Eric van Wijk is Principal Consultant bij Avanade

ADO.NET Entity Framework

DATA ACCESS GEMAKKELIJKER GEMAAKT

Met de volgende versie van Visual Studio, codenaam 'Orcas', vindt een upgrade plaats van het .NET Framework naar versie 3.5. Als onderdeel van deze versie van .NET introduceert Microsoft het ADO.NET Entity Framework, waardoor de ontwikkelaar niet meer een data acces-laag hoeft te programmeren. In dit artikel gaat de auteur nader in het framework en komt hij met enkele voorbeelden om te illustreren hoe je in deze nieuwe versie met data access kunt omgaan.

We staan er tegenwoordig nog nauwelijks bij stil, maar ooit waren relationele databases controversieel. Pas in de jaren negentig werden relationele databases gemeengoed. Vóór het relationele tijdperk hadden databases meestal een hiërarchisch of netwerkmodel. Om de gewenste data op te halen, was flink wat inspanning vereist, zowel in het ontwerp als in de te schrijven code. Toen in de jaren tachtig en negentig Oracle en anderen relationele databases commercialiseerden, was er aanvankelijk flink wat weerstand. Een van de voornaamste kritiekpunten was gericht op performance. Relationele databases introduceerden namelijk abstracties als tabellen en rijen, en een taal (SQL) die queries formuleerde in termen van verzamelingen in plaats van records. Daar moest, meenden velen, een te zware prijs voor betaald worden. De flexibiliteit die relationele databases toevoegden, woog uiteindelijk op tegen deze nadelen van het eerste uur. En die abstracties in tabellen en rijen maakte het ontwikkelen eigenlijk een stuk eenvoudiger.

Huidige ontwikkelingen

Anno 2007 vindt een soortgelijke verandering plaats in de manier waarin we met data in onze code omgaan. Het meeste (enterprise) ontwikkelwerk vindt plaats in objectgeoriënteerde talen als C#, Java en Visual Basic. In deze talen worden objecten gecreëerd en gemanipuleerd in plaats van tabellen en rijen. Zou het niet veel makkelijker zijn als op deze manier ook data gemanipuleerd worden? Voor velen is het antwoord hier op ja. Zeker als gekeken wordt naar de populariteit van een categorie van frameworks die object-relational mappers (ORM) heet. Object-relational mappers schermen de ontwikkelaar af van de taak objecten om te zetten in relationele data in de database, en omgekeerd. Ontwikkelaars hoeven zich niet te bekommeren om de SQL-statements die nodig zijn om wijzigingen in gemanipuleerde objecten op te slaan. Het enige dat zij moeten doen is een mapping definiëren (hetzij rechtstreeks in XML, hetzij via een visuele tool) tussen objecten en referenties enerzijds en tabellen en foreign key-relaties anderzijds. Voor .NET zijn er tientallen object-relational mappers, waarvan NHibernate de bekendste is (zie het artikel van Maarten Balliauw: 'Aan de slag met NHibernate' in .NET magazine #15). Microsoft onderkent nu ook deze behoefte naar object-relational mapping, en voegt het ADO.NET Entity Framework toe aan .NET.

Hiërarchische / Netwerkdatabase	Relationele database	Entity Data Model (EDM)
Fysiek schema	Logisch schema	Conceptueel schema
Records	Tabellen, rijen, kolommen	Entiteiten, objecten, referenties, properties
	Foreign Keys, referentiële integriteit	Relations, associations
	Constraints	Constraints
	Joins	Navigatie
	ANSI SQL	LINQ, Entity SQL

Tabel 1. Data access-abstractielagen

Entity Framework

Het centrale begrip in het Entity Framework is, zoals de naam al zegt, de *entiteit*. Een entiteit is een tastbaar object zoals een persoon, auto of contract. In het Entity Framework worden entiteiten en relaties uit het applicatiedomein gemodelleerd in een conceptueel schema (CSDL). De daadwerkelijke structuur in de database wordt beschreven in het storage-schema (SSDL). Een mapping-specificatie (MSL) legt de verbinding tussen conceptueel en storage-schema. Het geheel wordt het Entity Data Model (EDM) genoemd; zie tabel 1 voor een vergelijking van EDM-begrippen met databasebegrippen.

Van het conceptuele schema (CSDL) worden classes gegenereerd die entiteiten representeren. Het Entity Framework weet (via de bovengenoemde schema's en mapping-specificatie) hoe deze classes in de database liggen opgeslagen. Hierdoor weet het Entity Framework ook welke veranderingen in de database moeten worden uitgevoerd om gecreëerde of bewerkte instanties van deze classes op te slaan, en hoeft er niet naar DataAdapters, DataSets of DataReaders gegrepen te worden. Naast het bewerken van data is er nog een vorm van interactie met de database, en dat is het ophalen van data. Het Entity Framework voorziet hierin met de toevoeging van twee query-talen: Entity SQL en LINQ to Entities. In beide worden queries geformuleerd tegen het conceptuele model (CSDL). Met LINQ to Entities is dit het meest duidelijk, omdat LINQ-expressies samen met de gerefereerde classes (entiteiten) inline in de C#/VB.NET-code staan. Hieronder staan de onderdelen van een Entity Framework-applicatie, tevens is een overzicht van de architectuur te zien in afbeelding 1.

- **CSDL Schema** - Conceptual Schema Definition Language. Dit schema, een XML-file, beschrijft de entiteiten, hun properties en hun onderlinge relaties. Entiteiten kunnen ook een overervingsrelatie definiëren.
- **SSDL Schema** - Storage Schema Definition Language. Dit schema beschrijft het onderliggende databaseschema in XML. Het Entity Framework heeft dit schema op run-time nodig om te kunnen bepalen hoe queries vertaald moeten worden naar SQL.
- **MSL Mapping** - Mapping Specification Language. Dit schema bevat de daadwerkelijke mapping van entiteiten naar tabellen. Eén entiteit Customer kan bijvoorbeeld in twee tabellen opgeslagen worden (Person en Address).

- **Entity SQL** - Entity SQL is een variant van SQL die gebruikt kan worden om op entiteiten te queriën. Entity SQL heeft een iets andere syntax dan ANSI SQL en T-SQL.
- **LINQ to Entities** - LINQ is de query-taal die toegevoegd wordt aan de versies van C# en VB.NET die met Orcas gereleased worden. Met LINQ kunnen queries rechtstreeks in C# op VB.NET geschreven worden. Met LINQ kan in principe over iedere datastore gequeried worden. Zo bestaat er in Orcas LINQ to Datasets, LINQ to Entities, LINQ to SQL en LINQ to XML. Dit is een krachtig en revolutionair concept, maar zal de nodige gewenning vergen om te gebruiken.

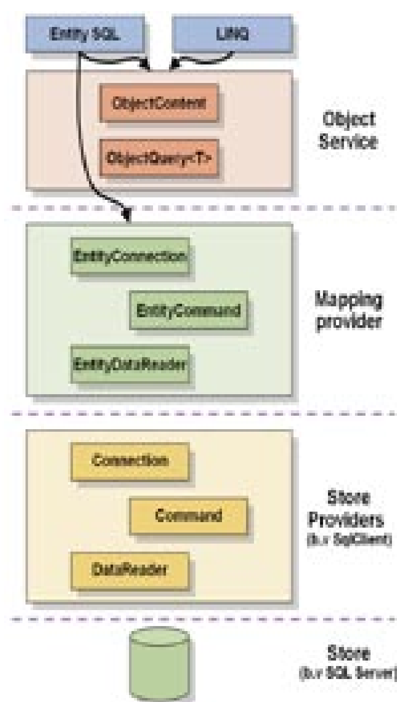
Aan de slag

Voor de voorbeelden die volgen, wordt gebruikgemaakt van de Northwind-database en -schema's die deel uitmaken van de samples die Microsoft beschikbaar heeft gemaakt voor de CTP's van Visual Studio Orcas. In bèta 1 is er nog geen design-time support voor het maken van schema's. Om die reden ontbreken screenshots van die diagrammen. Zie tabel 2 voor een beschrijving van de in het Entity Framework gebruikte schema's.

Entity SQL met overerving

Analoog aan T-SQL is het simpelste wat je kunt doen in Entity SQL 'SELECT VALUE p FROM Products AS p'. Zie codevoorbeeld 1. Deze query wordt uitgevoerd door de *CreateQuery*-method van een *ObjectContext*-instantie. Op basis van de gedefinieerde schema's genereert Visual Studio een class die hier van erft, in dit geval met de naam *EntityNorthwindContext*. De *CreateQuery*-method retourneert een type dat IEnumerable implementeert, dus kan er over geënumereerd worden en is databinding mogelijk. Overigens wordt de query pas echt uitgevoerd in de database op het moment van enumereren in de *foreach*-loop.

Wat er precies gebeurt bij het uitvoeren van de query wordt voor een belangrijk deel bepaald in het model. Zo zijn er in het CSDL-schema subclasses gedefinieerd voor de entiteit *Product*. In de mapping-specificatie (MSL) is bepaald dat er van een *DiscontinuedProduct* sprake is wanneer de kolom *Discontinued* bit-waarde 1 heeft, er is sprake van een *ActiveProduct* wanneer deze kolom waarde 0 heeft. De object-services-laag (zie afbeelding 1) zorgt er vervolgens voor dat instanties van de juiste types worden



Afbeelding 1. ADO.NET Entity Framework Architectuur

```
internal static void Voorbeeld1_Polymorphisme()
{
  using (EntityNorthwindContext db = new EntityNorthwindContext())
  {
    // De Entity SQL query in zijn simpelste vorm
    ObjectQuery<Product> products = db.CreateQuery<Product>(
    "SELECT VALUE p FROM Products AS p");

    foreach (Product p in products)
    {
      // Op basis van de definities van de entiteiten
      //  Product en DiscontinuedProduct in de CSDL
      //  en een overloaded method in partial classes
      //  wordt hier per type class verschillende info geprint
      Console.WriteLine("{0,-21}{1,-35}{2}",
        p.GetType().Name, p.GetStockInfo(), p.ProductName);
    }
  }

  Console.ReadLine();
}
```

Codevoorbeeld 1.

gecreëerd (*materialized* is de term die in het Entity Framework gebruikt wordt). Wanneer de code in dit voorbeeld wordt uitgevoerd, is dit te zien in de eerste kolom van de (console) output.

LINQ met overerving

In dit voorbeeld wordt de query uit codevoorbeeld 1 geformuleerd in termen van LINQ to Entities, met als toevoeging een *where*-clause waarin aangegeven is dat alleen instanties van *ActiveProduct* teruggegeven hoeven te worden; zie codevoorbeeld 2. Merk op dat de *ObjectContext* (lokale variabele *db*) rechtstreeks gerefereerd wordt in de *from*-clause van de LINQ-query. Het uiteindelijke resultaat bij het uitvoeren van dit voorbeeld is dat de output er bijna hetzelfde uit ziet als in het voorbeeld waar Entity SQL gebruikt werd, maar dat alleen instanties van *ActiveProduct* te zien zijn.

LINQ met navigatie

Soms willen we naar andere data navigeren door properties van objecten op te vragen, bijvoorbeeld *category.Products*. Dit is vergelijkbaar met het navigeren over een *DataRelation* in Datasets. In codevoorbeeld 3 proberen we naast het gevraagde *Category*-

```
internal static void Voorbeeld2_Polymorphisme()
{
  using (EntityNorthwindContext db = new EntityNorthwindContext())
  {
    IQueryable<Product> products =
      from p in db.Products
      where p is ActiveProduct
      select p;

    foreach (Product p in products)
    {
      // Op basis van de definities van de entiteiten
      //  Product en DiscontinuedProduct in de CSDL
      //  en een overloaded method in partial classes
      //  wordt hier per type class verschillende info geprint
      Console.WriteLine("{0,-21}{1,-35}{2}",
        p.GetType().Name, p.GetStockInfo(), p.ProductName);
    }
  }

  Console.ReadLine();
}
```

Codevoorbeeld 2.


```
internal static void Voorbeeld3_Navigatie()
{
  using (EntityNorthwindContext db = new EntityNorthwindContext())
  {
    // Retourneer een zgn. object graph
    //  met daarin van de entiteit Product, ook de property Category
    // In bèta 1 is het nog niet mogelijk middels een zgn. span
    //  op te geven dat geneste collecties geretourneerd moeten worden
    IQueryable<Category> categoriesWithProduct = from c in
                    db.Categories
                    where c.CategoryName == "Produce"
                    select c;

    foreach (Category category in categoriesWithProduct)
    {
      Console.WriteLine("{0,-17}", category.CategoryName);
      // Omdat in de bèta 1 geen geneste collecties worden
      //  teruggegeven (anders dan via anonymous types),
      //  expliciet de products collectie vullen
      //  via lazy loading
      // Dit resulteert in een extra query naar de database
      //  _per_ category (N)
      category.Products.Load();
      foreach (Product product in category.Products)
      {
        Console.WriteLine("{0,-17}{1,-21}{2}",
          null,
          product.GetType().Name,
          product.ProductName);
      }
    }
  }

  Console.ReadLine();
}
```

Codevoorbeeld 3.

object ook de bijbehorende *Product*-objecten terug te krijgen. Het is echter nog niet mogelijk (in bèta 1 van Orcas) in LINQ of Entity SQL op te geven dat van een *Category c* ook de *c.Products* in het resultaat moet zitten. Daarom wordt in het voorbeeld gebruikgemaakt van lazy loading. Met lazy loading worden data pas uit de database opgehaald wanneer die gebruikt gaan worden. In het Entity Framework werkt lazy loading expliciet, door middel van de *Load*()-method op de *EntityCollection* class (*c.Products.Load*()). Als alternatief kan er ook met zogenaamde anonymous types gewerkt worden. Hierbij wordt in de query ad hoc een type gedefinieerd en gecreëerd die bestaande types als properties heeft; zie codevoorbeeld 3a. Omdat het tijdelijk gedefinieerde type niet bestaat buiten de definitie van de method waar hij gebruikt wordt, moet ook van het *var*-keyword worden gebruikgemaakt. Hoewel de naam anders doet vermoeden, heeft dit keyword niets te maken met *var* in JavaScript, of *variant* in Visual Basic. De variabele is strongly typed, wat wil zeggen dat het niet mogelijk is het type te veranderen na de eerste assignment. Visual Studio snapt dit ook, zodat je intellisense kunt gebruiken alsof het type expliciet is gedefinieerd.

Updates

Een data access-laag is niet compleet zonder veranderingen te kunnen uitvoeren en persisteren naar de database. In het vierde en laatste codevoorbeeld wordt een bestaande *Category* opgehaald, wordt een nieuwe instantie van *Product* gecreëerd, en wordt deze laatste aan de bestaande *Category* toegevoegd. De LINQ-expressies in dit codevoorbeeld dienen louter om specifieke instanties van *Category* en *Supplier* op te halen. De code die nodig is om toevoegingen en wijzigingen te doen, is gewoon standaard C#: *new* om een nieuwe instantie te creëren, en *Add*()

```
internal static void Voorbeeld3a_Projectie()
{
  using (EntityNorthwindContext db = new EntityNorthwindContext())
  {
    // Retourneer een zgn. object graph
    //  met daarin van de entiteit Product, ook de property Category
    // Het geretourneerde type is een anonymous type,
    //  d.w.z. on-the-fly gedefinieerd en gecreerd
    //  Om deze reden is het ook nodig het var keyword te gebruiken
    //  i.p.v. IQueryable<>
    var productsWithCategory =
      from p in db.Products
      where p.Category.CategoryName == "Produce"
      select new { Product = p, Category = p.Category };

    foreach (var result in productsWithCategory)
    {
      Console.WriteLine("{0,-21}{1,-16}{2}",
        result.Product.GetType().Name,
        result.Category.CategoryName,
        result.Product.ProductName);
    }
  }

  Console.ReadLine();
}
```

Codevoorbeeld 3a.

om deze toe te voegen aan de *Category*. Om de veranderingen te persisteren naar de database, moet als laatste de *SaveChanges*()-method op de *ObjectContext* aangeroepen worden.

Argumenten voor het gebruik van het Entity Framework

Het gebruik van het Entity Framework brengt een aantal voordelen met zich mee ten opzichte van de manier waarin we in ADO.

```
internal static void Voorbeeld4_Update()
{
  using (EntityNorthwindContext db = new EntityNorthwindContext())
  {
    // Vind een category om een nieuw product aan toe te voegen
    // AsEnumerable() is nog nodig in bèta 1
   Category category =
      db.Categories.Where(
      o => o.CategoryName == "Produce"
      ).AsEnumerable().First();
    Console.WriteLine(category.CategoryName);

    Product newProduct = new ActiveProduct();
    //newProduct.Category = category;
    newProduct.ProductName = "New Product";

    // Wijs de eerste de beste supplier toe
    newProduct.Supplier = db.Suppliers.AsEnumerable().First();

    // Voeg het nieuwe product toe aan de context
    db.AddObject(newProduct);

    category.Products.Add(newProduct);

    // Persisteer veranderingen naar de database
    db.SaveChanges();
  }

  Console.ReadLine();
}
```

Codevoorbeeld 4.

NET nu DataAdapters, DataSets en DataReaders gebruiken:

- Het Entity Framework implementeert een volwaardige data access-laag, waardoor ontwikkelaars zich kunnen richten op andere (business) logica.
- LINQ-queries worden compile-time gevalideerd. Hierdoor is zeker dat ze syntactisch juist zijn en dat ze in sync zijn met andere managed code.
- Architecten kunnen van tevoren bepalen welke data op welke wijze ontsloten worden. Een CSDL-model hoeft voor een applicatie immers slechts een bepaald deel van een datamodel te ontsluiten.
- Omdat het databaseschema (SSDL) is losgekoppeld van het conceptuele schema (CSDL), is het mogelijk databaseoptimalisaties zoals (de-)normalisaties uit te voeren zonder dat er managed code gewijzigd hoeft te worden.
- Databinding in WinForms- en WPF-applicaties wordt nog simpeler dan voorheen. Bij 2-weg-databinding hoeft namelijk alleen maar SaveChanges() op een ObjectContext aangeroepen te worden om veranderingen in de database op te slaan, dat is alles.

Er zijn echter ook opmerkingen te plaatsen bij het gebruik van het Entity Framework:

- Standaard wordt runtime dynamisch SQL gegenereerd. In omgevingen waar permissies op stored procedures worden gezet in plaats van op tabellen, is dit een probleem. Er kunnen wel stored procedures gebruikt worden, deze worden echter (nog) niet gegenereerd, waardoor alsnog op twee lagen queries geformuleerd moeten worden.
- In de gegenereerde code erven alle entity-classes van het type Entity, waardoor de businesslaag een afhankelijkheid heeft van de data access-laag (en andersom). Dit is in strijd met de Domain Driven Design/POCO (Plain Old CLR Object) filosofie, waar de businesslaag volledig onafhankelijk is van data access.
- Het lijkt de filosofie om in het Entity Framework met een databaseschema te starten, om vervolgens SSDL, CSDL en MSL te ontwerpen. Daar is de enige design-time tool die nu beschikbaar is, de EDM Wizard, op gericht. Het is nog niet duidelijk in hoeverre ondersteuning komt voor de omgekeerde aanpak, waarbij gestart wordt bij het conceptuele model. Dit laatste zou in overeenstemming zijn met de Domain Driven Design-filosofie, waar begonnen wordt bij het probleemdomein.
- Onduidelijke positionering van LINQ to ten opzichte van LINQ to SQL (voorheen DLINQ). LINQ to SQL is een andere toevoeging in Orcas met vergelijkbare mogelijkheden, alleen zijn die beperkter; zie tabel 2. Hopelijk zal later in het bètatraject van Orcas meer duidelijkheid volgen.
- Onduidelijk wanneer design-time support aan Visual Studio Orcas wordt toegevoegd. De zogenaamde EDM Designer wordt wellicht pas na de release van Orcas als aparte download beschikbaar.

<table>
<tr><th>Eigenschap</th><th>LINQ to SQL</th><th>Entity SQL</th><th>LINQ to Entities</th></tr>
<tr><td>Dynamische queries</td><td></td><td>X</td><td></td></tr>
<tr><td>Compile-time validatie</td><td>X</td><td></td><td>X</td></tr>
<tr><td>Language Extensions Support</td><td>X</td><td></td><td>X</td></tr>
<tr><td>Language Integrated Database Queries</td><td>X</td><td></td><td>X</td></tr>
<tr><td>N-op-M Relaties</td><td></td><td colspan="2">X</td></tr>
<tr><td>Stored Procedures</td><td>X</td><td colspan="2">X</td></tr>
<tr><td>Overerving</td><td></td><td colspan="2">X</td></tr>
<tr><td>Eén entiteit op meerdere tabellen afbeelden</td><td></td><td colspan="2">X</td></tr>
<tr><td>Identity Management</td><td>X</td><td colspan="2">X</td></tr>
<tr><td>Schema</td><td>DBML</td><td colspan="2">SSDL, CSDL, MSL</td></tr>
</table>

Tabel 2. Query-talen

Samenvattend

Hoewel het nog betrekkelijk vroeg is in het bètatraject van Visual Studio Orcas (er staan nog officiële bèta-milestones op stapel), is het duidelijk dat de komst van het ADO.NET Entity Framework verstrekkende gevolgen heeft voor de manier waarop in de toekomst data access-lagen gebouwd worden. Nu Microsoft zelf een OR-mapper opneemt in Visual Studio, valt voor velen de drempel weg om op deze manier met data access om te gaan.

Eric van Wijk is Principal Consultant bij Avanade. Zijn emailadres is ericv@avanade.com.

Referenties

http://blogs.msdn.com/adonet/
http://msdn.microsoft.com/data/default.aspx
http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=533&SiteID=1

(advertentie Microsoft Press)

Microsoft Mobile Development Handbook
Auteurs: Andy Wigley; Daniel Moth; Peter Foot
ISBN: 9780735623583
Pagina's: 688

Managing Projects with Microsoft Visual Studio Team System
Auteurs: Joel Semeniuk; Martin Danner
ISBN: 9780735622166
Pagina's: 272