Sander Gerz
is in het dagelijks leven technisch architect voor Sogeti Nederland. Naast zijn gewone werkzaamheden beheert hij ook de website www.devtips.net.

Strongly Typed DataSets

KARAKTER EN MOGELIJKHEDEN

In dit artikel wordt de rol en het gebruik van de Strongly Typed DataSet, ook wel Typed DataSet, besproken. Een Typed DataSet is een specifieke implementatie van de DataSet. De auteur gaat in op het specifieke karakter en de mogelijkheden van de Typed DataSet. Vervolgens komt de plaats van de businesslogica ter sprake alsmede de performance in vergelijking met een DataReader. Ten behoeve van het begrip is het van belang eerst meer te weten over de basisvorm: de DataSet.

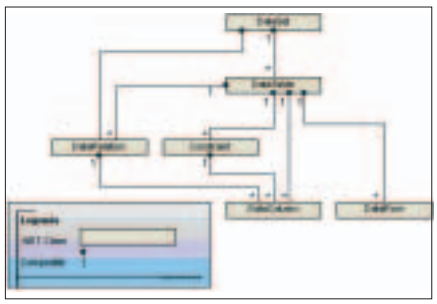
ADO.NET beschikt over twee objecten voor het ophalen van relationeel opgeslagen gegevens: de DataSet en de DataReader. De DataSet levert een relationele weergave van de gegevens in het geheugen en bevat de tabellen die de gegevens kan bevatten, sorteren en beperken alsmede de relaties tussen deze tabellen. De DataReader levert een snelle, forward-only en read-only datastroom van een database.

Wat is een DataSet?

Een DataSet bevat gegevens volgens een consistent relationeel programmeermodel en is onafhankelijk van de bron van deze gegevens. Je vindt de DataSet daarom ook in de System.Data namespace van het .NET Framework en niet in één van de namespaces die specifiek voor SQL Server (SqlClient) of ole-db databases (OleDb) zijn bedoeld. Een DataSet bevat tabellen die de gegevens sorteren en beperken alsmede de relaties tussen de verschillende tabellen; zie afbeelding 1.

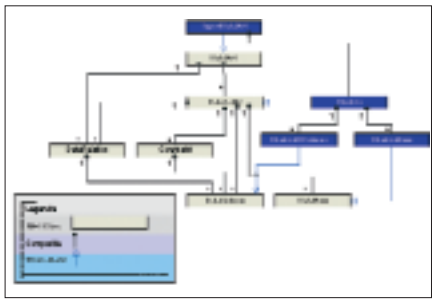
Je kunt een DataSet op verschillende manieren creëren. Deze manieren zijn ook nog te combineren. Een DataSet kan

- via programmacode worden opgebouwd door DataTables, DataRelations en Constraints te maken binnen een DataSet om vervolgens de tabellen met data te vullen.
- via een DataAdapter een DataSet met tabellen vullen vanuit een relationele database.
- via XML geladen en weer bewaard worden.



Afbeelding 1. Objectmodel van de DataSet

Wat maakt een DataSet 'Strongly Typed' Een Strongly Typed DataSet is te beschouwen als een databaseobject dat in het objectmodel van een .NET-project past. Een ontwikkelaar hoeft hiervoor niet zelf de code van het object te schrijven, maar laat dat over aan .NET. Doordat de code voor het object automatisch wordt gegeneerd kan zo'n object efficiënter en robuuster zijn dan wanneer de ontwikkelaar zelf de code moet schrijven; zie afbeelding 2.



Afbeelding 2. Objectmodel van de Strongly Typed DataSet

Zonder ADO.NET zouden de kolommen van een tabel handmatig in de DataSet moeten worden geprogrammeerd – met veldnamen en datatypen voor ieder van deze kolommen. Op deze manier zijn de methoden en eigenschappen van een DataSet wel beschikbaar in run-time, maar niet in design-time. Het doel van .NET is nu juist om de ontwikkelaar zowel in run-time als tijdens design-time een complete en functionele ontwikkelomgeving te bieden. Zodoende bieden strongly typed DataSets diverse voordelen. Je kunt beide soorten DataSets gebruiken in je applicatie. Typed DataSets maken het ontwikkelen eenvoudiger en minder foutgevoelig. De typed DataSet genereert een objectmodel waarin tabellen en kolommen class-objecten zijn. Je kunt daarom met een typed DataSet de code gebruiken zoals in

codevoorbeeld 1. Wanneer je een Untyped DataSet gebruikt, hanteer je de syntax zoals in voorbeeld 2.

Het creëren van een Typed DataSet

Nu we weten wat een Typed DataSet inhoudt, moeten we natuurlijk ook weten hoe een Typed DataSet wordt gegenereerd. Hier zijn twee manieren voor: via Visual Studio .NET, gebruikmakend van de Server Explorer, en via een command-line tool 'XSD.EXE'. De eerste en eenvoudigste manier om een Typed DataSet te creëren is door het slepen van een tabel of view uit een database naar een Form in de applicatie. Hierdoor wordt automatisch code aan het Form toegevoegd voor het instantiëren van een databaseverbinding en een volledig geconfigureerde DataAdapter. Het nadeel van deze aanpak is dat Visual Studio .NET dan een SELECT-statement opstelt die alle rijen en alle kolommen van de tabel selecteert, ongeacht of deze bestaat uit 40 kolommen en 40 miljoen regels. Voor kleinere tabellen (minder dan 1000 regels) werkt deze vorm overigens prima. Je kunt als alternatief ook een DataSet-item toevoegen aan je project. Wanneer je dan een tabel of view naar dit item sleept, wordt alleen de Typed DataSet class gegenereerd, zonder DataAdapter.

De tweede manier om een Typed DataSet te maken is ietwat omslachtiger. Deze wijze gaat uit van een XML Schema Document (XSD) waarvan via het command-line programma XSD.exe een Typed DataSet class wordt gegenereerd. XSD.exe zit in de .NET Framework SDK. Deze manier is vooral van toepassing voor hen die niet beschikken over Visual Studio .NET. Als je VS.NET wel hebt, is het beter de eerste methode te gebruiken, omdat deze automatisch een XSD-bestand oplevert. We kunnen wel een XSD-bestand opbouwen aan de hand van een (untyped) DataSet. In voorbeeldcode 3 staat hoe dat in zijn werk gaat.

Gegevens in de DataSet bewaren

Een lastige en foutgevoelige taak van de programmeur is het schrijven van de code om de gegevens in de database te manipuleren. In .NET Magazine #3 heeft Alex Thissen hier al een uitgebreid artikel over geschreven. We zullen er dan ook niet al te lang bij stilstaan. Omdat in ADO.NET de DataSet geabstraheerd is van de database, vooral via zogenaamde managed providers, hoef je weinig code te schrijven om bewerkingen op de database mogelijk te maken. Je kunt zelf de stored procedures of SQL-statements schrijven voor de create, read, update en delete (CRUD) bewerkingen en deze stored procedures koppelen aan de DataAdapter. Het is ook mogelijk om deze statements automatisch te genereren via de CommandBuilder. Zo wordt het meeste werk door ADO.NET uit handen genomen.

```
' Visual Basic
' Hiermee bereiken we de title_id kolom van de eerste
' rij in de titles tabel.
s = dsPubs1.titles(0).title_id
```

```
// C#
// Hiermee bereiken we de title_id kolom van de eerste
// rij in de titles tabel.
s = dsPubs1.titles[0].title_id;
```

Voorbeeldcode 1

```
'Visual Basic
s = dsPubs1.Tables("titles")(0).Columns("title_id")

//C#
s = dsPubs1.Tables["titles"][0].Columns["title_id"];
```

Voorbeeldcode 2

```
// maak een DataAdapter voor elke tabel die we aanspreken.
SqlDataAdapter daCustomers =
         new SqlDataAdapter("SELECT * FROM Employees", conn);

// Instantieer een lege DataSet
DataSet dataSet = new DataSet();

// Vul de DataSet via de DataAdapter(s)
daCustomers.Fill(dataSet, "Employees");

// Refereer een DataTable in de DataSet
DataTable employeesTable = dataSet.Tables["Employees"];

// Verbeter het schema
employeesTable.Columns["EmployeeID"].ReadOnly = true;
employeesTable.Columns["EmployeeID"].Unique = true;
employeesTable.Columns["LastName"].MaxLength = 50;
employeesTable.Columns["LastName"].AllowDBNull = false;
employeesTable.Columns["FirstName"].MaxLength = 50;
employeesTable.Columns["FirstName"].AllowDBNull = false;
employeesTable.Columns["Address"].MaxLength = 129;
employeesTable.Columns["Country"].DefaultValue = "NL";
employeesTable.Columns["State"].MaxLength = 2;
employeesTable.Columns["HomePhone"].Unique = true;

// Schrijf het XML Schema voor de
// typed DataSet
dataSet.WriteXmlSchema(@"c:\EmployeesDS.xsd");
```

Voorbeeldcode 3

De businesslaag

Wat overblijft is de businesslogica, en dat kan eenvoudig zijn of lastig. Businesslogica bestaat veelal uit extra regels die moeten worden toegepast op de gegevens in een database. De meest voor de hand liggende oplossing bij de bepaling waar je deze logica moet plaatsen, ligt in het erven van de gegenereerde Typed DataSet. Omdat een Typed DataSet een volwaardige .NET class is, kan je erven van deze class en er methoden aan toevoegen of overriden waar dat nodig is.

Voor een programmeur is het schrijven van de businessobjectlaag een veel voorkomende klus. Het werk dat hierbij gepaard gaat bestaat veelal uit de volgende taken:

1. het mappen van een relationeel model naar een hiërarchisch model,
2. het toevoegen van specifieke businessregels die op de data van toepassing zijn,
3. het lezen en weerschrijven van objectstate van en naar de database, waarover in de vorige paragraaf al is geschreven.

De mapping van het relationele model naar een hiërarchisch model, en het schrijven van de code voor het wijzigen van de data in een database zijn de meest tijdrovende werkzaamheden. Door gebruik te maken van Typed DataSets is het vaak niet meer noodzakelijk deze code zelf te schrijven. Je kunt een typed DataSet opbouwen met meerdere tabellen en relaties, waardoor je automatisch de relationeel-naar-object mapping voor elkaar hebt. Wanneer je een enkele rij uit een enkele tabel beschouwt als de top van een objecthiërarchie, dan kun je door de relaties navigeren om een boomstructuur van objecten te krijgen. Er zijn twee mogelijkheden voor de implementatie van de businesslogica aan de hand van een Typed DataSet. In beide opties creëren we eerst een class die erft van de Typed DataSet.

```
public class KlantenObject : TDSKlanten
{
    public KlantenObject() : base()
    {
        Register();
    }

protected KlantenObject(SerializationInfo info,
   StreamingContext context) : base(info, context)
    {
        Register();
    }
    private void Register()
    {
    Factuur.FactuurRowChanging +=
              new FactuurRowChangeEventHandler(FactuurChanging);
    }

    private void FactuurChanging(object source,
       FactuurRowChangeEvent args)
    {
        if (args.Action == DataRowAction.Add ||
            args.Action == DataRowAction.Change)
        {
            if (args.Row.FactuurDate > DateTime.Today)
            {
                throw new Exception(
                "Facturen kunnen niet in de toekomst gemaakt worden);
            }
        }
    }
}
```

Voorbeeldcode 4

Event-driven DataSet

Wanneer je niet veel businesslogica hebt kun je volstaan met het erven van de Typed DataSet en het implementeren van event-notificaties voor de afhandeling van de eventuele businessregels. In een DataSet kun je event-notificaties krijgen van diverse events, maar de meest voor de hand liggende zijn de RowChanging en RowChanged events; zie codevoorbeeld 4.

We maken eerste een nieuwe class die erft van de Typed DataSet (TDSKlanten) en vervolgens maken we twee constructors. De eerste is die voor normale instantiëring, de tweede is bedoeld voor XML-deserialisatie. Deze tweede constructor moet geïmplementeerd worden, anders is er geen ondersteuning voor XML (de)serialisatie. In de constructor wordt naast het aanroepen van de constructors van de base-class niets anders gedaan dan het aanroepen van de nieuwe Register-methode die we gebruiken om die events te registreren waarin we zijn geïnteresseerd. In dit geval zijn we geïnteresseerd in de FactuurRowChangingEvent. Dit event wordt afgevuurd voordat de rij feitelijk wordt aangepast. In de methode die het event behandelt (de handler FactuurChanging), controleren we of de aangepaste rij was toegevoegd of veranderd. In beide situaties wordt gecontroleerd dat de datum van de factuur niet in de toekomst is. Als dat wel het geval is, werpt de code een exception op.

In voorbeeldcode 5 zie je hoe de Typed DataSet wordt gebruikt.

Het gebruik van de afgeleide Typed DataSet is identiek aan het gebruik van de originele Typed DataSet. Maar omdat we ons hebben aangemeld voor events, zullen bepaalde handelingen gemeld worden zodat we deze in onze businesslogica kunnen afhandelen. In de voorbeeldcode maken we een factuur met een datum die 8 dagen in de toekomst ligt. Zodra het

```
// Creëer een DataAdapter voor iedere tabel die we gaan gebruiken
SqlDataAdapter daKlanten =
       new SqlDataAdapter("SELECT * FROM KLANTEN;", conn);

// Creëer de Factuur DataAdapter
SqlDataAdapter daFacturen =
       new SqlDataAdapter("SELECT * FROM FAKTUREN", conn);

// Instantieer onze Typed DataSet (die erft van de
// gegenereerde Typed DataSet)
KlantenObject dataset = new KlantenObject();

// Use the DataAdapters to fill the DataSet
daKlanten.Fill(dataset, "Klant");
daFacturen.Fill(dataset, "Factuur");

// This will throw an exception because we're creating
// an Factuur date in the future
TDSKlanten.FactuurRow Factuur;
Factuur = dataset.Factuur.AddFactuurRow(Guid.NewGuid(),
                       DateTime.Now + new TimeSpan(8,0,0,0),
                       "",
                       "",
                       "",
                       dataset.Klant[0]);
```

Voorbeeldcode 5

event (FactuurRowChanging) wordt afgevuurd, zal een exception optreden en weet de gebruiker dus dat iets is misgegaan.

Erven van DataTable en DataRow

Een andere aanpak om businesslogica te implementeren bestaat niet alleen uit het erven van de Typed DataSet, maar ook van de Typed DataTable en de Typed DataRow classes. Hiermee kunnen we ieder gedrag 'overriden' om specifieke businessregels te implementeren. We hoeven niet van iedere DataTable en iedere DataRow te erven, alleen die classes die specifieke businessregels hebben. Als we willen erven van een DataRow, dan moeten we erven van de DataTable waartoe de DataRow behoort. Door een eigen implementatie van de Rows.Add-methode kan je ook eigen code toevoegen voor het toestaan of weigeren van een nieuwe rij in de tabel. Je bent zo in staat complexere businessregels te schrijven waarbij je nog steeds profiteert van de type-safety van de Typed DataSet en de updatemechanismen van de DataAdapter.

Performance

In de inleiding is kort ook de DataReader aangehaald. De DataReader is een tweede vorm van databasebenadering die via ADO.NET beschikbaar is. Het belangrijkste verschil tussen een DataSet en een DataReader is, zoals de naam al doet vermoeden, dat de gegevens die via een DataReader beschikbaar komen alleen gelezen kunnen worden. Het is dus niet mogelijk om met de DataReader updates te verzorgen. Dat heeft als voordeel dat het DataReader-object zeer klein kan zijn, zeker in vergelijking met de veel grotere DataSet.

Een DataSet haalt via het SelectCommand van de DataAdapter alle gegevens op om ze lokaal bewaren. Daarna is de verbinding met de database gesloten. De DataReader houdt echter de verbinding met de database zolang vast als nodig is om door de regels in de geselecteerde tabel of view te lopen. Het ligt dan ook voor de hand dat het initialiseren van een DataSet meer tijd kost dan van een DataReader. Een select-statement die 10.000 regels van 2kB elk ophaalt kan al snel 20MB aan intern geheugen vergen. Het is met name voor webapplicaties van belang dat dergelijke DataSets niet worden gebruikt op het niveau van gebruikersessies. Je zou eens honderd gebruikers tegelijk in de webapplicatie hebben! Het verschil in performance tussen de Typed DataSet en een Untyped DataSet is overigens nihil.

Minder risico's

We kunnen het er allemaal wel over eens zijn dat type-safety een belangrijk goed is. Via Typed DataSets lopen we minder risico's op programmeerfouten en misbruik van gegevens in een database. In dit artikel is getoond hoe je een Typed DataSet maakt en hoe deze is te gebruiken.

De conclusie is dan ook dat een strongly typed DataSet

- prima past in het .NET objectmodel waarbij de classmembers volledig zichtbaar zijn
- beter gebruikt kan worden in de tools van Visual Studio .NET, zoals de DataGrid
- kan worden gegenereerd via een databaseconnectie die een DataSet en een XSD-bestand tijdens het proces creëert, of kan gegenereerd worden vanuit een XSD-bestand.

Een Typed DataSet, maar ook een Untyped DataSet, is niet de oplossing voor alle databasegerelateerde bewerkingen. Het disconnected karakter van de DataSet maakt dat alle benodigde data lokaal worden onthouden. Bij tabellen met veel kolommen (> 40) en veel rijen (> 1000) is het beter om Select-statements te gebruiken die alleen de strikt noodzakelijke data ophalen. Wanneer dat niet mogelijk is, kan beter gebruik worden gemaakt van de DataReader.

Meer informatie:

- www.adoguy.com
- "Designing Data Tier Components and Passing Data Through Tiers", msdn.microsoft.com
- "ADO.NET PowerToys Project", www.gotdotnet.com/workspaces