CONCURRENCY-PROBLEEM OPLOSSEN

onder andere de velden van en de associaties tussen entities beschrijven.

Hoe concurrency oplossen?

Bij iedere multi-user databaseapplicatie kan de situatie voorkomen dat verscheidene gebruikers met dezelfde data werken en ieder ook wijzigingen aanbrengt. Het sturen van de updates naar de database kan bij de SaveChanges() method leiden tot concurrency-problematiek. Het is aan de ontwikkelaar om dit op te lossen. Entity Framework en de ObjectContext gebruiken optimistic locking. Dit wil zeggen dat men er van uit gaat dat een concurrency-probleem zich meestal niet voordoet en dat er actie wordt ondernomen, mocht dit dan toch gebeuren. Als er zich een concurrency-probleem voordoet, ontstaat een OptimisticConcurrencyException. Na het opvangen van deze exception is van elk van de records die het probleem hebben veroorzaakt de originele waarde en de huidige te zien (codevoorbeeld 9). Op basis van deze waarden kan de applicatie eventueel beslissen hoe verder te gaan. Bedoeling is om de ObjectContext te verversen waarbij opgegeven kan worden wat het gedrag moet zijn bij de volgende SaveChanges(). Ofwel beslist men dat de database ongewijzigd blijft en dus de updates van de tweede user verloren zijn (StoreWins), ofwel kan het overschrijven van de database geforceerd worden. Hierdoor past de tweede gebruiker de door de eerste gebruiker aangepast data opnieuw aan.

Links:

http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx
http://blogs.msdn.com/adonet/
http://blogs.msdn.com/efdesign/
http://www.microsoft.com/belux/msdn/nl/chopsticks/default.aspx?id=513

Kurt Claeys is een .NET Architect voor Ordina België met focus op WCF/WF en O/R Mapping in ADO.NET 3.5, MCT en actief in de Belgische .NET community. Je kunt hem bereiken via kurt.claeys@ordina en via zijn persoonlijke blog: www.devitect.net