Patrick Boom en Frank Robijn zijn werkzaam bij Capgemini

Objecten en SQL Server 2005: XML als intermediair

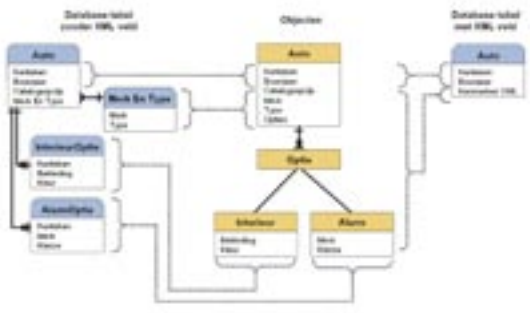
PERSISTENTIE VAN OBJECTMODELLEN IN XML-VELDEN VAN SQL SERVER

Objecten met businesslogica, data access-laag, database: het zijn bijna altijd de onderste drie lagen in een architectuurplaatje van een applicatie. De data access-laag is nodig, omdat de objectgeoriënteerde structuur van de business-logica niet gemakkelijk past op de relationele structuur van databasetabellen. Lastig, want als er iets aan een object verandert, moet ook de database aangepast worden. Een object in een enkel databaseveld stoppen is not done, want dan kunnen de krachtige relationele functies van een database niet meer worden gebruikt.

Dat is voorbij, dankzij de samenwerking tussen .NET en SQL Server 2005. De laatste introduceert een nieuw (XML) veldtype dat via SQL/XPath-queries ondervraagd kan worden, en dat ook kan deelnemen in databaserelaties en indexing. Daarover heeft Peter ter Braake al uitgebreid geschreven in .NET Magazine #7. Het .NET Framework maakt het eenvoudig om objecten naar XML om te zetten. Met een juiste mix van opslag als XML en opslag in tabelvelden kan de data access-laag veel eenvoudiger en sneller uitgevoerd worden en is ze veel beter bestand tegen veranderingen in de businesslogica.

Een voorbeeld: zoals het is…

De voordelen zijn duidelijk te maken aan de hand van een voorbeeld. Stel: een leasemaatschappij vraagt om een applicatie voor klantondersteuning. Een medewerker moet snel kunnen inzien welke auto en welke klant bij elkaar horen, maar moet ook kunnen vastleggen wat de kenmerken van een nieuwe leaseauto van een klant zijn. Mailings worden vaak uitgestuurd op basis van een beperkt aantal kenmerken, zoals bouwjaar. De gegevens worden opgeslagen in een database, die gesynchroniseerd wordt met andere systemen. Een gebruikelijke keuze bij het ontwerpen van de database is om de eigenschappen van objecten één op één op tabelkolommen leggen; zie afbeelding 1. Dat leidt al snel tot onoverzichtelijke databaseschema's, die bovendien veranderlijk zijn. Tegenwoordig is een navigatiesysteem een normale optie, maar om die in de applicatie te krijgen moet er niet alleen een nieuw object gemaakt worden, maar ook nieuwe tabelkolommen of zelfs een nieuwe tabel. En dus moeten er extra testen worden uitgevoerd om te controleren dat alle andere code nog blijft werken. Dat is een zware ingreep als het om een optie gaat die niet raakt aan de kern van de applicatie.



Afbeelding 1. Het vertalen van een stel objecten naar een databaseschema kan op twee

Het kan beter met XML-velden

Het kan ook anders door XML-velden te gebruiken. De eigenschappen waarop het meest wordt gezocht, zoals het kenteken, worden nog steeds als kolom in een tabel opgenomen, maar alle andere eigenschappen en objecten worden naar XML vertaald en in een XML-veld opgeslagen; zie afbeelding 1. Het vertalen van een stel objecten naar een databaseschema kan nu op twee manieren: alle eigenschappen van objecten opnemen als tabelvelden, of alleen de belangrijkste en de overige gegevens als XML opslaan. De objectstructuur en het databaseschema zijn voor een groot deel ontkoppeld: als er een optie bij komt heeft dat alleen gevolgen voor de inhoud van het XML-veld. Bovendien vereist het opslaan en terughalen van een Auto met Optie-objecten veel minder database-interactie. Als er incidenteel gezocht moet worden in de opties, bijvoorbeeld voor een jaarlijkse bepaling van het aandeel van de verschillende merken (codevoorbeeld 1), kunnen de nieuwe SQL/XPath query-mogelijkheden gebruikt worden. Zie ook het artikel van Peter ter Braake.

Weinig applicaties zijn nog stand-alone; vrijwel altijd moet er data uitgewisseld worden met andere applicaties. Vaak wordt dan eerst het objectmodel opgebouwd uit de database, de verschillende objecteigenschappen gecombineerd in afgesproken datastructuren, en het resultaat, gecodeerd als XML, verstuurd naar de andere applicaties. Als een XML-veld gebruikt is, kan de XML ook rechtstreeks via een XSLT-transformatie aangemaakt worden zonder de tussenstap van het objectmodel.

Lezen en schrijven van objecten

Het lezen en schrijven van XML kan via standaard ADO.NET-klassen en het SqlXml-type. Het SqlXml- type is niet gebaseerd op een XML DOM; het biedt geen mogelijkheden voor het direct

```
SELECT Bouwjaar, Merk, COUNT (Merk) as Aantal
FROM (
  SELECT Bouwjaar, [xml].value('(/Auto/Merk)[1]', 'nvarchar(50)') as
Merk FROM Auto
  ) as DATA
GROUP BY Bouwjaar, Merk
```

Codevoorbeeld 1.

```
public partial class Auto
{
    private void Write (SqlConnection conn, Stream data)
  {
    SqlXml xml = new SqlXml (data);
      conn.Open ();
    SqlCommand comm = new SqlCommand (@"INSERT INTO Auto (Kenteken,
                                       CatalogusPrijs, BouwJaar, [xml])
        VALUES (@Kenteken, @CatalogusPrijs, @BouwJaar, @Xml)", conn);
    comm.Parameters.Add ("Kenteken", SqlDbType.NChar).Value = this.Kenteken;
    comm.Parameters.Add ("CatalogusPrijs", SqlDbType.Money).Value =
                        this.CatalogusPrijs;
    comm.Parameters.Add ("BouwJaar", SqlDbType.DateTime).Value = this.BouwJaar;
    comm.Parameters.Add ("Xml", SqlDbType.Xml).Value = xml;
    comm.ExecuteNonQuery ();
  }
  private static string Read (SqlConnection conn, string kenteken)
  {
    conn.Open ();
    SqlCommand comm = new SqlCommand (@"SELECT [xml] FROM Auto WHERE
                                       kenteken = @Kenteken", conn);
    comm.Parameters.Add ("Kenteken", SqlDbType.NChar).Value = kenteken;
    using (SqlDataReader reader = comm.ExecuteReader ())
    {
      if (reader.Read ())
      {
        return reader.GetSqlXml (0).Value;
      }
    }
    return null;
  }
}
```

Codevoorbeeld 2. Lezen en schrijven van gegevens via het SqlXml-type

```
  [Serializable] // markeer klasse als serializable.
  public partial class Auto
  {
    public static Auto Create (SqlConnection conn, string kenteken)
    {
      return Deserialize (Read (conn, kenteken));
    }

    public void Save (SqlConnection conn)
    {
      Write (conn, Serialize (this));
    }
    public string Kenteken
    {
      get { return _kenteken; }
      set { _kenteken = value; }
    }
    private string _kenteken = "AA-12-34";
    public string Merk
    {
      get { return _merk; }
      set { _merk = value; }
    }
    private string _merk = "Volkswagen";
    public string Type
    {
      get { return _type; }
      set { _type = value; }
    }
    private string _type = "Golf V 1.6 benzine Highline";
    public DateTime BouwJaar
    {
      get { return _bouwjaar; }
      set { _bouwjaar = value; }
    }
    private DateTime _bouwjaar = new DateTime (2006, 6, 6);
    public double CatalogusPrijs
    {
      get { return _catalogusPrijs; }
      set { _catalogusPrijs = value; }
    }
    private double _catalogusPrijs = 25000;
    public List<Optie> Opties
    {
      get { return _opties; }
      set { _opties = value; }
    }
    private List<Optie> _opties = new List<Optie> ();
  }
[Serializable]
public abstract class Optie
{
}
[Serializable]
public class Interieur : Optie
{
  // ... properties ...
}
[Serializable]
public class Alarm : Optie
{
  // ... properties ...
}
```

Codevoorbeeld 3. Het objectmodel uit afbeelding 1 als serialiseerbare klassen

bewerken of uitlezen van XML. Het type kan alleen uit een stream worden geïnstantieerd en als een string of XmlReader worden uitgelezen. In codevoorbeeld 2 is in de Read-method de XML als string teruggegeven. Om de objecten van en naar een stream om te zetten, biedt .NET een standaard mechanisme: serialisering. Een klasse kan eenvoudig serialiseerbaar gemaakt worden (zie codevoorbeeld 3) door het toevoegen van het Serializable-attribuut, en te zorgen dat alle eigenschappen die opgeslagen moeten worden als read/write-property zijn gedeclareerd. De uiteindelijke vertaling van en naar een stream wordt gedaan door een serializer, die bepaalt hoe de objecten worden weggeschreven. Voor XML beschikt .NET over de XmlSerializer.

Serialiseren met de XmlSerializer

De XmlSerializer maakt gebruik van reflectie om uit te zoeken welke gegevens moeten worden opgeslagen. De serializer kijkt hierbij naar de publieke properties en velden van een object. Bij het deserialiseren wordt een nieuw object aangemaakt via de default constructor en de velden en properties van het object gezet. Daarom moeten de properties writable zijn. De serializer werkt recursief: alle child-objecten worden ook geserialiseerd. De XmlSerializer moet van tevoren weten

```
  public partial class Auto {

    private static XmlSerializer _serializer = new XmlSerializer
        (typeof (Auto), new Type[]{typeof(Interieur), typeof(Alarm)});

    private static MemoryStream Serialize (Auto auto)
    {
      MemoryStream stream = new MemoryStream ();
      _serializer.Serialize (stream, auto);
      stream.Position = 0; // reset de stream naar het begin
      return stream;
    }

    private static Auto Deserialize (string xml)
    {
      Auto auto = null;
      using (MemoryStream stream = new
                MemoryStream (Encoding.UTF8.GetBytes (xml)))
      {
        auto = (Auto) _serializer.Deserialize (stream);
      }
      return auto;
    }
  }
```

Codevoorbeeld 4. Serialiseren en deserialiseren van objecten

```
<Auto xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Kenteken>01-AA-BB</Kenteken>
  <Merk>Saab</Merk>
  <Type>93</Type>
  <BouwJaar>2002-01-01T00:00:00</BouwJaar>
  <CatalogusPrijs>30000</CatalogusPrijs>
  <Opties>
    <Optie xsi:type="Interieur">
      <Kleur>beige</Kleur>
      <Bekleding>leder</Bekleding>
    </Optie>
    <Optie xsi:type="Alarm">
      <Merk>OEM</Merk>
      <Klasse>A</Klasse>
    </Optie>
  </Opties>
</Auto>\
```

Codevoorbeeld 5. Het Auto-object geserialiseerd naar XML

welke typen hij tegen gaat komen. Bij het aanmaken van de serializer wordt dynamisch code gegenereerd om elk type snel te lezen en schrijven. Als de serializer een type tegenkomt dat hij niet kent – bijvoorbeeld omdat het object van een afgeleid type is – dan weet hij niet meer wat hij moet doen. Dat komt voor bij de Auto.Opties-property: de serializer denkt dat de objecten van het type Optie zijn, maar in werkelijkheid gaat het om Interieur- of Alarmobjecten. De afgeleide klassen moeten expliciet worden meegegeven of via een XmlInclude-attribuut bekend gemaakt worden. In codevoorbeeld 4 zijn de serialisering en deserialisering uitgewerkt. Het aanmaken van de XmlSerializer

kost relatief veel tijd vanwege het aanmaken van de dynamische code. Om die reden is het efficiënter de XmlSerializer als static field op te nemen. De resulterende XML is weergegeven in codevoorbeeld 5.

Serialiseren met de XmlSerializer

De XmlSerializer maakt gebruik van reflectie om uit te zoeken welke gegevens moeten worden opgeslagen. De serializer kijkt hierbij naar de publieke properties en velden van een object. Bij het deserialiseren wordt een nieuw object aangemaakt via de default constructor en de velden en properties van het object gezet. Daarom moeten de properties writable zijn. De serializer werkt recursief: alle child- objecten worden ook geserialiseerd. De XmlSerializer moet van tevoren weten welke typen hij tegen gaat komen. Bij het aanmaken van de serializer wordt dynamisch code gegenereerd om elk type snel te lezen en schrijven. Als de serializer een type tegenkomt dat hij niet kent – bijvoorbeeld omdat het object van een afgeleid type is – dan weet hij niet meer wat hij moet doen. Dat komt voor bij de Auto.Opties-property: de serializer denkt dat de objecten van het type Optie zijn, maar in werkelijkheid gaat het om Interieur- of Alarmobjecten. De afgeleide klassen moeten expliciet worden meegegeven of via een XmlInclude-attribuut bekend gemaakt worden. In codevoorbeeld 4 zijn de serialisering en deserialisering uitgewerkt. Het aanmaken van de XmlSerializer kost relatief veel tijd vanwege het aanmaken van de dynamische code. Om die reden is het efficiënter de XmlSerializer als static field op te nemen. De resulterende XML is weergegeven in codevoorbeeld 5.

Meer controle over serialisering

Het gedrag van de XmlSerializer is maar beperkt te beïnvloeden. Het is mogelijk de structuur van de XML aan te geven. Bijvoorbeeld of een property naar een XML-element of attribuut vertaald wordt. Het is ook te voorkomen dat properties of velden geserialiseerd worden door het toevoegen van het XmlIgnore-attribuut. Maar het is niet mogelijk niet-publieke properties en velden zichtbaar te maken voor de serializer: een object wordt geserialiseerd zoals de buitenwereld het object kan zien. .NET kent wel een manier om zeer veel controle uit te oefenen over de serialisering, via de ISerializable-interface. Een object dat deze interface implementeert, heeft een GetObjectData-methode waarin exact aangegeven staat wat moet worden geserialiseerd. Het object heeft een speciale constructor die het object vanuit serialisering weer opbouwt. Een voorbeeld is gegeven in codevoorbeeld 6. Helaas ondersteunt de XmlSerializer deze interface niet. Standaard formatters die dat wel doen zijn de BinaryFormatter en de Soap-

```
public partial class Auto {

   private static MemoryStream Serialize (Auto autoObject)
   {
      SoapFormatter serializer = new SoapFormatter ();
      MemoryStream stream = new MemoryStream ();
      serializer.Serialize (stream, autoObject);
      stream.Position = 0; // reset de stream naar het begin
      return stream;
   }

   private static Auto Deserialize (string xml)
   {
      Auto auto = null;
      SoapFormatter serializer = new SoapFormatter ();
      using (MemoryStream stream = new
           MemoryStream (Encoding.UTF8.GetBytes (xml)))
      {
        auto = (Auto) serializer.Deserialize (stream);
      }
      return auto;
   }
}
```

Codevoorbeeld 7. Serialisering en deserialisering met de SoapFormatter; dit vervangt de code uit codevoorbeeld 4.

```
public partial class Auto : ISerializable
{
   // serializatie
   public void GetObjectData (SerializationInfo info, StreamingContext
context)
   {
      info.AddValue ("Kenteken", Kenteken);
      info.AddValue ("Merk", Merk);
      info.AddValue ("Type", Type);
      info.AddValue ("BouwJaar", BouwJaar);
      info.AddValue ("CatalogusPrijs", CatalogusPrijs);
      info.AddValue ("Opties", Opties.ToArray (), typeof (Optie[]));
   }

   // speciale constructor voor deserializatie
   public Auto (SerializationInfo info, StreamingContext context)
   {
      _kenteken = info.GetString ("Kenteken");
      _merk = info.GetString ("Merk");
      _type = info.GetString ("Type");
      _bouwjaar = info.GetDateTime ("BouwJaar");
```

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://
schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://schemas.
xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.com/
soap/encoding/clr/1.0" SOAP-ENV:encodingStyle="http://schemas.xmlsoap.
org/soap/encoding/">
  <SOAP-ENV:Body>
    <a1:Auto xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Capge
mini.DotNetMagazine.September.Examples/Artikel%2C%20Version%3D1.0.0.0%2C
%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull" id="ref-1">
      <Kenteken id="ref-3">02-AA-BB</Kenteken>
      <Merk id="ref-4">Saab</Merk>
      <Type id="ref-5">93</Type>
      <BouwJaar xsi:type="xsd:dateTime">2002-01-01T00:00:00.0000000+01
:00</BouwJaar>
      <CatalogusPrijs>30000</CatalogusPrijs>
      <Opties href="#ref-6" />
    </a1:Auto>
    <SOAP-ENC:Array xmlns:a1="http://schemas.microsoft.com/clr/nsassem/
Capgemini.DotNetMagazine.September.Examples/Artikel%2C%20Version%3D1.0.
0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull" id="ref-6" SOAP-
ENC:arrayType="a1:Optie[2]">
      <item href="#ref-7" />
      <item href="#ref-8" />
    </SOAP-ENC:Array>
    <a1:Interieur xmlns:a1="http://schemas.microsoft.com/clr/nsassem/
Capgemini.DotNetMagazine.September.Examples/Artikel%2C%20Version%3D1.0.0
.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull" id="ref-7">
      <_kleur id="ref-9">beige</_kleur>
      <_bekleding id="ref-10">leder</_bekleding>
    </a1:Interieur>
    <a1:Alarm xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Capge
mini.DotNetMagazine.September.Examples/Artikel%2C%20Version%3D1.0.0.0%2C
%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull" id="ref-8">
      <_merk id="ref-11">OEM</_merk>
      <_klasse id="ref-12">A</_klasse>
    </a1:Alarm>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Codevoorbeeld 8. Het Auto-object als SoapMessage

```
public partial class Auto
{

   private static MemoryStream Serialize (Auto auto)
   {
      MemoryStream stream = new MemoryStream ();
      XmlFormatter serializer = new XmlFormatter (typeof (Auto));
      serializer.Serialize (stream, auto);
      stream.Position = 0; // reset de stream naar het begin
      return stream;
   }
   private static Auto Deserialize (string xml)
   {
      Auto auto = null;
      XmlFormatter serializer = new XmlFormatter (typeof (Auto));
      using (MemoryStream stream = new MemoryStream (
             Encoding.UTF8.GetBytes (xml)))
      {
         auto = (Auto) serializer.Deserialize (stream);
      }

      return auto;
   }
}
```

Codevoorbeeld 9. Serialisering en deserialisering met onze XmlFormatter; dit vervangt de code uit codevoorbeeld 4

Formatter. De eerste levert geen XML terug en is dus niet bruikbaar voor ons doel. De tweede is vooral bedoeld voor webservices en resulteert in XML die moeilijk te benaderen is met XPath-queries in de database; zie codevoorbeeld 7 en codevoorbeeld 8.

IFormatter-interface

Het is mogelijk zelf een formatter te maken door de IFormatter-interface te implementeren. Dit houdt in dat we zelf code schrijven die het object en eventuele child-objecten afloopt en de waarden als XML wegschrijft en inleest. Het schrijven van een goede eigen formatter is echter lastig. De formatter moet een aantal interfaces ondersteunen om de verschillende gebruiksscenario's aan te kunnen. Daarnaast vereist de vertaling van XML-elementen naar .NET-types en vice versa een behoorlijke administratie. Het beschrijven van een eigen formatter valt buiten de scope van dit artikel. Wij hebben gemerkt dat op internet weinig voorbeelden en werkwijzen te vinden zijn. Daarom hebben wij een door ons geschreven XmlFormatter ter download op Codeproject gezet. Het is een complete formatter die alle basisfunctionaliteit heeft; zie codevoorbeeld 9. De broncode is openbaar; dus hij kan worden aangepast aan specifieke wensen, bijvoorbeeld voor het sturen van de gegenereerde XML.

In de praktijk

De beschreven combinatie van objecten en XML in SQL Server 2005 maakt het mogelijk snel een flexibele data access-laag te bouwen. In ons huidige project hebben we in eerste instantie gewerkt met de standaard XmlSerializer. Dit heeft ons veel werk bespaard in de tijd dat het objectmodel voortdurend veranderde, omdat de ontwikkel- en testdatabases niet steeds aangepast hoefden te worden. Dit woog ruim op tegen de nadelen: verminderde performance en veel publieke properties. We zijn later overgestapt naar een eigen serializer die een betere performance geeft en die ons in staat stelde het objectmodel beter af te schermen. Ondanks een redelijk uitgebreid objectmodel is het databaseschema erg compact en overzichtelijk gebleven. De voorbeeldcode uit dit artikel is te downloaden van www.microsoft.nl/netmagazine14

Patrick Boom en Frank Robijn zijn werkzaam bij Capgemini Nederland als software-engineer en softwarearchitect. www.nl.capgemini.com patrick.boom@capgemini.com frank.robijn@capgemini.com

Referenties

MSDN Magazine #7: http://www.microsoft.nl/netmagazine7

Codeproject XmlFormatter: www.codeproject.com/XmlFormatter

XML in SQL Server 2005: http://msdn.microsoft.com/sql/learning/prog/xml/default.aspx

Controlling Serialization Using Attributes: http://msdn2.microsoft.com/en-us/library/2baksw0z.aspx