WPF Data Binding

Verder kijken dan mooie plaatjes

Joris Bos

Met behulp van Windows Presentation Foundation kunnen we indrukwekkende user interfaces bouwen. In plaats van het weggestopte, automatische gegenereerde onderdeel in Windows Forms, is het zelf opbouwen van de user interface in WPF de standaard manier van zakendoen. Maar door de focus op UI en grafische opties worden helaas andere krachtige mogelijkheden van WPF over het hoofd gezien. Eén daarvan is Data Binding, het synchroon houden van twee punten van data.

Aan de hand van twee voorbeelden bespreken we de theorie achter Data Binding. Hierbij gebruiken we de kracht van WPF en XAML om deze Data Binding in onze voorbeeldapplicatie te integreren. Data Binding is overigens niet alleen toepasbaar op WPF, het overgrote deel van de besproken functionaliteit geldt ook voor Silverlight 2.0. Met Data Binding kunnen we twee punten van data, een doel en een bron, gelijk aan elkaar houden. De bron mag hierbij elk .NET-object of een property van dit object zijn. Het doelobject is altijd een property van een Dependency Object. Het gaat om een 'afhankelijk' .NET-object waar de waarde van de properties afhangt van de uitkomst van een evaluatie van tien stappen. Een Dependecy Object geeft notificaties zodra deze properties veranderen en is WPF-specifiek (meer over Dependency Objects vind je op Microsofts MSDN, http://msdn.microsoft.com/en-us/library/ms752914.aspx).

Optioneel bestaat er keuze uit validatie bij het terugschrijven naar de bron en/of voor een eigen implementatie van conversie van data tussen doel en bron. Conversie en validatie vallen buiten de scope van dit artikel. Ter illustratie beginnen we met een simpel voorbeeld. We nemen twee tekstvakken en noemen ze myTargetTextBox en mySourceTextBox. We gaan de binding zo instellen dat wanneer je in een van beide tekstboxen een wijziging aanbrengt, deze direct in het andere tekstvak zichtbaar is.

Om dit te bereiken definiëren we allereerst een Bindingobject en de bron van onze binding. In dit voorbeeld nemen we de 'Text' property van een TextBox-object. Interessant is dat ook het binden aan minder voor de hand liggende, of zelfs een combinatie van properties mogelijk is. Zo zouden we de bron kunnen binden aan de Text property van een TextBox en het doel aan de BackGround property. Met behulp van de default-conversie maken we de achtergrondkleur van een control afhankelijk van de ingevoerde tekst!

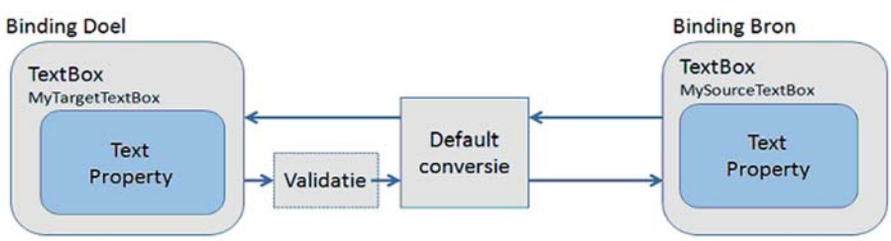
```
Binding myBinding = new Binding();
myBinding.Source = mySourceTextBox;
// òf myBinding.ElementName = "mySource-
TextBox";
myBinding.Path = new PropertyPath(TextBox.
TextProperty);
```

Belangrijk is dat je voor PropertyPath de statische dependency property TextBox.TextProperty nodig hebt, niet de Text property van het daadwerkelijke sourceobject. Vervolgens definiëren we het doel door een property, in dit geval de tekst van andere tekstbox myTargetTextBox, te koppelen aan onze bron. Dit gebeurt tevens met de dependency property TextBox.TextProperty.

```
myTargetTextBox.SetBinding(TextBox.Text-
Property, myBinding);
```

Na het compileren van deze code zien we bij het typen in mySourceTextBox direct de ingevoerde tekst in myTargetTextBox. Dit komt omdat alle bronveranderingen direct worden doorgevoerd in het doelobject. Omgekeerd verandert mySourceTextBox alleen wanneer myTargetTextBox de focus verliest.

Dit heeft te maken met een aantal standaardconfiguraties in het Binding-object. Om een binding volledig te kunnen configureren zijn twee extra instellingen van belang. Allereerst de Binding.Mode. Deze geeft aan op wat voor manier de beide objecten binnen de binding met elkaar communiceren. Daarnaast is er de Binding UpdateSourceTrigger, die eigenlijk de vraag stelt: wanneer moet ik wijzigingen in het doel doorgeven naar de bron?



ILLUSTRATIE 1. SCHEMATISCH OVERZICHT VAN TWOWAY DATABINDING

Binding.Mode

Met deze eigenschap van een binding stel je de wijze van binding in. Er zijn vier modes beschikbaar:

- *OneTime*
  Snapshot-binding. Het doelobject wordt eenmalig gebonden aan de bron. Verdere wijzigingen in de bron worden niet weergegeven in het doelobject. Minst resource-intensief.
- *OneWay*
  Read-only binding. Alle wijzigingen in de bron zijn zichtbaar weergegeven in het doelobject.
- *OneWayToSource*
  Enkel wijzigingen in het doelobject worden naar de bron doorgegeven. Handig voor het doorgeven van wijzigingen naar niet-dependency objects.
- *TwoWay*
  Zowel doelobject als bron wordt synchroon gehouden, dit type binding is het meest resource-intensief.

Binding.UpdateSourceTrigger

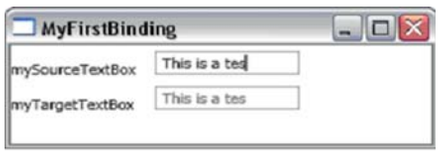
Met deze eigenschap van de binding stel je de manier in waarop wijzigingen van het doel worden doorgevoerd naar de bron.

- *Explicit*
  Updaten gebeurt alleen na het aanroepen van de UpdateSource()-methode op de bindingexpressie.
- *LostFocus*
  Updaten gebeurt wanneer het doelobject de focus verliest
- *PropertyChanged*
  Updaten gebeurt zodra het doelobject verandert.

Concluderend uit onze ervaringen kunnen we stellen dat de Binding.Mode en Binding.UpdateSourceTrigger properties van de binding hier respectievelijk TwoWay en LostFocus zijn. Om er nu voor te zorgen dat in beide gevallen de wijzigingen direct van het doelobject naar het bronobject worden doorgevoerd, dienen we aan te geven dat de UpdateSourceTrigger PropertyChanged moet zijn.

```
Binding myBinding = new Binding();
myBinding.Source = mySourceTextBox;
myBinding.Path = new PropertyPath(TextBox.
TextProperty);
MyBinding.UpdateSourceTrigger = Up-
dateSourceTrigger.PropertyChanged;
myTargetTextBox.SetBinding(TextBox.Text-
Property, myBinding);
```

Als we nu het programma opnieuw compileren en opstarten, zien we dat tekst



ILLUSTRATIE 2. TWOWAY BINDING

inderdaad geüpdatet wordt zodra we iets in myTargetTextBox invoeren.

XAML, leesbaarder en compacter

WPF introduceert de declaratieve opmaaktaal XAML (Extensible Application Markup Language). In XAML kunnen we de hierboven in code uitgeschreven binding in het XAML-document als volgt opstellen.

```
<TextBox Name="myTargetTextBox" Text
="{Binding ElementName= mySourceTextBox,
Path=Text, UpdateSourceTrigger=
PropertyChanged}"/>
```

Met behulp van de accoladetekens geven we in XAML aan dat we een andere syntax gaan gebruiken. In dit geval het opstellen van een Binding en haar properties. Kommagescheiden kunnen we tevens de UpdateSourceTrigger en Binding.

Mode veranderen, mochten we deze willen wijzigen van hun defaultwaarde. Het gebruik van XAML heeft hier duidelijk de voorkeur: leesbaarder en compacter. Let wel, als je als ontwikkelaar niet wilt dat een designer via XAML de bindingfunctionaliteit kan bereiken, is het definiëren van een binding in code altijd mogelijk.

OneWay binding met XML

We hebben nu gezien dat het configureren van een databinding eenvoudig is aan de hand van een simpel voorbeeld. Als bron namen we de Text property van een TextBox, een WPF-element. We kunnen echter als bron vele complexere objecten gebruiken. In het volgende voorbeeld nemen we een XML-bestand als bron. We gaan een simpele applicatie maken waarin we in een lijst een overzicht geven van beschikbare boeken en hun ISBN-waarde. Een verandering van selectie moet in een TextBox onder de lijst vervolgens een korte samenvatting van het verhaal geven. We willen het XML-bestand zelf niet updaten, dus kiezen we voor een OneWay binding.

Allereerst creëren we onderstaand XML-bestand data.xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<books>
  <book title="The Great Marsian War"
ISBN="123456789" category="War">
    <bookpreview>
    This book is about the great war that
raged on Mars in the early 20's.
    </bookpreview>
  </book>
  <book title="Froglike" ISBN="987654321"
category="Animals">
    <bookpreview>
      This book is about the little green
machine : Everything about frogs.
    </bookpreview>
  </book>
  <book title="It's raining stones"
ISBN="159487623" category="Drama">
    <bookpreview>
      This book is a love-story at the
time of a giant meteor-strike.
    </bookpreview>
  </book>
  <book title="Twinkelbelly"
ISBN="362951847" category="Drama">
    <bookpreview>
      This book tells the story of Twin-
kelbelly, an adventure through Australia.
    </bookpreview>
  </book>
  <book title="WPF for dummies"
ISBN="159486273" category="IT">
    <bookpreview>
      This book is about Windows Presenta-
tion Foundation, the next generation of
desktop software.
    </bookpreview>
  </book>
</books>
```

We hebben nu een verzameling boeken met elk een attribuut title, ISBN en category. En een element bookpreview waarin een korte samenvatting van het boek staat beschreven. Om deze data beschikbaar te stellen aan onze applicatie maken we in XAML een instantie van de klasse XmlObjectProvider en voegen deze toe aan de Resource-collectie van ons Window-object. WPF Resources zijn vanuit onderliggende objecten met hun key eenvoudig te benaderen.

```
<Window.Resources>
        <XmlDataProvider x:Key="myData"
Source="data.xml" XPath="books/*" />
</Window.Resources>
```

Door hier de XmlDataProvider in de resource te plaatsen komt een instantie van de klasse beschikbaar voor onderliggende elementen via de sleutel 'myData'. In code ziet dit er als volgt uit :

```
XmlDataProvider myData = new XmlDataPro-
vider();
myData.Source = new Uri("data.xml", Uri-
Kind.Relative);
    myData.XPath = "books/*";
    this.Resources.Add("myData", myData);
```

Door gebruik te maken van een XPath-expressie (een query-taal voor XML) kunnen we bepalen welke items we beschikbaar willen stellen: In dit geval alle boeken in de collectie books.

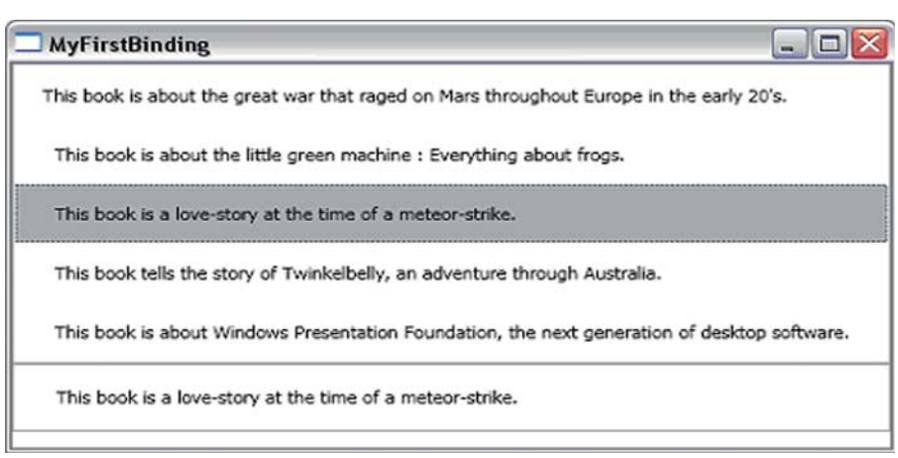
We introduceren in dit voorbeeld het gebruik van DataContext in WPF. Alle User Interface-elementen in WPF hebben een DataContext property. Deze DataContext doet precies waar het voor staat: zorgen voor een context waarin je data kunt plaatsen die voor het element en alle onderliggende elementen te benaderen is. Als groot voordeel van een DataContext geldt dat we voor een binding niet expliciet een bron hoeven aan te geven. We kunnen de binding opdracht geven gebruik te maken van de dichtstbijzijnde, bovenliggende DataContext door simpelweg de Source property van de binding weg te laten. Hierbij kunnen verschillende onderliggende elementen één DataContext gebruiken en manipuleren.

Nu we ons XmlDataProvider-object hebben toegevoegd aan de Resource-collectie van ons Window kunnen we deze toegankelijk maken voor objecten op een gewenst niveau. We kiezen in dit voorbeeld voor alle objecten onder het niveau van ons hoofd-layoutpanel, een StackPanel.

```
<StackPanel DataContext="{StaticResource
myData}">
…
</StackPanel>
```

De StaticResource staat voor een verwijzing naar een Resource die we eerder hebben toegevoegd aan de resource-collectie van een bovenliggend object. Static wijst erop dat we een eenmalige, vaste verwijzing maken naar het object. Wijzigingen in het object vinden nog wel plaats maar een verandering van objecttype zal niet worden opgemerkt.

Kortom, een verandering van de source van het XmlDataProvider-object met de sleutel 'myData' zal wel worden opgemerkt maar een verandering van het huidige objecttype XmlDataProvider naar een nieuw objecttype ObjectDataProvider niet. Mocht dit toch noodzakelijk zijn, dan dienen we gebruik te maken van een zwaardere variant: DynamicResource.



<BIJSCHRIFT>ILLUSTRATIE 3

Read-only binding. Alle wijzigingen in de bron zijn zichtbaar weergegeven in het doelobject.

(http://msdn.microsoft.com/en-us/library/ms748942.aspx)

Nu we de datacontext van het StackPanel hebben ingevuld, kunnen we gebruikmaken van een zogenaamde 'lege' binding. Deze gaat, zoals eerder vermeld, op zoek naar de dichtstbijzijnde, bovenliggende DataContext en gebruikt deze als source. In dit geval onze verwijzing naar de XmlDataProvider.

```
<StackPanel DataContext="{StaticResource
myData}">
<TextBox Name="myTargetTextBox"
Text="{Binding XPath=bookpreview,
Mode=OneWay}"/>
</StackPanel>
```

In dit voorbeeld zal de tekst in de TextBox de inhoud van het bookpreview-element laten zien van het huidige item. De XPath wordt uitgevoerd op de verzameling boeken die we beschikbaar hebben gesteld door middel van het 'myData'-object. Momenteel kunnen we echter geen selectie maken in deze collectie van boeken. Dus we zien constant het bookpreview-element van het eerste boek. Om een ander boek te kunnen selecteren, gaan we een ListBox gebruiken. De ItemsSource property van deze ListBox bepaalt de inhoud van de lijst. Hier willen we de totale collectie aan boeken tonen. Aangezien deze collectie van boeken de DataContext in zijn geheel is, kunnen we een lege binding gebruiken (!) om aan het totale object te binden.

<BIJSCHRIFT>ILLUSTRATIE 4

```
<ListBox ItemsSource="{Binding,
Mode=OneWay}" IsSynchronizedWithCurrent
Item="True">
</ListBox>
```

Dit ziet er op het eerste gezicht wellicht raar uit maar aangezien we kunnen binden aan het totale object in de datacontext hoeven we geen Path op te geven. Het ontbreken van een source zorgt er tenslotte voor dat de dichtstbijzijnde DataContext wordt gebruikt; die van ons StackPanel. IsSynchronizeWithCurrentItem zorgt voor veranderen van selectie in de ListBox naar de active index in ons XmlDataProvider-object. Belangrijk om eventuele andere afnemers van de DataContext op de hoogte te brengen van de verandering.

De uitkomst van ons programma is nog niet wat we willen, want we zien nog niet de gewenste gegevens in onze lijst. Voor het opmaken van de items in onze lijst gebruiken we de ItemTemplate property van de ListBox. Met behulp van een DataTemplate geven we aan hoe de items in de listbox getoond moeten worden.

```
<ListBox ItemsSource="{Binding}" IsSyn-
chronizedWithCurrentItem="True">
    <ListBox.ItemTemplate>
<DataTemplate>
    <StackPanel>
<TextBlock>
    <TextBlock Text="Title : "/>
    <TextBlock Text="{Binding XPath=
@title, Mode=OneWay}"/>
</TextBlock>
<TextBlock>
    <TextBlock Text="ISBN: "/>
    <TextBlock Text="{Binding XPath=
@ISBN, Mode=OneWay}"/>
</TextBlock>
    </StackPanel>
</DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

De notatie van het definiëren van een ItemTemplate met ListBox.ItemTemplate geeft aan dat we de ItemProperty van ListBox uitgebreid gaan definiëren. Net als bij Window.Resources eerder in dit artikel. We kunnen namelijk niet een volledige DataTemplate op één regel kwijt. Properties mag je in XAML altijd op deze manier definiëren.

```
<TextBox>
  <TextBox.Text>
    Hi
  </TextBox.Text>
</TextBox>
```

Staat gelijk aan

```
<TextBox Text="Hi">
```

Met als resultaat dat we nu in de ListBox een lijst van items volgens een zelf opgemaakt template tonen. Het wijzigen van het geselecteerde item zorgt dat de index in onze DataContext wijzigt en daarmee ook de samenvatting in de TextBox.
Uniek aan dit voorbeeld is dat we uitsluitend gebruik hebben gemaakt van XAML zonder een regel C#-code. Geen OnSelectionChanged events, geen synchronisaties met het XML-bestand, helemaal niks. Enkel het opmaken van het XAML-document zorgt voor het gedrag van de applicatie. De notificaties vanuit de Dependency Objects zorgen voor de update van informatie in alle luisterende objecten.

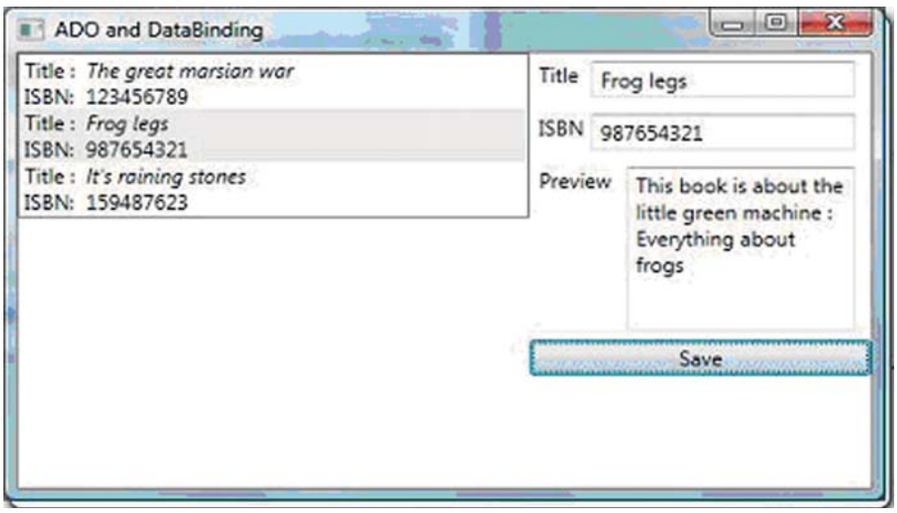
TwoWay binding met ADO.NET

In het eerste voorbeeld hebben we gebruikgemaakt van een binding die uitsluitend bestaande informatie toont. In het tweede voorbeeld richten we ons op een TwoWay binding. We nemen hetzelfde voorbeeld maar gebruiken nu een database om onze gegevens uit te halen. De tabel in de database ziet er als volgt uit:

Tabel naam: Book	
BookID	Int (auto increment +1)
BookTitle	Nvarchar(50)
BookISBN	Nvarchar(50)
BookPreviewText	Nvarchar(MAX)

We willen in dit voorbeeld wijzigingen doorvoeren naar de database met een update-knop. We gaan de data beschikbaar maken op het niveau van ons hoofdscherm en de gegevens tonen in een lijst. In de tekstvakken aan de rechterzijde zien we de informatie nogmaals en we kunnen deze ook wijzigen. De uiteindelijke update naar de database zal gaan via een knop.

In plaats van het creëren van een XmlDataProvider-object in de resource-collectie van



ILLUSTRATIE 5. TWOWAY BINDING MET ADO

ons Window (zoals in het eerste voorbeeld) gaan we nu een .NET DataSet creëren en vullen met data uit de database. Dit doen we niet in XAML maar in C#-code.

```
public Window1()
        {
            InitializeComponent();
            LoadDataContext();
        }
```

Het plaatsen van code voor of na InitializeComponent doet er niet toe bij het laden van gegevens in de DataContext van het hoofdscherm. Deze volgorde is alleen van belang als we properties moeten zetten op elementen in het window1-object. Neem voor de vorm aan dat je alle aanvullende code ná InitializeComponent plaatst. LoadDataContext ziet er als volgt uit.

```
private void LoadDataContext()
        {
            DataSet bookDs = new Data-
Set();
            SqlConnection con = new
SqlConnection(Properties.Settings.Default.
BooksConnection);
            SqlCommand com = new
SqlCommand("SELECT * FROM Book", con);
            SqlDataAdapter da = new
SqlDataAdapter(com);
            try
            {
                con.Open();
                da.Fill(bookDs, "Book");
                con.Close();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Something
went wrong : ", ex.Message);
            }
            this.DataContext = bookDs;
        }
```

We vullen een DataSet *bookDs* met alle gegevens uit de Book-tabel uit onze database en plaatsen deze in een de DataTable *Book* in onze DataSet. Ten slotte koppelen we *bookDs* aan de DataContext van het huidige object, Window1, ofwel ons hoofdscherm.
Op dit moment hebben we de DataSet beschikbaar in ons hoofdscherm en kunnen we hieraan gaan binden. Aangezien we ditmaal wijzigingen willen doorvoeren naar de bron maken we gebruik van een TwoWay inding. Deze binding-mode is de defaultmode dus we hoeven deze niet specifiek aan te geven in onze XAML-code. We nemen wederom een ListBox en een aantal Tekstvakken om de data in weer te geven.

```
<ListBox Name="lstBooks" IsSynchronizedWit
hCurrentItem="True" ItemsSource="{Binding
Path=Books}">
```

De ItemsSource van de ListBox koppelen we niet aan de gehele DataSet, maar enkel aan de DataTable Books die we hierin hebben aangemaakt en gevuld met gegevens. Dit doen we door de property Path op te geven naar de DataTable naam. IsSynchronizedWithCurrentItem zorgt ervoor dat de active-rij in de tabel wordt geüpdatet in de DataContext en dus naar alle leden van deze DataContext.
De TextBoxen binden we tevens aan de DataContext:

```
<TextBlock  Text="Title" Width="30"/>
<TextBox Text="{Binding Path=Book/BookTi-
tle}" Width="150" />

<TextBlock  Text="ISBN" Width="30" />
<TextBox Text="{Binding Path=Book/BookIS-
BN}" Width="150">
```

Let op dat het pad van de binding via de DataTable Book gaat.
Tot zover weinig nieuws. Bij het compileren worden de boeken in de lijst getoond en de details verschijnen in de tekstvakken, net als bij ons XML-voorbeeld. Nieuw is de functie dat we nu de informatie kunnen wijzigen. Zodra we de focus van het actieve tekstvak halen worden de wijzigingen doorgevoerd. Klaar? Nee, nog niet helemaal. De wijzigingen worden op dit moment alleen doorgevoerd in de DataSet, niet naar de database. We dienen dus nog een update uit te voeren met de SqlDataAdapter.Update()-functie. Deze actie heb ik voor het voorbeeld achter de knop geprogrammeerd en ziet er vergelijkbaar uit met de vul-actie:

```
private void btnUpdate_Click(object sen-
der, RoutedEventArgs e)
        {
            DataSet bookDs = this.DataCon-
text as DataSet;
SqlConnection con = new
```

ILLUSTRATIE 6

Zowel doelobject als bron wordt synchroon gehouden, dit type binding is het meest resource-intensief.

```
SqlConnection(Properties.Settings.Default.
BooksConnection);
            SqlCommand com = new
SqlCommand("SELECT * FROM Book", con);
            SqlDataAdapter da = new
SqlDataAdapter(com);
            SqlCommandBuilder combld = new
SqlCommandBuilder(da);
            da.UpdateCommand = combld.
GetUpdateCommand();
            try
            {
                con.Open();
                da.Update(bookDs,
"Books");
                con.Close();
            }
            catch (Exception ex)
            {
                MessageBox.Show("Something
went wrong : ", ex.Message);
            }
            MessageBox.Show("Update suc-
cesful");
        }
```

We halen hier de DataSet simpelweg uit de DataContext en voeren vervolgens een update door naar de brondatabase.

Tot slot

We hebben natuurlijk maar een fractie gezien van wat Data Binding allemaal kan. Maar duidelijk is dat deze manier van werken met data een andere denkwijze eist. Experimenteer gerust verder met het binden aan bijvoorbeeld ADO.NET-objecten, resultaten van Webservices, LINQ queries, CLR-objecten en bekijk vooral ook de mogelijkheden van de klasse CollectionView (http://msdn.microsoft.com/en-us/library/system.windows.data.collectionview.aspx). Hiermee kun je eenvoudig grote hoeveelheden data groeperen, sorteren en filteren. Wil je jezelf verder verdiepen in Data Binding? Bestudeer dan MultiBinding, het binden aan een uitkomst van verschillende objecten, of bekijk de mogelijkheden van conversies en validaties. Veel succes!

Joris Bos is werkzaam als trainer bij 4dotnet en schrijft daarnaast regelmatig artikelen over WPF op zijn weblog: jorisbos.wordpress.com.