Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

LINQ for SQL med C#

LINQ for SQL er en af de nye teknologier med .NET 3.5. Den gør det muligt at skrive (typesikre!) database søgninger direkte i programkoden uden at behøve at spekulere (alt for meget) på hvordan de skulle formuleres i SQL.

Skrevet den **06. feb 2009** af **nielle** I kategorien **Programmering / C#** | ★★★★☆

Indledning

Den store introduktion med .NET 3.5 er uden tvivl LINQ. Forkortelsen står for *Language INtegrated Query* og er en teknologi til at skrive søgninger direkte i koden.

LINQ er i virkeligheden en familie af relaterede teknologier: LINQ for Objekter, LINQ for XML og LINQ for SQL (samt LINQ for Entiteter). Denne artikel handler specifikt om LINQ for SQL.

Tidligere gik denne i øvrigt under navnet DLINQ for Database LINQ, og man kan finde masse materiale om emnet under dette navn på Internettet.

LINQ har berøringsflade til de fleste andre større introduktioner i .NET 3.5 - lambda udtryk, extension metoder, partielle metoder og anonyme variable. For en bred gennemgang af nyhederne i .NET 3.5 vil jeg henvise til arne_v's artikel:

http://www.eksperten.dk/artikler/1153

De følgende eksempler er baseret på Visual Studio 2008, .NET 3.5 og C# 3.0. Den anvendte database er en SQL Server 2005, men kunne sagtens have været en anden.

v. 1.0: 05/02/2008 - Første version.

ORM

LINQ for SQL er et eksempel på en ORM: *Object-Relational Mapping*.

Når udviklere arbejder med OOP og relationelle databaser, sidder vi ofte med problemstillingen om hvordan objekter i programmeringssproget skal repræsenteres ved rækker og relationer i databasen. For selv om der er en hel del ligheder, så er der også en del væsentlige forskelle imellem OOP verdenen og den Relationelle verden. En ORM er en teknologi som forsøger at slå bro imellem de to verdener.

LINQ er langt fra det første forsøg på at gøre dette, og inden for .NET verdenen er NHibernate måske nok et af de mest succesfulde:

http://www.hibernate.org/343.html

Tiden må vise om LINQ for SQL kan klare sig i konkurrencen.

Eksempel: en film database

Som demo benyttes denne tabel og disse data:

```
CREATE TABLE [dbo].[Film](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [titel] [nvarchar](50) NULL,
    [premiere] [datetime] NULL,
    [sprog] [nvarchar](50) NULL,
)

1   I Kina spiser de hunde  10-09-1999 00:00:00  DK
2   Villa paranoia          12-03-2004 00:00:00  DK
3   Blinkende lygter        03-11-2000 00:00:00  DK
4   Inkasso                 22-10-2004 00:00:00  DK
5   Forsømte forår, Det     29-01-1993 00:00:00  DK
6   Dansen med Regitze      17-11-1989 00:00:00  DK
7   Midt om natten          09-03-1984 00:00:00  DK
8   Gamle mænd i nye biler  12-07-2002 00:00:00  DK
9   Ambulancen              15-07-2005 00:00:00  DK
10  Casino Royale           24-11-2006 00:00:00  US
11  Adams æbler             05-04-2005 00:00:00  DK
```

Entitets-klasser

LINQ for SQL baserer sig på at man opretter og mapper en klasse for hver tabel man er interesseret i. Klassen dekoreres med en [Table] attribut og de enkelte felter med en [Column].

Det kan f.eks. se sådan her ud:

```
using System;
using System.Linq;

// Der skal desuden oprettes en reference til System.Data.Linq.dll
using System.Data.Linq;
using System.Data.Linq.Mapping;

namespace Film01
{
    [Table]
    class Film
    {
        [Column(IsPrimaryKey = true)]
        public int id;

        [Column]
        public string Titel;

        [Column]
        public DateTime Premiere;

        [Column]
        public string Sprog;
    }
```

```
    ...
}
```

I dette eksempel er der sammenfald mellem hvad tabellen/klassen og felterne i tabellen hhv. klassen hedder. Det eneste andet, som er nødvendigt, er derfor at angive hvilket felt som svare til primær-nøglen:

En sådan klasse kaldes for en *entitets-klasse* (engelsk: *entity class*).

Søgning

Når man har angivet hvordan klassen og tabellen forholder sig til hinanden, kan man bruge klassen til at lave søgninger i tabellen:

```
    class Program
    {
        static void Main(string[] args)
        {
            // Vælg databasen: LinqDemoDatabase.
            string connStr =
                "Data Source=dataexpe-dfocde;Initial
Catalog=LinqDemoDatabase;Integrated Security=SSPI;";

            // Opret en binding til databasen.
            DataContext dataCtx = new DataContext(connStr);

            // Opret en binding mellem filmTbl og Film-tabellen.
            Table<Film> filmTbl = dataCtx.GetTable<Film>();

            // Definér en forespørgsel som:
            // ... henter alle dansksprogede film
            // ... sorteret faldende efter premiere
            // ... og vi vil specifikt have titlen
            IQueryable<string> titelQuery = filmTbl
                .Where(f => f.Sprog == "DK")
                .OrderByDescending(f => f.Premiere)
                .Select(f => f.Titel);

            // Udskriv titlerne.
            foreach (string titel in titelQuery)
            {
                Console.WriteLine(titel);
            }
        }
    }
```

Forklaring af koden:

Først oprettes en forbindelse mellem tabellen og databasen. Dette håndteres via DataContext-klassen, og denne er så at sige hjertet i LINQ for SQL. Det er ultimativt den som oversætter LINQ kommandoerne til

SQL kommandoer.

Man kunne fristes til at læse næste kodelinje som om at indholdet af Film-tabellen bliver hentet ind i listen filmTbl. Dette ville imidlertid være forkert, for der hentes nemlig ikke noget fra databasen før at man rent faktisk begynder at bruge data - i dette tilfælde sker det først et par linjer længere nede i koden, i foreach-løkken.

Det er derfor mere korrekt at sige at filmTbl-listen bindes til Film-tabellen. Grunden til at det fungere på denne måde er af performance hensyn, og det kaldes for *defeered execution* (direkte oversat som: *udskudt udførsel*).

Man kan nu søge i filmTbl. Er man allerede familiær med SQL kan man se at syntaksen har lånt derfra.

Midt inde i koden står der pludselig sådan noget som:

```
f => f.Sprog == "DK"
```

Dette er et *lambda udtryk* og er en af de nye tilføjelser med .NET 3.5. Mere om dem om lidt, men for nu er det nok at sige at det skal læses som noget i stil med: "de film hvor sproget er 'DK'".

Titlerne udskrives til sidst vha. en foreach-løkke, og samlet giver koden i outputtet:

```
Ambulancen
Adams æbler
Inkasso
Villa paranoia
Gamle mænd i nye biler
Blinkende lygter
I Kina spiser de hunde
Forsømte forår, Det
Dansen med Regitze
Midt om natten
```

Lambda- og query syntaks

Den viste syntaks kaldes for *lambda syntaks* og man kalder titelQuery for en *lambda query*.

C# har derudover en alternativ syntaks, som minder endnu mere om SQL; i denne kan queryen fra før skrives som:

```
            IQueryable<string> titelQuery = from f in filmTbl
                                            where f.Sprog == "DK"
                                            orderby f.Premiere descending
                                            select f.Titel;
```

Dette kaldes for *query comprehension syntaks*.

(Man kan sammenligne situationen lidt med at .NET frameworket har typen System.Int32 og at C# har genvejen "int".)

Et lille smut forbi VB.NET

LINQ syntaksen i VB.NET ligner meget. Samme eksempel:

```
Imports System.Data.Linq
Imports System.Data.Linq.Mapping

Module Module1

    <Table()> _
    Class Film
        <Column(IsPrimaryKey:=True)> _
        Public id As Integer

        <Column()> _
        Public Titel As String

        <Column()> _
        Public Premiere As DateTime

        <Column()> _
        Public Sprog As String
    End Class

    Sub Main()
        ' Vælg databasen: LinqDemoDatabase.
        Dim connStr As String = _
        "Data Source=dataexpe-dfocde;Initial
Catalog=LinqDemoDatabase;Integrated Security=SSPI;"

        ' Opret en binding til databasen.
        Dim dataCtx As DataContext = New DataContext(connStr)

        ' Opret en binding mellem filmTbl og Film-tabellen.
        Dim filmTbl As Table(Of Film) = dataCtx.GetTable(Of Film)()

        ' Definér en forespørgsel som:
        ' ... henter alle dansksprogede film
        ' ... sorteret faldende efter premiere
        ' ... og vi vil specifikt have titlen
        Dim titelQuery As IQueryable(Of String) = From f In filmTbl _
        Where f.Sprog = "DK" _
        Order By f.Premiere Descending _
        Select f.Titel

        ' Udskriv titlerne.
        For Each titel As String In titelQuery
            Console.WriteLine(titel)
        Next
    End Sub

End Module
```

Den bagvedliggende SQL

Når koden eksekveres, oversætter DataContekst-objektet queryen til en SQL-kommando. DataContext giver mulighed for at lytte med på dette ved at koble sig på Log propertyen:

```
            // Opret en binding til databasen.
            DataContext dataCtx = new DataContext(connStr);
            dataCtx.Log = Console.Out;
```

I dette tilfælde er resultatet:

```
SELECT [t0].[Titel]
FROM [Film] AS [t0]
WHERE [t0].[Sprog] = @p0
ORDER BY [t0].[Premiere] DESC
-- @p0: Input NVarChar (Size = 2; Prec = 0; Scale = 0) [DK]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.21022.8
```

Der bruges parametre i SQL-kaldet - her i form af @p0. Dette er med til at sikre mod uhensigtsmæssigheder og eventuelle sikkerhedshuller. Efter lidt simplificering ser den sådan her ud:

```
SELECT Titel
FROM Film
WHERE Sprog = 'DK'
ORDER BY Premiere DESC
```

Lambda udtryk

I Where()-funktionen i queryen:

```
IQueryable<string> titelQuery = filmTbl
    .Where(f => f.Sprog == "DK")
    .OrderByDescending(f => f.Premiere)
    .Select(f => f.Titel);
```

indgik dimsen:

```
f => f.Sprog == "DK"
```

Dette kaldes for et *lambda udtryk* (engelsk: *lambda expression*) og er indført med C# 3.0.

Hvordan skal den så læses? Where() forventer at man levere et Film-objekt og at der returneres en boolsk værdi. Man kan udspecificere dette ved at sige at den kræver en delegate med følgende signatur:

```

```

```
delegate bool DanskSprogetFilmDelegate(Film f);
```

Where() udvælger så alle de film fra filmTbl hvor at den boolske værdi er true. I dette tilfælde bruges den til at filtrere på de film der er dansksproget.

Indførelsen af de anonyme metoder i C# 2.0 var egentlig blot et mellemstadie til indførelsen af lambda udtrykkene:

C# 1.0:

En delegate (f.eks. en event-handler) skal i C# 1.0 assignes med en metode man havde oprettet andetsteds i koden:

```
DanskSprogetFilmDelegate c1 =
    DanskSprogetFilm;

static public bool DanskSprogetFilm(Film f)
{
    return f.Sprog == "DK";
}
```

C# 2.0 - Anonyme metoder:

Med indførelsen af anonyme metoder kunne man i stedet definere metoden i samme linje hvor den assignes til delegaten:

```
DanskSprogetFilmDelegate c2 =
    delegate(Film f) { return f.Sprog == "DK"; };
```

C# 3.0 - Lambda udtryk:

Med lambda udtryk er syntaksen blevet gjort endnu mere kompakt:

```
DanskSprogetFilmDelegate c3 =
    (Film f) => f.Sprog == "DK";
```

eller blot:

```
DanskSprogetFilmDelegate c3 =
    f => f.Sprog == "DK";
```

idet C# slutter sig til typen af argumentet, f, via sammenhængen som den bruges i.

CRUD

Når man arbejder med databaser, har man brug for hele CRUD spektrummet: Create, Retreive, Update og Delete - INSERT, SELECT, UPDATE og DELETE.

Fokus i LINQ synes primært at ligge på SELECT - det er her at vi har den pæne nye syntaks. Imidlertid understøttes de andre muligheder selvfølgelig også:

Tilføjelse af nye data - INSERT

Før at vi kan tilføje data, er der brug for et par ændringer til entitets-klassen.

For det første har vi brug for en ordentlig constructor, sådan at vi kan oprette nye Film-objekter på en fornuftig måde. Når vi tilføjer denne render vi imidlertid straks ind i at LINQ for SQL kræver at der også er en constructor helt uden nogen argumenter. Tidligere har vi kunne nøjes med default constructoren, men når vi selv tilføjer en constructor har vi ikke denne mere. Den skal altså også tilføjes.

Det næste der skal til er at id-feltet skal markeres som værende databasens eget ansvar. Hvis man ikke gør det vil LINQ forsøge at skrive id-værdien fra klassen ned i databasen, og det går galt hvis feltet f.eks. er et identity-felt (autonumber i Access, auto_increment i MySQL).

```
[Table]
class Film
{
    [Column(IsPrimaryKey = true, IsDbGenerated = true)]
    public int id;

    [Column]
    public string Titel;

    [Column]
    public DateTime Premiere;

    [Column]
    public string Sprog;

    // Krævet triviel constructor.
    public Film() { }

    // Ikke-triviel constructror.
    public Film(string Titel, DateTime Premiere, string Sprog)
    {
        this.Titel = Titel;
        this.Premiere = Premiere;
        this.Sprog = Sprog;
    }
}
```

Vi er hermed parat til at kunne oprette og tilføje nye film til tabellen:

```
    static void Main(string[] args)
    {
```

```
            string connStr =
                "Data Source=dataexpe-dfocde;Initial
Catalog=LinqDemoDatabase;Integrated Security=SSPI;";

            DataContext dataCtx = new DataContext(connStr);

            Table<Film> filmTbl = dataCtx.GetTable<Film>();

            // Opret et nyt Film-objekt.
            Film film = new Film(
                "Humørkort-stativ-sælgerens søn",
                new DateTime(2002, 6, 28),
                "DK");

            // Tilføj det til tabellen.
            filmTbl.InsertOnSubmit(film);

            // Bed data-conteksten om at submitte ændringerne.
            dataCtx.SubmitChanges();
        }
```

Man kan i øvrigt, efter kaldet til SubmitChanges(), aflæse det id den nye film blev indsat under via film-objektets id property.

Ændring af eksisterende data - UPDATE

Denne kræver ikke yderlige ændringer til entitets-klassen.

```
    class Program
    {
        static void Main(string[] args)
        {
            string connStr =
                   "Data Source=dataexpe-dfocde;Initial
Catalog=LinqDemoDatabase;Integrated Security=SSPI;";

            DataContext dataCtx = new DataContext(connStr);

            Table<Film> filmTbl = dataCtx.GetTable<Film>();

            // Hent filmen. Id=10 er Casino Royale.
            Film film = filmTbl.Single(f => f.id == 10);

            // Ret titlen.
            film.Titel = "Agent 007, " + film.Titel;

            dataCtx.SubmitChanges();
        }
    }
```

Funktionen Single() er en af LINQ funktionerne og den udtrækker et enkelt objekt.

Hvis man ønsker at opdatere mere end én række, laver man en query som udtrækker de ønskede rækker. De opdateres så i en passende foreach-løkke, hvorefter SubmitChanges() kaldes til sidst.

Sletning af eksisterede data - DELETE

```
    class Program
    {
        static void Main(string[] args)
        {
            string connStr =
                "Data Source=dataexpe-dfocde;Initial
Catalog=LinqDemoDatabase;Integrated Security=SSPI;";

            DataContext dataCtx = new DataContext(connStr);

            Table<Film> filmTbl = dataCtx.GetTable<Film>();

            // Hent filmen. Id=12 er ...
            Film film = filmTbl.Single(f => f.id == 12);

            // Marker den for sletning.
            filmTbl.DeleteOnSubmit(film);

            dataCtx.SubmitChanges();
        }
    }
```

Hvis man ønsker at slette flere rækker gøres det som for update: Lav en query som udtrækker de ønskede poster, kald DeleteOnSubmit() på hver af dem i en foreach-løkke, og kald så SubmitChanges() tilsidst.

Lidt mere om entites-klasser

Hidtidig har jeg gået ud fra at entitets-klassen hedder det samme som tabellen. Der kan være forskellige grunde til at man ønsker at kalde den noget andet. Hvis vi f.eks. ønsker at kalde entitets-klassen for "Movie" og have den til at mappe i mod "Film" tabellen:

```
    [Table(Name="Film")]
    class Movie
    {
        ...
    }
```

Det samme kan man gøre for de enkelte felter:

```
        [Column(Name="Sprog")]
```

```
        public string Language;
```

Men her har man også en anden mulighed, nemlig at gemme værdien bag en property. Dette er f.eks. nyttigt hvis man gerne vil kunne lave validering på værdierne:

```
        private string _language;

        [Column(Name = "Sprog", Storage = "_language")]
        public string Language
        {
            get { return _language; }
            set
            {
                if (value != "DK" && value != "UK")
                    throw new ArgumentException("Unknown language");

                _language = value;
            }
        }
```

Eksempel #2: Databasen udvides lidt

Lad os prøve med et mere kompliceret eksempel:

```
CREATE TABLE [dbo].[Film](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [titel] [nvarchar](50) NULL,
    [premiere] [datetime] NULL,
    [sprog] [nvarchar](50) NULL
)

CREATE TABLE [dbo].[Skuespiller](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [navn] [nvarchar](50) NULL
)

CREATE TABLE [dbo].[Rolle](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [filmId] [int] NULL,
    [skuespillerId] [int] NULL,
    [rolle] [nvarchar](50) NULL
)
```

Id-felterne er desuden sat op som primærnøgler, og der er oprettet en fremmednøgle-constraint mellem filmId og skuespillerId i Rolle tabellen til id'erne i hhv. Film- og Skuespiller-tabellerne.

For et passende udvalg af data vil jeg henvise til http://www.imdb.com

LINQ til SQL designeren

Man kunne begynde at skrive entites-klasser for disse tre tabeller. Imidlertid findes der allerede værktøjer som kan gøre det for os. Der er dels indbygget et direkte i Visual Studio 2008 og der findes også kommandolinje værktøjet SqlMetal.

Lad os prøve med LINQ til SQL designeren i VS 2008:

Trin 1)
I Solution Explore:
Højre-klik på ikonet for projektet > Add > Add New Item ...

Trin 2)
Add New Item dialogen bliver nu vist:
Click på LINQ to SQL Classes > Giv den navnet FilmSkuespillerRolle.dbml > Tryk Add.
Der kommer nu et hvid todelt, næsten tomt, skærmbillede op.

Trin 3) I Server Explorer
Højre-klik på Data Connections > Add Connection ...

Trin 4)
Add Connection dialogen bliver nu vist:
Udfyld Server name > vælg den ønskede database under "Select or enter a databasename" > Tryk OK.
Der skulle nu være kommet et punkt svarende til valgene fra ovenfor under Data Connections i Server Explore vinduet.

Trin 5)
Navigér ned til punktet Tables. Her finder man de tre tabeller: Film, Rolle og Skuespiller. Markér alle tre, og træk dem så ind på det hvide skærmbillede fra før. De skal droppes på den venstre halvdel.

De tre tabeller bliver nu vist grafisk. Hvis man har husket at oprette relationer imellem dem, vil disse også blive vist som pile fra hhv. Film- og Skuespiller-tabellen til Rolle-tabellen.

Det man ser, er i virkeligheden en grafisk repræsentation af de tre entitets-klasser som Visual Studio nu har oprettet for os.

Note 1) Hvis man bruger den samme database sammen med forskellige applikationer kan det være en fordel at have alle entitets-klasserne i én DLL som man så referere fra alle de programmer som skal bruge databasen. Dermed samler man alt database adgang på et centralt sted og det bliver meget nemmere at lave strukturelle ændringer i databasen uden at skulle lave en masse følgerettelser i koden

Note 2) Selvom du er typen som ønsker at skrive dine entitets-klasser selv for at have 100 % styr på dem, kan det godt være meget lærerigt at kigge designerens kode igennem for inspiration.

Lad os straks tage dem i brug - samme eksempel som ovenfor:

```
using System;
using System.Linq;
using System.Data.Linq;

namespace Film05
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            // Opret en type-sikker DataContext.
            FilmSkuespillerRolleDataContext db =
                new FilmSkuespillerRolleDataContext();

            // Opret en binding mellem filmTbl og Film-tabellen.
            Table<Film> filmTbl = db.Films;

            // Definér en forespørgsel som:
            // ... henter alle dansksprogede film
            // ... sorteret faldende efter premiere
            // ... og vi vil specifikt have titlen
            var titelQuery = from f in filmTbl
                             where f.sprog == "DK"
                             orderby f.premiere descending
                             select f.titel;

            foreach (string titel in titelQuery)
            {
                Console.WriteLine(titel);
            }
        }
    }
}
```

Et par ting man bør lægge mærke til:

1) Der er ikke behov for at angive en connection string. Dette skyldes at denne er defineret inde sammen med alt den andet VVS kode som VS 2008 har genereret for os - "VVS" fordi det kaldes for *plumbing* på engelsk.

2) Forbindelsen oprettes nu med klassen FilmSkuespillerRolleDataContext, som er en typesikker version af DataContext. Dette betyder bl.a. at man kan få fat på Film-tabellen på den viste måde - altså uden at skulle kalde dataCtx.GetTable<Film>() som før.

3) Læg i øvrigt mærke til det lille 's' i db.Films. Det skyldes simpelthen at værktøjet er Microsoft's og at det forsøger at sætte et amerikansk flertals-s på navnet. Det virker godt nok en smule malplaceret her i denne sammenhæng, men havde sikkert virket ganske naturligt hvis tabellen havde heddet "Movie".

4) Der er ikke nogen vigtig grund til at bruge "var" som typen for titelQuery. Hidtil har jeg skrevet IQueryable<string> og sådan kunne titelQuery også have været stylet ovenfor. Var angiver at der er tale om en *anonym klasse*. Nogen gange - som om lidt - er det praktisk at kunne anvende denne betegnelse. Men i dette tilfælde er det egentlig bare sjusk og dovenskab. Så lad være med at gøre som jeg.

Data på tværs af tabeller

Nu vi har introduceret flere tabeller, og relationer mellem dem, kan man begynde at arbejde med

joins og den slags. Hvis vi f.eks. var interesseret i alle film som Mads Mikkelsen havde været med i:

```
FilmSkuespillerRolleDataContext db =
    new FilmSkuespillerRolleDataContext();
db.Log = Console.Out;

Table<Film> filmTbl = db.Films;
Table<Skuespiller> skuespillerTbl = db.Skuespillers;
Table<Rolle> rolleTbl = db.Rolles;

var query = from f in filmTbl
            join r in rolleTbl on f.id equals r.filmId
            join s in skuespillerTbl on r.skuespillerId equals s.id
            where s.navn == "Mads Mikkelsen"
            orderby f.titel
            select f.titel + ", " + f.premiere.Value.Year;

foreach (string film in query)
{
    Console.WriteLine(film);
}
```

(Suk, igen de der amerikanske flertals s'er.)

Læg specielt mærke til select-leddet; Der returneres en string som består af sammensatte værdier fra de fundne Film-objekter. Hvis man kigger nøje efter på den SQL sætning som der genereredes kan man se at dette faktisk gøres direkte i databasen **før** at resultatet returneres til C# programmet.

Når man er kommet sig efter wow-effekten bør man nok lige vurdere om det rent performancemæssigt er den rette måde at gøre det på...

En anden måde at opnå samme resultat:

```
FilmSkuespillerRolleDataContext db =
    new FilmSkuespillerRolleDataContext();
db.Log = Console.Out;

Table<Film> filmTbl = db.Films;

var query = from f in filmTbl
            from r in f.Rolles
            where r.Skuespiller.navn == "Mads Mikkelsen"
            orderby f.titel
            select new { f.titel, r.rolle1 };

foreach (var result in query)
{
    Console.WriteLine("{0}, {1}",
        result.titel,
        result.rolle1);
```

```
}
```

Kigger man godt efter kan man se at LINQ for SQL designeren har lavet et par ekstra håndtag til os:

Før det første er Film-klassen udvidet med en collection med navnet Rolles. Det er vores fremmednøgle til Rolle tabellen der er blevet inkluderet der - med flertals-s og det hele. Havde man ikke oprettet en fremmednøgle-constraint (en "relation" i Access) i databasen, ville denne property ikke være blevet oprettet af designeren.

Rolles indeholder Rolle-objekter, ét objekt for hver relation i databasen. Hver Rolle-objekt har desuden en Skuespiller- og en Film-property som peger den anden vej i relationen. I koden er det dog kun Skuespiller-propertyen der bliver brugt.

Endelig er der den der sjove new()-notation. I dette tilfælde bruges den til at oprette en ny klasse med to værdier - nemlig titel og rolle. Vi har overhovedet ikke defineret sådan en klasse på noget tidspunkt i vores kode, og den er i øvrigt heller ikke defineret i de klasser som LINQ til SQL designeren har lavet i baggrunden. Så hvor er den egentlig defineret henne? Svaret er: lige der hvor new() kaldes - klassen defineres på dét sted.

Det er et eksempel på en *anonym klasse*, og C# sørger selv for at holde styr på at den består af to string-felter. Det vil man hurtigt opdage hvis man forsøgte at bruge dem som f.eks. Int32. Klassen er altså stadigvæk typestærk.

Da det returnerede altså er en samling af anonyme klasser, bruges der også en var i foreach-løkken.

Til sidst et lille hmmm: Hvorfor står der egentlig "rolle1" et par steder? Det skyldes simpelthen at både tabellen og et af dens felter begge hedder "Rolle" - designeren har derfor sat et '1' efter den ene af dem for at skelne skarpt imellem dem. Rart at vide at den kan finde ud af det, men normalt fraråder jeg nu at man bruger det samme navn mellem tabellen og dens felter. Simpelthen for at undgå misforståelser.

Vær flink mod database serveren

Hvis vi ikke ønsker at udføre de vilde streng manipulationer i select-leddet, kan vi i stedet gøre det i foreach-løkken. Det næste eksempel viser hvordan vi udskriver alle film og deres rolle besætning:

```
FilmSkuespillerRolleDataContext db =
    new FilmSkuespillerRolleDataContext();
db.Log = Console.Out;

Table<Film> filmTbl = db.Films;
var query = from f in filmTbl
            from r in f.Rolles
            orderby f.titel, r.rolle1
            select new { f.titel, r.rolle1, r.Skuespiller.navn };

foreach (var result in query)
{
    Console.WriteLine("{0} - {1} ({2})",
        result.titel, result.rolle1, result.navn);
```

```
}
```

Så hvad er der mere at fortælle?

Temmelig meget faktisk; emnet er nemlig meget stort.

Det vigtigste er nok at LINQ til SQL er *fortolket*; Selv om LINQ syntaksen ligner den fra f.eks. LINQ for Objekter så adskiller den sig ved at DataContext-klassen forsøger at oversætte LINQ koden til en tilsvarende SQL kode. LINQ for Objekter bliver derimod udført 100 % lokalt i applikationen.

Det er imidlertid ikke alt LINQ kode som har en SQL ækvivalent. I de tilfælde vil LINQ motoren enten smide en runtime exception, eller udførelsen vil går fra at blive håndteret 100 % i SQL koden til at noget af den bliver suppleret med lokal kode.

Jeg har helt undladt at nævne noget om hvordan LINQ bliver brugt sammen med kontroller som f.eks. DataGrid. For det vil jeg henvise til videoen nedenfor som giver et super eksempel på hvordan at man f.eks. kan arbejde med et GridView og et DetailView i den sammenhæng.

Noget andet som videoen viser er hvordan at SQL koden faktisk først bliver udført i det øjeblik man skal bruge data. Det var det der kaldes for *defeered execution*.

I det ovenstående har jeg vist nogle udvalgte LINQ-operatore som f.eks. Where(), Select(), OrderByDescending() og Single(). Der er mange flere end disse. Heldigvis er de universelle på tværs af de forskellige LINQ teknologier, og i min næste artikel vil jeg derfor komme meget mere ind på de manglende - den næste artikel bliver om LINQ for Objekter.

Yderlig læsning

Denne artikel er kraftigt inspireret af:

C# 3.0 in a Nutshell
O'Reilly
2007
Joseph Albahari & Ben Albahari
ISBN: 978-0-596-52757-0

Specielt kapitlerne 8, 9 og 10 som handler om LINQ i forskellige afskygninger.

.oOo.

For en generel artikel om O/RM, ORM, eller O/R mapping:

http://en.wikipedia.org/wiki/Object-relational_mapping

.oOo.

Generelt om LINQ teknologierne:

http://en.wikipedia.org/wiki/Language_Integrated_Query

.oOo.

En rigtig god video om emnet (eksemplerne er i VB.Net)

http://channel9.msdn.com/ShowPost.aspx?PostID=337692

.oOo.

Og til sidst en lille samling How-do-I videoer:

http://msdn2.microsoft.com/en-us/vbasic/bb466226.aspx#linq

Kommentar af nico26 d. 31. mar 2008 | 1

Kommentar af truelz d. 17. feb 2008 | 2

Kanon artikel. Jeg havde set Linq begrebet, men var ikke helt på det rene med hvad det var. Nu føler jeg mig klar til selv at skrive kode.

Kommentar af frosty-dk d. 27. aug 2008 | 3

Har arbejdet med linq i noget tid nu, og kan bestemt fået større forståelse for linq efter denne artikel. Rigtig godt forklaret!