Ewart Nijburg en Patrick Vorgers

LINQ-ing webservices

ONTSLUIT WEBSERVICES DOOR MIDDEL VAN EEN EIGEN LINQ-PROVIDER

Visual Studio 2008 biedt de ontwikkelaar vele nieuwe uitbreidingen. Een van deze uitbreidingen is Language Integrated Query of kortweg LINQ. Met LINQ is het mogelijk data op een uniforme manier in code te benaderen. Vanzelfsprekend levert Microsoft standaard met LINQ ook providers voor bijvoorbeeld SQL-databases en XML-files. Aangezien LINQ op een uniforme manier data kan ontsluiten, zou het ook mogelijk moeten zijn andere databronnen via LINQ te benaderen. Op het eerste gezicht lijkt dit een moeilijke klus. Dit artikel laat echter stap voor stap zien hoe LINQ kan worden gebruikt om webservices te ontsluiten.

Met de introductie van .NET zijn de afgelopen jaren vele systemen door middel van webservices ontsloten. Deze webservices bieden de gebruiker veelal de mogelijkheid in de data te zoeken die zij beheren. Een ontsloten CRM-systeem biedt via webservices de gebruiker waarschijnlijk de mogelijkheid naar klanten en hun interacties te zoeken. Naast traditionele systemen zijn er ook steeds meer webapplicaties die door middel van webservices functionaliteit beschikbaar stellen (Virtual Earth, eBay, Amazon). Al deze aanbieders hebben met elkaar gemeen dat er door middel van webservices in de data kan worden gezocht die zij beheren. Deze webservices zijn echter onafhankelijk van elkaar ontwikkeld en zijn niet uniform in gebruik. Zou het niet handig zijn om deze data op dezelfde manier te kunnen benaderen als de data in onze SQL-databases en XML-files?

Even voorstellen: LINQ

LINQ is een naam voor een set van uitbreidingen op het .NET Framework (3.x) die het mogelijk maakt om in bijvoorbeeld C# of Visual Basic query-, set- en transformatie-operaties uit te voeren. Dit komt er in het kort op neer dat het mogelijk is eenvoudig SQL-achtige code te schrijven. Het beste is dit uit te leggen door middel van een voorbeeld.

```
// Selecteer voorwerpen op eBay op basis van een LINQ-query
var query = from item in Ebay.Items
   where
      item.Title.Contains("Car") &&
      !(item.Title.Contains("Jeep") &&
      item.Title.Contains("NSX"))
   select item;

// Voeg de resultaten van de LINQ-query toe aan het resultaat
List<string> results = new List<string>();
foreach (var item in query)
   results.Add(item.Title);
```

Codevoorbeeld 1. Voorbeeldquery

```
where
   item.Title.Contains("Car") &&
   !(item.Title.Contains("Jeep") &&
   item.Title.Contains("NSX"))
select item;
```

Codevoorbeeld 2. Expressieboomgedeelte

Een ieder die SQL kent, zal het stukje code in codevoorbeeld 1 kunnen begrijpen. Het laat zien dat de code erg veel op een standaard SQL-statement lijkt. Met LINQ is het mogelijk verschillende databronnen op een uniforme manier te ontsluiten. Tabel 1 geeft een overzicht van de standaard providers van LINQ.
De standaard LINQ-providers laten zien dat LINQ op een veelzijdig scala van databronnen kan worden gebruikt. Als we nog eens goed kijken naar codevoorbeeld 1, dan zien we dat deze query voorwerpen selecteert op de veilingsite eBay. In plaats van dat deze LINQ-query naar een database wordt gestuurd, wordt hier onder de motorkap via een webservice een verzoek naar eBay gestuurd om voorwerpen te selecteren die voldoen aan de opgegeven condities. Om dit te realiseren, moet een eigen LINQ-provider worden geschreven.

LINQ onder de motorkap

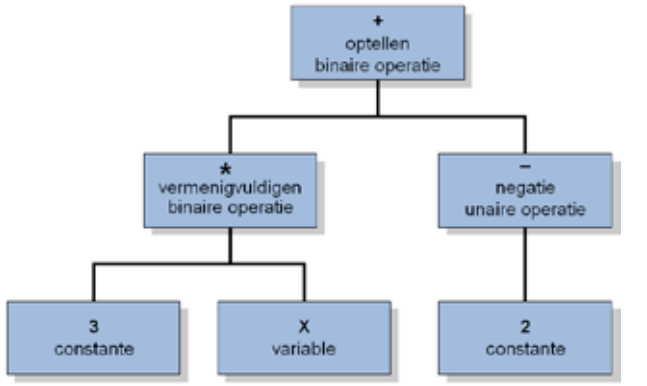
De interne werking van LINQ is gebaseerd op het concept van een expressieboom en de IQueryable-interface. De IQueryable-interface is een speciale interface waar de .NET-compiler specifieke kennis van heeft. Wanneer de compiler in het *from*-gedeelte van een LINQ-query een klasse tegenkomt die de IQueryable-interface implementeert, dan zal de compiler alles wat daarna komt, als onderdeel van de LINQ-query, omzetten naar een expressieboom. Codevoorbeeld 2 geeft weer welke gedeelte van de LINQ-query wordt omgezet in een expressieboom.
Een expressieboom is een datastructuur die tijdens het parsen van source-code veelvuldig wordt gebruikt. Het maakt het voor de compiler/interpreter mogelijk code te genereren door middel van het doorlopen van de expressieboom. In het geval van onze koppeling van LINQ aan webservices wordt de expressieboom gebruikt om de juiste aanroep van de webservice te genereren.

Expressieboom

Een expressieboom is een datastructuur die één of meer lambda-expressies op een eenduidige manier inzichtelijk maakt. Het gaat

Naam	Korte naam	Omschrijving
LINQ to SQL	DLINQ	Query-data uit een SQL-database op basis van een LINQ-expressie
LINQ to XML	XLINQ	Query-data uit een XML-file op basis van een LINQ-expressie
LINQ to Objects		Query-data uit een collectie objecten op basis van een LINQ-expressie

Tabel 1. Standaard LINQ-providers



Afbeelding 1. Expressieboom simpele expressie

bij een expressieboom niet om de code van de lambda-expressies, maar om de metadata om tot die code te komen. Lambda-expressies vertegenwoordigen een beter gestructureerde functionele syntax voor anonymous methods en hebben als voordeel dat ze binnen het .NET-platform kunnen worden omgezet in expressies voor gebruik in expressiebomen. Om een expressieboom goed te begrijpen, bekijken we eerst de simpele expressie: "3 * X + -2". In de vorm van een lambda-expressie zou dit er als volgt uit zien: (int x) => 3 * x + -2
De zojuist beschreven lambda-expressie kan op twee manieren worden gebruikt. Allereerst als Delegate die het resultaat van de berekening retourneert, zie codevoorbeeld 3. De tweede manier ligt wat minder voor de hand. Door de lambda-expressie in een Expression-klasse te wrappen, kunnen we de expressieboom van de lambda-expressie krijgen; zie codevoorbeeld 4.

De variable calcExpression bevat de expressieboom van de lambda-expressie. Door de methode ToString() aan te roepen, kan de oorspronkelijke expressie worden teruggehaald, zij het in een door .NET genormaliseerd formaat. De expressieboom kan vervolgens worden gecompileerd en uitgevoerd. Afbeelding 1 laat de expressieboom zien voor de simpele expressie "3 * X + -2". Hierin zien we dat "3 * X" netjes onder in de boom wordt gezet, omdat deze als eerste moet worden bekeken (vermenigvuldigen gaat voor optellen). Om deze expressie daadwerkelijk te kunnen berekenen, moet de boom in post-order worden doorlopen. Bij het doorlopen van een boom in post-order wordt recursief bij een node eerst het meest linksgelegenelement opgezocht, daarna het rechterelement en als laatste de node zelf.

Voor de voorbeeldexpressie worden de elementen in de boom als volgt benaderd:
1. 3 constante
2. X variabele
3. * vermenigvuldigen binaire operatie
   (Temp1 = 3 * X wordt berekend)
4. 2 constante
5. – negatie unaire operatie (Temp2 = -2 wordt berekend)
6. + optellen binaire operatie
   (Resultaat = Temp1 + Temp2 wordt berekend)

Het gebruik van een expressieboom maakt het voor de compiler eenvoudiger code te genereren en optimalisaties toe te passen. Als er bijvoorbeeld in een expressieboom verscheidene keren dezelfde expressie voorkomt, dan kan het resultaat van deze expressie in een variabele worden gezet en op beide plekken worden gebruikt. Als we even teruggaan naar codevoorbeeld 2, dan zien we dat deze eigenlijk uit twee expressiebomen bestaat. De eerste expressieboom

```
Func<int, int> calc = (int x)_=> 3 * x + -2;
int value = calc(10);
Console.WriteLine(value); // 28
```

Codevoorbeeld 3. Lambda als Delegate

```
Expression<Func<int, int>> calcExpression = (int x) => 3 * x + -2;
Console.WriteLine(calcExpression.ToString()); // x => ((3 * x) + -2)
var Calc = calcExpression.Compile();

int value2 = Calc.Invoke(10);
Console.WriteLine(value2); // 28
```

Codevoorbeeld 4. Lambda als expressieboom

heeft 'where' als rootnode en de tweede heeft 'select' als rootnode. De .NET-compiler is echter slim genoeg om de tweede expressieboom weg te optimaliseren, omdat daar niet veel in gebeurt. Als de query ook nog een 'order by'-constructie had gehad, dan was dat ook een aparte expressieboom geworden. De compiler zorgt er in het geval van meerdere expressiebomen voor, dat deze als een soort ketting aan elkaar worden geregen (chaining). Het resultaat van de ene expressieboom wordt gebruikt als uitgangspunt bij de evaluatie van de volgende expressieboom. In het geval van een *where*- en *order by*-clause zal bij de evaluatie van de order by-expressieboom het resultaat van de evaluatie van de *where*-expressieboom worden gebruikt. Hierdoor is het mogelijk met behulp van verscheidene expressiebomen het uiteindelijke resultaat op te bouwen. De implementatie van de eBay LINQ-provider is bewust simpel gehouden, omdat het geheel vrij snel complex zou worden. Vanwege de genoemde simpliciteit wordt de *order by*-constructie niet ondersteund. Afbeelding 1 laat zien hoe de expressieboom er voor onze voorbeeldquery uitziet.
Op basis van de nodes in de expressieboom is eenvoudig een string te maken die de hele expressieboom voorstelt. De ToString-operatie maakt deze string en genereert de volgende representatie van de expressieboom:

```
{value(LinqToEbay.EbaySearch).Where(item => (item.Title.
Contains("Car") && Not((item.Title.Contains("Jeep") && item.
Title.Contains("NSX"))))))}
```

Uit de expressieboom valt af te leiden dat de compiler het hele gedeelte van de query na de *where* in een Lambda-expressie vertaalt. Voor het interpreteren van de expressieboom moet de body van de Lambda-expressie worden doorlopen, aangezien deze de condities van de query bevat.

IQueryable-interface

De IQueryable-interface is afgeleid van de Ienumerable-interface en heeft zowel een standaard als een generieke variant.
Voor het realiseren van de LINQ-provider voor eBay moet de IQueryable-interface worden geïmplementeerd. Met de introductie van Visual Studio 2008 Orcas Beta 2 is de IQueryable-interface gewijzigd en opgedeeld in twee aparte interfaces, namelijk de IQueryable- en de IQueryProvider-interface. De CreateQuery- en Execute-methodes zijn naar een aparte IQueryProvider-interface verplaatst, zie codevoorbeeld 5. Hierdoor is het mogelijk een generieke klasse te schrijven voor de IQueryable-interface met daar onder meer providers. De IQueryable-interface hoeft nog maar één keer te worden geïmplementeerd. In het geval dat er bijvoorbeeld meer webservices moeten worden ontsloten, krijgt elke webservice zijn eigen provider. Het is echter ook mogelijk beide interfaces op dezelfde klasse te implementeren, waardoor de ontwikkelaar maar van één klasse gebruik hoeft te maken. Om de implementatie zo simpel mogelijk te houden, is voor deze aanpak voor de eBay LINQ-provider gekozen.

De IQueryable-interface vereist de implementatie van de read-only properties *ElementType*, *Expression* en *Provider*. De property *ElementType* is eenvoudig en retourneert het type van het object dat in de query wordt teruggegeven. In het geval van de eBay LINQ-provider geeft deze methode het *EbayItem*-type terug. Dit type wordt gebruikt om de gevonden eBay-voorwerpen te representeren.
De *Expression*-property wordt door de .NET-compiler aangeroepen

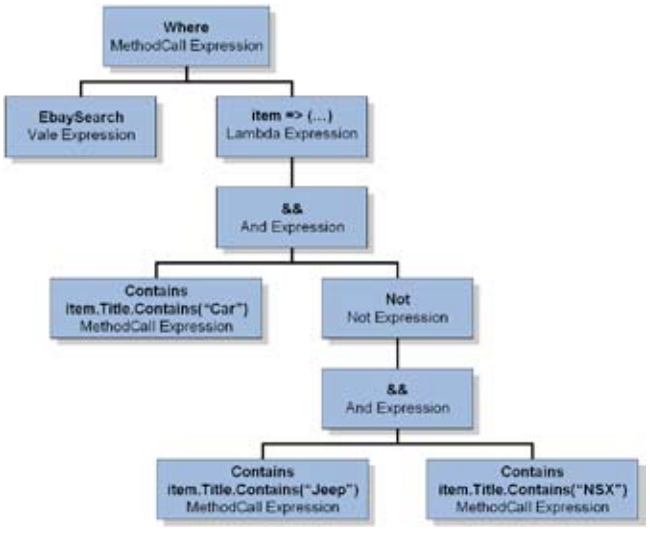
```
// Generieke IQueryable interface
public interface IQueryable<T> : IEnumerable<T>, IQueryable, IEnumerable
{
}

// Standaard IQueryable interface
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}

// IQueryProvider interface
public interface IQueryProvider
{
   IQueryable CreateQuery(Expression expression);
   IQueryable<TElement> CreateQuery<TElement>(Expression expression);
   object Execute(Expression expression);
   TResult Execute<TResult>(Expression expression);
}
```

Codevoorbeeld 5. IQueryable-interface

op het object dat de IQueryable-interface implementeert, wanneer deze een LINQ-query tegenkomt. Met deze property is het voor een LINQ-provider mogelijk zichzelf in een expressieboom te zetten. In het geval van de eBay LINQ-provider wordt het object *EbaySearch* als een *Expression.Constant* in de expressieboom gezet. Als we nog een keer naar afbeelding 2 kijken, dan zien we dat in de linkertak van de *Where*-node de *EbaySearch*-expressie terugkomt. Al het echte werk van de LINQ-provider, zoals het maken en uitvoeren van de query, vindt plaats in de instantie van de IQueryProvider-interface. De compiler kan via de *Provider*-property de instantie van deze interface ophalen en gebruiken.

IQueryProvider-interface

De instantie van de IQueryProvider-interface bevat de standaard en generieke variant van de *CreateQuery*- en *Execute*-methodes. De generieke varianten worden over het algemeen het meest gebruikt wanneer queries direct in de programmeertaal worden geschreven. Ze performen beter, omdat er geen gebruikgemaakt hoeft te worden van reflectie om instanties te maken vanuit de *expression.Type*. De implementatie van de standaard *CreateQuery*- en *Execute*-methodes is relatief eenvoudig, omdat zij de generieke varianten aanroepen met het *EbayItem*-type. Hierdoor blijven de twee generieke varianten over die nog moeten worden geïmplementeerd. Maar waar zijn deze twee belangrijke methodes nu eigenlijk voor



Afbeelding 2. Expressieboom LINQ-query

nodig? Een hint vinden we in de naamgeving van de methodes. De methode *CreateQuery* creëert op basis van een opgegeven expressieboom een nieuwe instantie van een IQueryable-query. In het geval van LINQ-to-SQL zal in de *CreateQuery* de SQL-query worden opgebouwd, zodat deze later naar de database kan worden gestuurd. In het geval van de eBay LINQ-provider gaan we de zoektekst opbouwen voor de webrequest die het uiteindelijke resultaat oplevert van onze zoekopdracht. In de implementatie van de *CreateQuery* zal rekening moeten worden gehouden met het feit dat aan de *CreateQuery* een willekeurige expressieboom kan worden meegegeven. Het kan dus ook zijn dat een expressieboom wordt meegegeven die niet kan worden vertaald in een correcte eBay-webrequest. De *CreateQuery* zorgt er voor dat er een vertaling komt van een LINQ-query naar een ander formaat dat waarschijnlijk een stuk minder rijk is aan features. De *CreateQuery* voert de query echter nog niet uit. Als met een debugger codevoorbeeld 1 wordt bekeken, dan wordt bij de query-definitie wel de *CreateQuery* uitgevoerd, maar zal er nog geen webrequest worden uitgevoerd (Delayed Execution). Deze wordt pas later bij het opvragen van de enumerator uitgevoerd. Zoals aangegeven moet de expressieboom worden vertaald naar een webrequest dat de eBay zoekwebservice ondersteunt. Tabel 2 geeft hiervan een overzicht. De voorbeeldquery moet dus met behulp van de expressieboom worden vertaald naar: “Car”, -(“Jeep”, “NSX”).

De expressieboom is, in het kader van de generatie van de eBay-webrequest, interessant vanaf de body van de Lambda-expressie; zie afbeelding 2 voor de expressieboom. Als we op die plek inhaken op de webrequest-generatie, dan is de tekst van de webrequest nog leeg en komen we in de expressieboom als eerste de *And Expression* tegen. Op dit moment kan er nog geen tekst aan de webrequest worden toegevoegd, omdat eerst de linker- en rechtertak van de expressieboom moeten worden doorlopen. Als eerste wordt de linkertak van de expressieboom bekeken. De node in de linkertak is een *MethodCall Expression*. In dit geval gaat het om de methode *Contains* die op de *Title*-property wordt uitgevoerd van het *Item*-object. Het object *Item* is van het type *EbayItem*. De definitie van het type *EbayItem* bevat echter niet de methode *Contains*. Voor LINQ is dit ook niet noodzakelijk en daarom wordt deze methode alleen maar gebruikt bij het definiëren van de query. De LINQ-provider moet echter wel weten wat het met een bepaalde methodeaanroep moet doen. Zo kan afhankelijk van de methodeaanroep de juiste code worden gegenereerd. Net zo als de *Where*-expressie bevat deze *MethodCall Expression* ook subexpressies. In dit geval heeft de *MethodCall Expression* één argument, namelijk de *Constant Expression* ‘Car’ en nog een *Member Expression* ‘Title’. Op basis van deze expressies kunnen we voor deze tak ‘Car’ als onderdeel van de webrequest genereren. De rechtertak is iets moeilijker. Hier komen we als eerste een *Not Expression* tegen. Net zoals bij de *And Expression* kunnen we pas tekst aan de webrequest toevoegen wanneer we de tekst voor de subexpressie hebben bepaald. Na de *Not Expression* komen we een *And Expression* tegen. Met deze constructie hebben we al eerder kennisgemaakt. Het resultaat van de linkertak is ‘Jeep’ en van de rechtertak is ‘NSX’. Na het bepalen van de tekst van de rechtertak is de hele expressieboom doorlopen en gaan we terug naar de node *Where*. Onderweg wordt de resterende tekst van de webrequest gegenereerd. Afbeelding 3 geeft per node aan wat de tekst is van de webrequest wanneer de expressieboom onder de node volledig is doorlopen. De rootnode geeft de volledige tekst van de webrequest weer en bevat het eindresultaat van de methode *CreateQuery*.

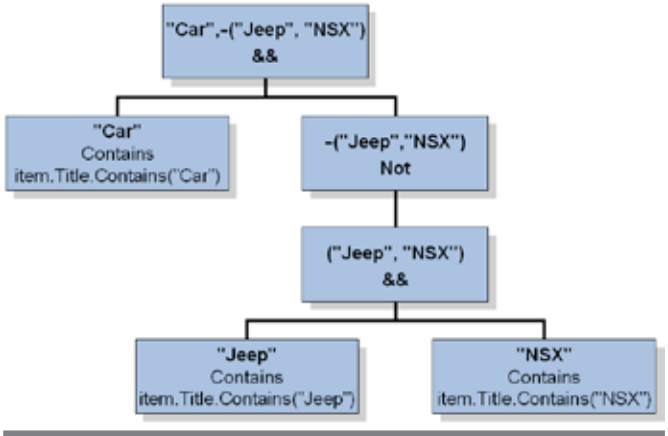
Operatie	eBay zoekformaat (X en Y zijn de zoekteksten)
AND	“X“, “Y”
OR	(“X” , “Y”)
NOT	-(“X”)

Tabel 2. eBay webservice-zoekformaat

Nu de webrequest is opgebouwd, moet deze alleen nog worden uitgevoerd. Hiervoor wordt de methode Execute gebruikt. Queries kunnen over het algemeen naast een resultset ook gewoon een enkele waarde opleveren Een voorbeeld hiervan zijn de Count- en Sum-functies. Deze worden ook wel scalar-functies genoemd. Om het voorbeeld simpel te houden, worden deze scalar-functies niet ondersteund. Dit vereenvoudigt de implementatie aanzienlijk. Hierdoor hoeft de methode Execute niet te worden geïmplementeerd en kan in de methode GetEnumerator (Ienumerable-interface) de daadwerkelijke webrequest worden uitgevoerd. Als we naar ons voorbeeld kijken, dan wordt de methode GetEnumerator aangeroepen zodra het foreach-statement wordt uitgevoerd. Dit komt perfect overeen met de Delayed Execution die we voor ogen hebben. Wanneer de provider echter wel scalar-functies moet ondersteunen, dan kan deze vereenvoudiging niet worden doorgevoerd en is het aan te raden om de Execute-methode niet alleen verantwoordelijk te maken voor de scalar-functies, maar ook voor de resultsets. De eBay-webrequest geeft op basis van de zoekvraag alle resultaten in één keer terug. Er zijn echter ook webrequests die werken op basis van pagina's, waarbij de resultaten in blokken worden teruggeven. Het werken met pagina's is eenvoudig te ondersteunen doordat in de GetEnumerator de resultaten met een yield return worden teruggegeven. Zodra een pagina-einde wordt bereikt, wordt de volgende webrequest gedaan. De GetEnumerator voert dus in plaats van één webrequest er meer uit. Hierdoor is het mogelijk de query-resultaatset eenvoudig met een foreach-lus te doorlopen. Voordat er met een resultaatset kan worden gewerkt, moeten de resultaten van de eBay-webrequest eerst worden geconverteerd. De resultaten worden namelijk in de vorm van een XML-file teruggegeven. Ze moeten ook nog worden omgezet naar eBayItems. Dit zijn de interne objecten met welke wordt gewerkt. De eBay-webservices zijn alleen te gebruiken als er een eBay-ontwikkelaarsaccount wordt gebruikt. Iedereen kan zo'n account aanvragen en de bijbehorende SDK downloaden. Met deze SDK is het mogelijk webservices op een eBay-testomgeving aan te roepen. Om de code die bij dit artikel hoort te kunnen gebruiken, moet er een eigen eBay-ontwikkelaarsaccount worden aangevraagd. De accountgegevens moeten in de 'app.config' worden ingevuld.

Nog meer LINQ-providers

Language Integrated Query draait niet alleen om SQL en XML en het gebruik daarvan in een .NET-taal. Met LINQ kunnen querymogelijkheden net zo goed worden losgelaten op andere databronnen zoals webservices. Het gebruik van LINQ op webservices impliceert wel dat er een provider wordt geschreven die de brug slaat tussen beide. Bij de implementatie van de Ebay-provider hebben we gekozen voor het rechtstreeks implementeren van entiteiten als EbayItem en de query-logica. Een mooiere variant zou echter zijn dat de WSDL van de webservice wordt gebruikt om de provider automatisch op te bouwen, zodat de oplossing voor meer webservices bruikbaar is. Een ontsloten webservice is prima te gebruiken als invoer voor bijvoorbeeld een LINQ-to-objects-query, waarin het resultaat van de Ebay-query wordt gebruikt om een sortering aan te brengen met behulp van LINQ-to-objects. Het is dus niet altijd noodzakelijk om zelf een sortering te implementeren. In de nabije toekomst zullen er vele LINQ-providers worden geschreven. Momenteel zijn er al providers voor LINQ to LDAP, LINQ to Flickr en LINQ To Google Image and Google Groups.



Afbeelding 3. Webrequesttekst per node

Patrick Vorgers is als technisch architect werkzaam bij de Management- en Consultancy-afdeling van Ordina Software Integration & Development (www.ordina.nl).

Ewart Nijburg is, vanuit zijn eigen onderneming, als technisch architect werkzaam bij diverse ondernemingen (www.troolean.nl). Hun specialisaties zijn softwarearchitecturen, high availability en software performance engineering. Voor vragen en opmerkingen kun je ze bereiken op enijburg@troolean.nl. en patrick.vorgers@ordina.nl

Referenties

The Linq project - http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx
Post-order Tree traversal - http://en.wikipedia.org/wiki/Tree_traversal
LINQ: .NET Language-Integrated Query - http://msdn2.microsoft.com/en-us/library/bb308959.aspx
Building an IQueryable Provider - http://blogs.msdn.com/mattwar/archive/2007/07/30/linq-building-an-iqueryable-provider-part-i.aspxeBay Windows Developer Center - http://developer.ebay.com/windows/