Peter van Ooijen is zelfstandig .NET-consultant en ontwikkelaar

Met het Compact Framework het internet op

HET WEB ALS INFRASTRUCTUUR

Het internet is alom aanwezig. Voor een bedrijf is het vanzelfsprekend om continu met het web verbonden te zijn, voor menig huishouden begint dat een vanzelfsprekendheid te worden en internet op je mobieltje is ook al niets bijzonders meer. In dit artikel wil ik dieper ingaan op twee aspecten: hoe beperk je de hoeveelheid dataverkeer en hoe ga je om met de inherente trage reactiesnelheid van het web? Twee aspecten die bij het ontwikkelen, als het verkeer jouw machine of LAN niet afkomt, niet zo snel op zullen vallen. Maar bij het gebruik kunnen ze een applicatie maken of breken.

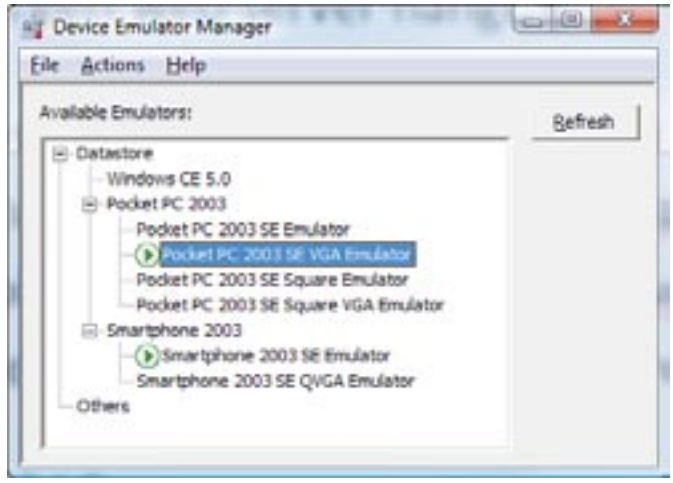
Het web verbindt dus zo langzamerhand alles met alles. Dit is infrastructuur waarmee je de wildste fantasieën op ICT-gebied kunt realiseren. Zeker als je bedenkt in wat voor bochten je je vroeger moest wringen om de onderdelen van jouw oplossing goed te laten communiceren. Jij als ontwikkelaar om het geprogrammeerd te krijgen en de gebruiker om het te laten werken. Deze infrastructuur is er nu, maar om er goed mee om te gaan vraagt nog enige aandacht. Alles is met alles verbonden, maar dat wil nog niet zeggen dat je overal onbeperkt van alles heen en weer kunt sturen. De verbinding is er, maar de weg is te smal en te bochtig om er met enorme trucks met lichtsnelheid overheen te kunnen denderen. Met name als je verkeer van en naar een mobieltje hebt, dan kan het lang duren en erg in de papieren gaan lopen. Flat rate UMTS is niet voor iedereen en ook niet overal beschikbaar.

Smart devices, de Visual Studio emulator en het internet

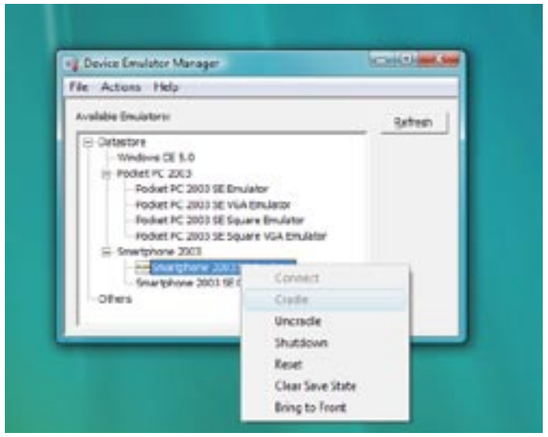
Om te beginnen ga ik een applicatie maken voor een Windows SmartPhone die gaat communiceren met een webservice. De gebruiker kan met zijn smartphone producten opvragen uit een NorthWind-database die ergens aan een webserver hangt. We kunnen de applicatie ontwikkelen zonder een fysieke smartphone in handen te hebben. Onderdeel van Visual Studio is de smart device-emulator. Het is een virtueel Windows Mobile-device waarop je applicaties volledig kunt debuggen. Het device kan vele gedaantes hebben, van de smartphone met een klein schermpje tot een joekel van een PDA met een roteerbaar VGA-scherm; zie afbeelding 1. Deze emulator heeft een volledige netwerkfunctionaliteit maar om die op gang te krijgen kost enige moeite. Op het web zijn wel stappenplannen te vinden, maar je zal niet de eerste zijn die midden in een presentatie ineens geen verbinding heeft. Ik spreek uit ervaring en doorloop het proces hier nog eens stap voor stap. In Visual Studio vind je in het menu Tools de optie Device Emulator Manager, kortweg DEM. Die toont een lijst van beschikbare emulatoren en van hieruit kun je ze opstarten; zie afbeelding 2.
De communicatie tussen de emulator en het netwerk loopt via ActiveSync; de applicatie die je ook gebruikt om gegevens uit te wisselen tussen een fysieke PDA en een Windows-pc. Meestal gebeurt dat met een USB-cradle waar je de PDA insteekt. Het maken van de verbinding en het synchroniseren gaan dan vanzelf. Als ActiveSync geïnstalleerd is, dan kun je in de DEM met de rechtermuisknop een draaiende emulator virtueel aan de cradle koppelen. Waarna ActiveSync even begint te rommelen. En daarna kun je vanuit de browser op de emulator zo het internet op; zie afbeelding 3.

Je kunt meer emulatoren tegelijkertijd draaien, maar er kan er maar één tegelijkertijd met de (virtuele) cradle verbonden zijn en er kan er ook maar één tegelijkertijd verbinding met het netwerk hebben. ActiveSync draait niet onder Vista. De functionaliteit is

Afbeelding 1. Device-emulatoren in actie



Afbeelding 2. De Device Emulator Manager



Afbeelding 3. Gebruik de DEM om connectie te maken met het (inter-)netwerk

overgenomen door het Windows Mobile Device Centre. Dat maakt intern gebruik van dezelfde API als de ActiveSync-applicatie voor XP. In de DEM kun je nog steeds cradleën, maar de emulator krijgt geen verbinding. Dit werkt (nog) niet. Een oplossing kan zijn om een virtuele XP-pc te starten. Virtual PC 2007 werkt heerlijk en ook Visual Studio 2005 draait er soepel in. Maar het grote probleem is dat de hele smart device-emulator zelf het niet meer doet in Virtual PC. Het levert wel een erg grappige foutmelding op. Het slot van het liedje is dat je voor het ontwikkelen van netwerkfunctionaliteit in een Smart Device-applicatie een fysieke XP-machine nodig hebt. Het internet is nu binnen bereik van de emulator. Op de emulator gaan we straks een applicatie bouwen die een webservice op de host-pc gaat benaderen. Deze webservice is benaderbaar via een url, normaal gesproken verwacht je iets in de geest van http://localHost/GekkoService/MyWebService.asmx. Vanuit de emulator gezien werkt dit niet. De emulator is een zelfstandige node in het netwerk met een eigen IP-adres. Localhost wordt door Windows vertaald naar het IP-adres 127.0.0.1, de interne loopback. De emulator kent geen localhost en de webservice is ook niet te vinden op de emulator zelf. De url van de webservice moet naar het concrete IP-adres van de pc zelf gaan verwijzen. Dit zou in principe kunnen via de naam van de machine, de url van de webservice wordt dan iets in de geest van http://Sturisoma/GekkoService/MyWebService.asmx. Behalve dat de nameserver van jouw locale netwerk hier goed voor op orde moet zijn, zit hier nog een addertje onder het gras. Een mobile device, dus ook de emulator, onderscheidt twee netwerken: intern en extern. Extern is in principe het internet en intern het lokale netwerk. De scheiding tussen intern en extern is recht-toe-recht-aan. Een domeinnaam met een punt, zoals *gekko-software.nl* is extern en een locatie zonder punt, zoals *sturisoma* is intern. Deze twee netwerken worden elk op een eigen manier benaderd. In het eindeloze woud van netwerkinstellingen, om precies te zijn in *settings|connections|advanced|select networks* stel je in welke connectie wordt gebruikt voor welk netwerk. Default wordt de emulator-cradle-connectie aan het externe netwerk gekoppeld. Om de pc met webservice op naam te vinden, moet je het interne netwerk ook koppelen aan dezelfde connectie. Een alternatief is om de scheiding tussen intern en iets subtieler te maken. Elders in het woud van instellingen kun je per url uitzonderingen op de regel opgeven.
Een alternatief is de afhankelijkheid van de dns over te slaan en direct het IP-adres van de pc in de url op te geven, iets in de geest van http://192.168.1.90/GekkoService/MyWebService.asmx. Het mag in ieder geval duidelijk zijn dat de url configureerbaar moet zijn. Om redenen van testbaarheid kan de locatie van de service op het web veranderen. Bij het consumeren van de webservice kom ik hier nog even op terug. Nu weet de emulator de pc wel te vinden, maar voor de pc is het een externe client die binnenkomt. Als jouw pc een firewall heeft - en welke heeft dat niet - dan moeten de requests naar de webservice wel worden doorgelaten. De betere firewalls doen dat in hun standaardconfiguratie niet. Op de firewall moet je poort 80 (standaard http-poort) openzetten. Als je het zo veilig mogelijk wilt doen, doe je dat alleen voor het lokale netwerk.

De webservice

Het maken van een webservice met Visual Studio is niets nieuws. Op het eerste gezicht een kwestie van een paar muisklikken en je hebt het skelet van de webservice staan. En als je ermee aan de slag gaat, lijkt het allemaal flitsend te gaan. In de inleiding van dit artikel noemde ik echter een aantal zaken dat in het echte gebruik, op het World Wide Web, enorm kan opspelen. In de eerste plaats de latentie. Op je pc heb je binnen een klik een antwoord van een webservice. Zoals je ook bij gewoon websurfen al merkt, kan het in het echt wel even duren voor je antwoord krijgt. Jouw applicatie werkt een stuk prettiger als die zo weinig mogelijk roundtrips naar webservices nodig heeft. Het is stukken beter om alles in één keer over te halen dan voor elk wissewasje weer terug het web op te moeten. In de praktijk ga je in één keer wat grotere (XML) gestructureerde boodschappen heen en weer sturen. De manier om in .NET grotere XML-structuren te beschrijven en gebruiken zijn XML-datasets. Maar hier komt meteen het tweede struikelblok op het echte web om de hoek kijken. Zo'n dataset gaat in tekstformaat over het net en in de praktijk kan een dataset behoorlijk groot worden. Op je lokale pc merk je daar niets van, maar als het over iets relatief traags als een GPRS-verbinding moet, dan krijg je er last van. Om over het kostenaspect, op menige verbinding betaal je per hoeveelheid dataverkeer, maar te zwijgen. Straks bespreek ik hoe je de hoeveelheid verstuurde data flink kunt beperken zonder aan functionaliteit te verliezen. De webservice zelf is niet meer dan een transportlaag. De echte functionaliteit hoort in een aparte class-library. Van daar uit is die te gebruiken door een willekeurig stuk code, waaronder de webservice die de functionaliteit over http beschikbaar stelt.

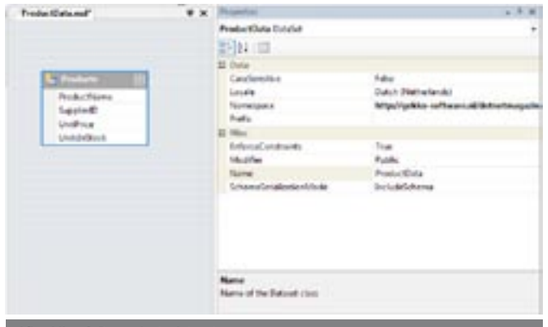
Opzet van de applicatie

Nu we in grote lijnen een overzicht hebben van de applicatie, kan de solution in Visual Studio gebouwd worden. Deze bestaat uit de volgende projecten:

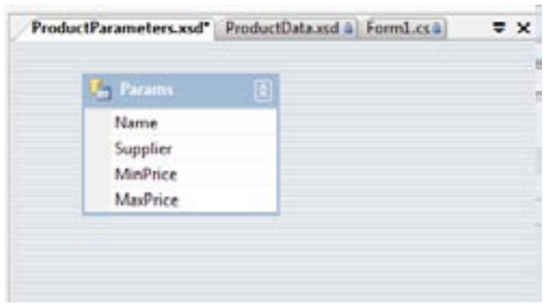
- Smart device-applicatie.
- Datalaag. Een class-library die de data definieert als XML-datasets. Intern leest hij de data uit een SQL-database.
- Webservice. Doorgeefluik tussen PDAapplicatie en datalaag.

De datalaag

De functionaliteit van deze datalaag is hier beperkt tot het noodzakelijke om de principes te demonstreren. Hij definieert een dataset voor producten uit de NorthWind-database en biedt de mogelijkheid een instantie van zo'n dataset volgens een aantal zoekcriteria te vullen. De essentie van een dataset is de definitie van tabellen en kolommen. In Visual Studio 2005 is de code om database-data te lezen en schrijven in de dataset-designer binnengeslopen; het



Afbeelding 4. De products-dataset



Afbeelding 5. De parameters zijn ook een dataset

zijn table-adapters. Maar deze laatste kunnen worden gemist. In dit verhaal ga ik het zonder doen, omdat ik de SQL zelf dynamisch vanuit code op wil bouwen. Toch kan de dataset-designer me goede diensten bewijzen. Na het gebruikelijke sleur-en-pleur-werk verwijder ik naast alle kolommen die ik niet hoef te zien ook alle data-adapters van de tabel. Van de *products*-tabel blijft een kaal, maar zeer bruikbaar dataschema over; zie afbeelding 4.

De database wordt benaderd vanuit een factory-class die 'on the fly' een data-adapter aanmaakt. De connectiestring naar de database wordt in de constructor meegegeven; zie codevoorbeeld 1.
Het minimale dat de data-adapter nodig heeft, is een SQL-select-query en een connectie. Het grote voordeel van een data-adapter is dat die de resultaatkolommen van de query mapt op de kolommen van de dataset. Mocht de kolom van jouw dataset een andere naam hebben dan de bijbehorende kolom in de database, dan is het sql AS-keyword afdoende om de mapping goed uit te voeren. Het grote voordeel van het 'on the fly' aanmaken van de adapter is, dat je de SQL dynamisch aanmaakt in code aan de hand van de gewenste selectie. De gebruiker wil producten kunnen selecteren aan de hand van (een combinatie) van verschillende parameters. Deze selectie wordt naar de webservice gestuurd; om daar door een web-method te worden verwerkt. Nou is het een verkeerd beeld om deze web-method als een procedure te zien die op afstand wordt uitgevoerd. Een beter werkende metafoor is het heen en weer sturen van berichten. Je stuurt de web-method een bericht en je krijgt een bericht terug. Deze berichten zijn gecodeerd in XML. Het bericht dat je naar de service stuurt, zijn de selectiecriteria, het bericht dat je terugkrijgt is een XML-dataset. Het bericht dat je verstuurt, is een verzameling van parameters. Een goede manier om die verzameling te omschrijven, is weer een dataset. In de dataset-designer ontwerp ik zo het parameterbericht; zie afbeelding 5.
Deze XML-parameter-message wordt nu verwerkt door de *ProductSelection*-web-methode. Deze bekijkt voor welke selectiecriteria

```
public class ProductFactory
{
  private SqlConnection con;
  private const string selectSQL = "SELECT ProductName, SupplierID,
       UnitPrice, UnitsInStock FROM Products";

  public ProductFactory(string connectionString)
  {
    con = new SqlConnection(connectionString);
  }

  public ProductData ProductSelection()
  {
    string theSQl = selectSQL;
    SqlDataAdapter da = new SqlDataAdapter(theSQl, con);
    ProductData ds = new ProductData();
    try
    {
      con.Open();
      da.Fill(ds.Products);
    }
    finally
    {
      con.Close();
    }
    return ds;
  }
}
```

Codevoorbeeld 1.

```
public ProductData ProductSelection(ProductParameters parms)
{
  Dictionary<string, object> sqlParams = new Dictionary<string, object>();

  StringBuilder parmBuild = new StringBuilder();
  if (parms.Params.Count > 0)
  {
    ProductParameters.ParamsRow selection = parms.Params[0];
    if (!selection.IsNameNull())
    {
      parmBuild.Append("AND (ProductName LIKE @Name)");
      sqlParams.Add("@Name", selection.Name);
    }
    if (!selection.IsSupplierNull())
    {
      parmBuild.Append("AND (SupplierID = @Supplier)");
      sqlParams.Add("@Supplier", selection.Supplier);
    }
    if (!selection.IsMinPriceNull())
    {
      parmBuild.Append("AND (UnitPrice >= @MinPrice)");
      sqlParams.Add("@MinPrice", selection.MinPrice);
    }
    if (!selection.IsMaxPriceNull())
    {
      parmBuild.Append("AND (UnitPrice <= @MaxPrice)");
      sqlParams.Add("@MaxPrice", selection.MaxPrice);
    }
  }

  StringBuilder sqlBuild = new StringBuilder(selectSQL);
  if (parmBuild.Length > 0)
  {
    sqlBuild.Append(" WHERE ");
    sqlBuild.Append(parmBuild.ToString().Substring(3));
  }

  string theSQl = sqlBuild.ToString();

  SqlDataAdapter da = new SqlDataAdapter(theSQl, con);
  foreach (KeyValuePair<string, object> parm in sqlParams)
    da.SelectCommand.Parameters.AddWithValue(parm.Key, parm.Value);

  ProductData ds = new ProductData();
  try
  {
    con.Open();
    da.Fill(ds.Products);
  }
  finally
  {
    con.Close();
  }
  return ds;
}
```

Codevoorbeeld 2.

```
[WebMethod]
public DataLibrary.ProductData ProductList(DataLibrary.ProductParameters parms)
{
   return data().ProductSelection(parms);
}
```
Codevoorbeeld 3.

```
[WebMethod]
public DataLibrary.ProductData ProductListWithoutSchema(DataLibrary.
ProductParameters parms)
{
   DataLibrary.ProductData ds = data().ProductSelection(parms);
   ds.SchemaSerializationMode = SchemaSerializationMode.ExcludeSchema;
   return ds;
}
```
Codevoorbeeld 4.

```
[WebMethod]
public DataSet ProductListUntyped(DataSet parms)
{
   DataLibrary.ProductData ds = data().ProductSelectionUntyped(parms);
   ds.SchemaSerializationMode = SchemaSerializationMode.ExcludeSchema;
   return ds;
}
```
Codevoorbeeld 5.

```
[WebMethod]
public string ProductListPox(DataLibrary.ProductParameters parms)
{
   DataLibrary.ProductData ds = data().ProductSelection(parms);
   return ds.GetXml();
}
```
Codevoorbeeld 6.

waarden zijn gezet en bouwt aan de hand daarvan dynamisch een geparametriseerd SQL-statement op. In de code wordt de WHERE-clause opgebouwd in een stringbuilder. Het resultaat daarvan wordt gecombineerd met de basissql-string. De eerste AND moet dan wel van de clause af worden geknipt; dat doet de substring-methode. De SQL-parameters worden verzameld in een dictionary-object en daarna aan de parameters-property van het select-command toegevoegd.; zie codevoorbeeld 2.
In eerste instantie ben je misschien geneigd de actuele parameterwaarden hard in het SQL-statement op te nemen, maar dat is iets wat je nooit en te nimmer moet doen. Het is de manier om onder een sql-injection attack te sneuvelen. Stel nou dat als gezochte naam de waarde *e' drop table products* -- wordt aangeboden. Als de databaserechten het toelaten, dan ben je de tabel echt kwijt. Succes verzekerd. Als je parameters gebruikt, levert dit selectiecriterium in NorthWind hooguit een overzicht op van leverbare table-wines. Als tweede pluspunt geldt dat SQL Server erg goed is in het cachen van geparametriseerde queries.

De data publiceren in de webservice

Nu hebben we een flexibele datalaag geschreven, maar dat wil nog niet zeggen dat deze meteen ook geschikt is om gepubliceerd te worden in de webservice. Stel dat je het resultaat strong typed publiceert in een web-method; zie codevoorbeeld 3.
Deze webservice wordt moeiteloos geconsumeerd door een .NET-webservice-consumer. Maar in de praktijk mankeert er het een en ander aan. In de eerste plaats is de hoeveelheid data die over de lijn gaat aan de grote kant, zeker voor mobiele applicaties. Als een dataset wordt geserialiseerd, dan wordt ook het schema van het dataset geserialiseerd, elke keer weer; zie afbeelding 6.
Voor de ontwikkeling van de consumerende applicatie is het gemakkelijk om het schema te hebben, maar in het gebruik kan het gemist worden. Gelukkig kun je in .NET, met ingang van versie 2, het serialiseren van het schema onderdrukken; zie codevoorbeeld 4. Dit scheelt al aanzienlijk in de hoeveelheid data die over het netwerk moet. De webservice publiceert gegevens als een getypeerde dataset. Een getypeerde dataset is rijk aan informatie, zo rijk zelfs dat maar weinig consumenten van webservices hier weg mee weten. In ons geval willen we de webservice gaan consumeren met .NET Compact Framework-applicaties. Versie 2 van het compact framework begrijpt getypeerde datasets, versie 1 nog niet. Dat framework kent alleen het ongetypeerde dataset, de basisklasse. Visual Studio kent in grote lijnen drie smaken Smart Device-applicaties: Pocket PC 2003, Smartphone 2003 en Windows CE. Versie 2 van het CF is alleen te gebruiken in een Pocket PC-applicatie. Wil de webservice goed geconsumeerd kunnen worden door een smartphone, zoals we straks zullen doen, dan zal die de data als ongetypeerde dataset moeten aanbieden; zie codevoorbeeld 5.
Merk op dat je een aangepaste selectiemethode nodig hebt en dat je ook nog steeds moet aangeven dat het schema niet mee moet worden geserialiseerd. Deze versie werk ik hier verder niet uit.

Afbeelding 6. Het schema komt mee in het resultaat

POX

De webservice levert gegevens in een rijke XML-datasetstructuur. Maar het bericht blijft aan de grote kant, met name door de vele attributen waarmee de gegevens zijn versierd. De consument van de service doet run-time niets met al die informatie en tijdens het ontwikkelen zal menige tool de informatie van de webservice niet begrijpen. Elke consument van webservices weet wat een string is, maar met een dataset weet menigeen zich geen raad. Een oplossing bied Plain Old Xml, kortweg POX. De .NET-datasetklasse heeft de methode *GetXML*. Deze levert een heel mooie eenvoudige XML-representatie van de inhoud op. Deze XML is een gewone string en kan direct worden gepubliceerd; zie codevoorbeeld 6.
Het resultaatbericht van de webservice is nu een heel stuk handzamer; zie codevoorbeeld 7.
De string bevat een minimale maar duidelijke XML-representatie van de data. Daarnaast zit er één belangrijk stuk metadata in: de xmlnamespace van het bijbehorende schema. Met of zonder schema, elke consument van webservice die iets met XML aankan, kan hier mee werken. Ook zonder schema kan met XML DOM (denk bijvoorbeeld aan *tXMLdocument* in de oudere Delphi-versies) alle data gestructureerd worden gelezen. De parameter van de web-method is ook een getypeerd XML-document, deze zal dezelfde behandeling moeten ondergaan om alle complexe types uit de webservice-interface te verwijderen. In de volgende code is ook te zien hoe je vanuit een string in een paar stappen weer een volledig getypeerde dataset kunt maken; zie codevoorbeeld 8.
De binnenkomende string wordt gelezen door een *StringReader*. Een *XmlTextReader* kan de XML in de string parsen en de methode *ReadXml* van de *DataSet* klasse vult het dataset. De data willen we (intern!) in een getypeerde dataset hebben; een nieuw getypeerde *ProductParams*-dataset wordt aangemaakt en het resultaat van de eerste tabel van de binnenkomende data wordt met een merge in de *Params*-tabel gelezen. In eerste instantie zou je de binnenkomende tekst rechtstreeks willen inlezen in de tabel van de getypeerde dataset. De methode *ReadXml* leest platte tekst, maar kijkt wel degelijk naar de inhoud. Ze zal de parameterdata alleen maar inlezen als die in een tabel met de naam *Params* staat. De tussenstap van een ongetypeerde dataset maakt de webservice flexibeler, de enige verwachting die de service heeft, is dat de eerste tabel in de dataset parameters bevat. Merk op dat de interface van de service onafhankelijk is van de inhoud van de lijst van de aangeboden parameters. Wat versiebeheer van de webservice aanzienlijk eenvoudiger zal maken. Als de datalaag verandert, heeft dat geen consequenties voor de communicatielaag. Dat is goed, want dat zijn ook twee verschillende onderdelen van jouw applicatie. Met deze implementatie van de webservice gaan we verder werken. Als je de eerdere datasetversies ook aan wilt bieden aan potentiële gebruikers van je service, dan kun je dat beter doen in een afzonderlijke webservice. Anders komen alle metadata rond de datasets toch in de WSDL (de beschrijving van de webservice die door je ontwikkeltools wordt gebruikt) en moet een consument, die alleen de POX-variant begrijpt, iets met die metadata gaan doen. Foutmeldingen genereren bijvoorbeeld. Wie wel eens in CF 1 of in Delphi 6 geprobeerd heeft een webservice te consumeren die een getypeerde dataset teruggeeft, herkent de frustratie. Het is voor de extra service niet nodig een nieuw project toe te voegen; één webproject kan meer webservices bevatten.

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://gekko-software.nl/dotnetmagazine/">
   <ProductData xmlns="http://gekko-software.nl/dotnetmagazine/
                ProductData.xsd">
      <Products>
         <ProductName>Chai</ProductName>
         <SupplierID>1</SupplierID>
         <UnitPrice>18.0000</UnitPrice>
         <UnitsInStock>11</UnitsInStock>
      </Products>
      <Products>
         <ProductName>Chang</ProductName>
         <SupplierID>1</SupplierID>
         <UnitPrice>19.0000</UnitPrice>
         <UnitsInStock>27</UnitsInStock>
      </Products>
   </ProductData>
</string>
```
Codevoorbeeld 7.

```
[WebMethod]
public string ProductListPox(string parameters)
{
   StringReader sr = new StringReader(parameters);
   XmlTextReader xr = new XmlTextReader(sr);
   DataSet pds = new DataSet();
   pds.ReadXml(xr);
   DataLibrary.ProductParameters parms =
new DataLibrary.ProductParame ters();
   parms.Params.Merge(pds.Tables[0]);
   DataLibrary.ProductData ds = data().ProductSelection(parms);
   return ds.GetXml();
}
```
Codevoorbeeld 8.

Smart client

De webservice staat klaar, we gaan deze consumeren met een smartphone-applicatie. En komen zo terecht in het CF-versie 1.

Afbeelding 7. De applicatie krijgt een gezicht

De applicatie is simpel: de gebruiker kan selectiecriteria invullen en krijgt een lijst terug van producten die daar aan voldoen; zie afbeelding 7.
Het project krijgt een webreference naar de webservice. De methode van de webservice verwacht de parameters in een XML-string. Om de code overzichtelijker en onderhoudbaar te maken, maak ik een hel-

```
internal class PwsParameters
{
  internal static string parameterMessage(string name, string supplier,
 string minPrice, string maxPrice)
  {
    DataSet ds = new DataSet();
    DataTable dt = new DataTable("Params");
    dt.Columns.Add("Name", typeof(String));
    dt.Columns.Add("Supplier", typeof(String));
    dt.Columns.Add("MinPrice", typeof(String));
    dt.Columns.Add("MaxPrice", typeof(String));
    ds.Tables.Add(dt);

    DataRow dr = dt.NewRow();
    if (name != string.Empty)
      dr["Name"] = name;
    if (supplier != string.Empty)
      dr["Supplier"] = supplier;
    try
    {
      if (decimal.Parse(minPrice) > 0)
        dr["MinPrice"] = minPrice;
    }
    Catch {}
    try
    {
      if (decimal.Parse(maxPrice) > 0)
        dr["MaxPrice"] = maxPrice;
    }
    Catch {}
    dt.Rows.Add(dr);

    return ds.GetXml();
  }
}
```
Codevoorbeeld 9.

```
internal class PwsProduct
{
   internal PwsProduct(DataRow dr)
   {
      Naam = dr[“ProductName”].ToString();
      Leverancier = dr[“SupplierID”].ToString();
      Prijs = decimal.Parse(dr[“UnitPrice”].ToString());
      Voorraad = int.Parse(dr[“UnitsInStock”].ToString());
   }
   internal string Naam;
   internal string Leverancier;
   internal decimal Prijs;
   internal int Voorraad;
}
```

Codevoorbeeld 10.

per voor de parameters. Deze maakt een parameterdataset, voegt een rij toe en vult deze met de parameterwaarden; waarna de parameterdataset naar string wordt geserialiseerd; zie codevoorbeeld 9.
Als extra functionaliteit valideert de methode ook nog de prijsparameters. Een tweede helper-class bouwt een set geretourneerde producten op aan de hand van de XML-string. Elk product wordt gevuld vanuit een *DataRow*-object. De constructor van de *PwsProduct*-class krijgt een ongetypeerde *DataRow* binnen en initialiseert aan de hand daarvan het product; zie codevoorbeeld 10.
De lijst van producten wordt ingepakt in de *PwsProducten*-class. Deze heeft een array van sterk getypeerde producten. De constructor van de *PwsProducten*-class krijgt de XML mee. Na de initialisatie hebben we een object met daarin getypeerde producten en daarmee de nadelen van ongetypeerde data achter ons gelaten; zie codevoorbeeld 11.
De methode *ReadXml* vult in één keer de dataset met structuur en inhoud. De code gaat er van uit dat de XML één tabel bevat die de kolommen bevat die in de *PwsProduct*-constructor verwacht worden. Dit is natuurlijk een kwetsbaar punt in de code. Wat een reden te meer is om al deze late-bound afhankelijkheden hier te concentreren. Een degelijk stel unit-tests zal moeten bewaken dat de code ook in latere versies van de applicatie goed blijft werken. De applicatie werkt met die webservice die het minst beslag legt op beschikbare bandbreedte. De late-bound code is ingekapseld. De invocatie van de webservice is recht-toe-recht-aan. Een proxy wordt aangemaakt waarna de url van de webservice wordt gezet. Het Compact Framework ondersteunt helaas geen configuratiebestanden. Voor CF 2 biedt opencfnet.org een config manager, maar voor CF 1 moet je zelf iets maken. Voor dit verhaal zou het een zijsprong zijn, ik laat dit over aan je eigen fantasie en codeer hier keihard een url; zie codevoorbeeld 12.
Gebruikmakend van de helper-classes wordt het parameterbericht in elkaar gezet en het resultaat van de invocatie wordt vertaald naar een productlijst.

```
internal class PwsProducten
{
   internal PwsProducten(string rawXML)
   {
   StringReader sr = new StringReader(rawXML);
   XmlTextReader xr = new XmlTextReader(sr);
   DataSet ds = new DataSet();
   ds.ReadXml(xr);
   DataTable dt = ds.Tables[0];
   ProduktLijst = new PwsProduct[dt.Rows.Count];
   for (int i = 0; i < dt.Rows.Count; i++)
      ProduktLijst[i] = new PwsProduct(dt.Rows[i]);
   }

   internal PwsProduct[] ProduktLijst;

}
```

Codevoorbeeld 11.

```
ProductsWebService.ProductService ws = new ProductsWebService.Product
Service();
// Te doen : configuratie in bestand
ws.Url = “http://192.168.1.110:50555/ProductService.asmx”;
string parmString = PwsParameters.parameterMessage(textBoxProdukt.Text,
 textBoxLeverancier.Text, textBoxMinPrijs.Text, textBoxMaxPrijs.Text);
string wsResult = ws.ProductListPox(parmString);
PwsProducten prods = new PwsProducten(wsResult);

foreach (PwsProduct prod in prods.ProduktLijst)
{
  ListViewItem li = new ListViewItem(new string[] { prod.Naam,
prod. Leverancier, prod.Prijs.ToString(), prod.Voorraad.ToString() });
  listView1.Items.Add(li);
}
```

Codevoorbeeld 12.

Asynchrone invocatie

De mobiele applicatie en de webservice houden nu wel rekening met de beperktheid van de bandbreedte op het echte net. Maar ze doen nog niets met de latentie, het andere nadeel. Als de gebruiker op de knop van z’n telefoon heeft geklikt, kan het een hele tijd duren voordat er enig resultaat is te zien. Bij de eerste invocatie moet de applicatie de proxy genereren en starten. Dat duurt even. Daarna moet de applicatie nog wachten op de response van de webserver. Ook dat kan een tijdje duren. Nu kun je al die tijd een zandloper laten lopen, maar daar wordt de gebruiker ook niet vrolijker van. Menigeen gaat nog eens op allerlei knoppen duwen, maar de applicatie reageert niet totdat de webservice antwoord heeft. Gelukkig kun je een webservice ook asynchroon aanroepen. Zodra het request aan de proxy is doorgegeven, reageert de applicatie weer en kan wat anders gaan doen totdat er antwoord is. De gegenereerde proxy biedt voor elke web-method ook een variant om hem asynchroon te invoken. Als extra parameter heeft die een delegate nodig van het type *AsyncCallback*. De constructor van dit delegate-object krijgt een methode mee, dit is de methode die wordt aangeroepen als de webservice geantwoord heeft. De invocatie van de webservice wordt dan zoals in codevoorbeeld 13 is te zien.

Als eerste wordt het menu-item tijdelijk uitgeschakeld om te voorkomen dat de gebruiker de actie nog een keer op gaat starten. De invocatie wordt gestart door de methode *BeginProductListPox* waarna de code meteen doordendert. Als de webservice terugkomt, dan wordt de methode *wscb* aangeroepen. Voor je die methode *wscb* gaat invullen, moet je je realiseren dat die op een thread in de achtergrond draait en niet op de thread van de UI. Dat betekent dat de methode ook niet aan de UI mag komen, het resultaat van de webservice kan die dus niet zomaar laten zien. Elke control, dus ook een form, kent de methode *Invoke*. Daarmee kun je vanuit een background-thread een methode op de UI-thread uitvoeren. Het probleem van de *Invoke*-methode is dat er maar één overload van is en die heeft geen enkele bruikbare parameter. Dat betekent dat je het resultaat van de webservice ergens op moet slaan en dat methodes op beide threads deze gegevens gaan delen. Hetgeen resulteert in de volgende code zoals in codevoorbeeld 14 is te zien.

```
private void menuItem1_Click(object sender, EventArgs e)
{
   menuItem1.Enabled = false;
   ProductsWebService.ProductService ws =
      new ProductsWebService.ProductService();
   // Te doen : configuratie in bestand
   ws.Url = “http://192.168.1.53:50555/ProductService.asmx”;

   string parmString = PwsParameters.parameterMessage(textBoxProdukt.
Text, textBoxLeverancier.Text, textBoxMinPrijs.Text, textBoxMaxPrijs. Text);
   ws.BeginProductListPox(parmString, new AsyncCallback(wscb), ws);
}
```

Codevoorbeeld 13.

```
private PwsProducten prods = new PwsProducten();

private void wscb(IAsyncResult asResult)
{
   ProductsWebService.ProductService ws = asResult.AsyncState as
 ProductsWebService.ProductService;
   string wsResult = ws.EndProductListPox(asResult);
   lock (prods)
   {
      prods = new PwsProducten(wsResult);
   }
   Invoke(new EventHandler(toonResultaat));
}

private void toonResultaat(object sender, EventArgs e)
{
   foreach (PwsProduct prod in prods.ProduktLijst)
   {
      ListViewItem li = new ListViewItem(new string[] { prod.Naam,
 prod.Leverancier, prod.Prijs.ToString(), prod.Voorraad.ToString() });
          listView1.Items.Add(li);
   }
   menuItem1.Enabled = true;
}
```

Codevoorbeeld 14.

De productenlijst is een member van het form geworden, zodat zowel de *wscb*- als de *toonResultaat*-methode hem kan gebruiken. De *wscb* callback-methode krijgt in zijn parameter de proxy binnen. Daarop wordt de methode *EndProductListPox* uitgevoerd om het resultaat van de webservice te pakken te krijgen. Voordat dit gebruikt kan worden om het *prods*-object te vullen, moet het *prods*-object gelocked worden om threading errors te voorkomen. Via de *Invoke* komen we in de methode *toonResultaat* weer terecht op de thread van de UI en kan het resultaat veilig worden getoond. Deze code is hier en daar een beetje cryptisch, maar het resultaat is dat de applicatie nu goed rekening houdt met de beperking van het World Wide Web en met onbedoelde multi-threading.

Compact Framework 2

In versie 2 van het Compact Framework kun je veel compactere en nettere code schrijven. Het kent getypeerde datasets, zodat de omweg van de productenlijst en parameter-helper-klasses niet meer nodig is. Daarnaast heeft de Invoke-methode een overload met een array van parameters, zodat het ook niet meer nodig is om data te delen via nieuwe members. Maar de verdere principes blijven gelijk, ook hier moet je rekening blijven houden met de UI versus de background-thread.

Samenvattend

- Verstuur over het web alleen dat wat echt nodig is, niet meer
- Verstuur alleen wanneer dat echt nodig is, niet vaker
- Houd er rekening mee dat het even kan duren voor je antwoord krijgt
- Het web is meer dan ASP.NET, bied (desgewenst) simpele interfaces.

Ik hoop duidelijk te hebben gemaakt hoe je dit alles met .NET op een goede manier voor elkaar kunt krijgen, waarbij we intern de volle rijkdom van het framework hebben gebruikt en naar buiten de prachtige infrastructuur van het web zo optimaal mogelijk hebben benut.

Peter van Ooijen is werkzaam als zelfstandig .NET-consultant en ontwikkelaar bij Gekko Software (www.Gekko-Software.nl). Naast de dagelijkse ASP.NET-kost zijn de tablet pc en mobile devices geliefde troeteldieren. Ook op zijn weblog http://codebetter.com/blogs/peter.van.ooijen komen deze geregeld voorbij. Voor vragen en opmerkingen is hij te bereiken via Peter.van.Ooijen@Gekko-Software.nl