



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

Business applications

Data-driven applications met .net

Inhoudstafel

Doelstellingen	6
Evaluatie.....	6
Model View ViewModel (MVVM)	6
Opwarmingsoefening.....	7
Styles en Templates in XAML.....	8
Intro.....	8
De class Style	8
Eerste demo	8
Styles schrijven in C#	9
Werken met resources	9
Een resource aanmaken.....	9
Local versus global	10
Resource dictionary	10
Style inheritance	11
Triggers	11
ControlTemplate.....	12
Styles in Windows Store Apps	12
Oefeningen Styles en templates	13
Oefening 1	13
Oefening 2	15
Databinding.....	15
Intro.....	15
Dependency property.....	15
Basis concept	16
Eerste demo	17
DataContext	17
Databinding in Expression Blend	18
Converteren van data	18
Binding van element naar element	19
Databinding met collections	19
Master-detail scenario.....	20
DataTemplates	20
Oefeningen databinding	21
Oefening 1	21
Oefening 2	23
ObservableCollection	25
Intro.....	25



INotifyPropertyChanged	25
INotifyCollectionChanged	26
Eerste demo	26
Commands en Routed Events.....	28
Intro.....	28
Logical tree en Visual tree.....	28
Routed Events	28
Commands.....	29
Terminologie.....	29
Demo	29
Oefeningen ObservableCollection, Commands en Routed Events	30
Oefening 1	30
Oefening 2.....	32
Databases ontwerpen.....	35
Intro.....	35
Relationele database	35
Definitie	35
Terminologie.....	35
Database normalisatie.....	36
1NF: eerste normaalvorm.....	36
2NF: tweede normaalvorm	37
3NF: derde normaalvorm.....	38
Soorten relaties.....	39
1 op 1 relatie	39
1 op N relatie	40
N op N relatie	40
Praktijkvoorbeeld	41
Identificeren entiteiten.....	41
Identificeren attributen	42
Toepassen eerste normaalvorm	42
Toepassen tweede normaalvorm.....	42
Toepassen derde normaalvorm	43
Van database model naar object model	43
Oefeningen: databases ontwerpen.....	44
Oefening 1	44
Oefening 2.....	44
Oefening 3	45
ADO.NET	45
Intro.....	45
Web Api	45



Structuur van ADO.NET.....	45
DbProviderFactories	46
Connectie maken	47
Command opstellen	49
Parameters meegeven	49
Commando uitvoeren.....	49
Resultaat verwerken.....	49
DBNull.....	50
Transacties	50
Error handling.....	51
Oefeningen ADO.NET.....	51
Oefening 1	51
Oefening 2.....	52
Oefening 3.....	55
Model View ViewModel (MVVM)	56
Intro.....	56
Eerste demo	56
Algemene structuur	56
Model.....	58
View	59
ViewModel	59
View: Deel 2	62
Conclusie demo.....	63
MVVM frameworks.....	63
Advanced commands	63
Meegeven van een parameter	63
Event omzetten naar command.....	64
Navigatie binnen MVVM	64
Navigatie binnen MVVM: demo	65
Oefeningen MVVM.....	67
Oefening 1	67
Oefening 2	68
Data validatie	69
Intro.....	69
Demo: conversie van integer naar string	70
ErrorTemplate	70
Validation rules	71
IDataErrorInfo.....	72
Data annotations	75
Enabelen en disabelen van controls	76



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

CURSUS BUSINESS APPLICATIONS

Doelstellingen

Tijdens deze module leer je gebruiksvriendelijke desktop applicaties ontwikkelen die de **gegevens uit een databron verwerkt en presenteert via een grafische user interface**. Deze databron kan zowel een database, een service of elk platform zijn dat gestructureerde data aanbiedt. De module maakt gebruik van de object georiënteerde ontwikkelingsmethode als aangeleerd in de module Object Oriented Programming uit semester 2.

Daarnaast zal de module ook gebruik maken van het **design pattern Model View ViewModel (MVVM)** dat het mogelijk maakt om de code beter te structureren om zo de kwaliteit naar onderhoudbaarheid toe te verhogen.

Aangezien dit een module is uit semester 3 moet je ook in staat zijn om op een **zelfstandige basis extra informatie over de geziene topics te kunnen opzoeken**. Er is tijdens deze module ook een project waar je op zelfstandige basis aan zal moeten werken.

Evaluatie

Het eindcijfer van de module wordt als volgt samengesteld:

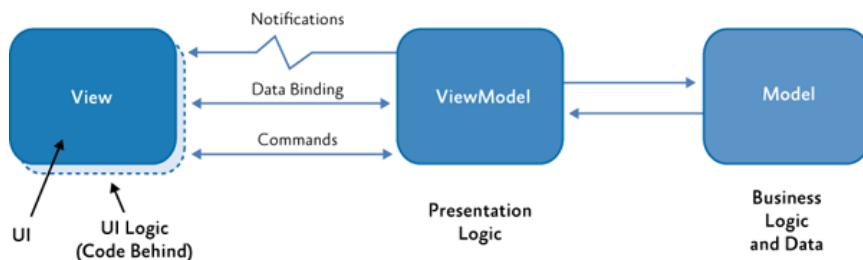
- Een schriftelijk examen over de theorie: 20%
- Een examen op je laptop over het labo: 40%
- Een project dat over gans de module loopt: 20%
- Het DES punt: 20%

Belangrijk om weten bij het project en de DES score is dat dit gedeelte **NIET hernomen kan worden in de tweede zittijd**. Je score van de eerste zittijd neem je dus mee voor dit onderdeel.

Model View ViewModel (MVVM)

MVVM is een architectural pattern dat gebruikt wordt voor WPF applicaties en als voornaamste doel heeft om een strikte scheiding te voorzien tussen de business logic code en de grafische user interface. Om dit doel te realiseren wordt er een systeem van databinding gebruikt. MVVM bestaat uit de volgende onderdelen:

- **Model:** dit is alle code die je data model beschrijft, aangevuld met alle code om data op te halen of te manipuleren in de database. In ons geval is dit C# code.
- **View:** dit is de grafische user interface van je applicatie. In ons geval is dit XAML code.
- **ViewModel:** is de laag die tussen het model en de view zit en in feite het model van de view. Deze laag zorgt ervoor dat het model en de view niet rechtstreeks met elkaar moeten communiceren.

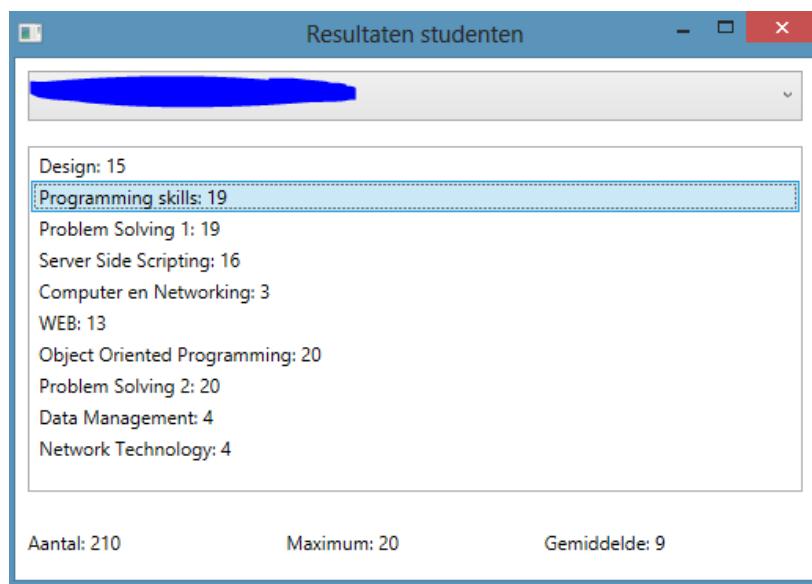


Figuur 1: het MVVM pattern ([http://msdn.microsoft.com/en-us/library/gg405484\(v=pandp.40\).aspx](http://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx))

Opwarmingsoefening

Als start van de cursus wordt er een korte oefening gemaakt die enkele concepten uit C# en WPF herhaalt ter oprissing van de topics die gezien werden in de module Object Oriented Programming. Normaal gezien moet je in staat zijn om deze oefening zelfstandig te kunnen maken.

Er is een CSV bestand gegeven dat telkens het e-mail adres van een student, zijn geboorteplaats, de naam van een module en zijn score voor die module bevat. Het is de bedoeling om een applicatie te maken waarbij een student geselecteerd kan worden. Alle resultaten van deze student worden dan in een listbox getoond. Bij het selecteren van een module uit de listbox worden het aantal resultaten, het maximum en het gemiddelde van deze module getoond.



Figuur 2: opwarming – eindresultaat

- Maak een nieuw WPF project aan en bouw in XAML de user interface na:
 - 3 rijen: de bovenste en onderste hebben een height van 48, de middelste rij neemt de rest van het grid in
 - In de bovenste rij komt een combobox met een margin van 8
 - In de middelste rij komt een listbox
 - De onderste rij wordt verdeeld in 3 kolommen waar telkens een textblock in komt
- Maak een class StudentModulePunt aan met als properties:
 - Email
 - Geboorteplaats
 - Module
 - Punt
- Maak een static method in deze class die de gegeven CSV file zal verwerken en als resultaat een List van StudentModulePunt instanties teruggeeft. Zorg voor de nodige controles: de lijn moet volledig zijn en de score moet een integer zijn.
- Zorg dat deze lijst wordt ingeladen in een module variabele bij het starten van de applicatie
- Maak een static method die een List van strings zal teruggeven met daar in het e-mail adres van elke student. Als input geef je de volledige List van StudentModulePunt instanties mee.
- Zorg dat deze lijst wordt getoond in de combobox.
- Maak een static method die een List van StudentModulePunt instanties zal bevatten op basis van het e-mail adres van een student. Enkel de resultaten van de opgegeven student mogen in de List komen.
- Bij het selecteren van een student in de combobox gebruik je de vorige method om de resultaten in de listbox te tonen. LET WEL: de listbox bevat instanties van de class

StudentModulePunt. Overschrijf de ToString() method om de naam van de module en het punt te zien.

- Maak in de class StudentModulePunt drie static methods om enkele statistieken weer te geven:
 - Het maximum punt van een module
 - Het aantal studenten dat een resultaat heeft voor de module
 - De gemiddelde score van de module
- Roep deze methods op als een module geselecteerd is in de listbox en toon het resultaat in de textblocks. Controleer wel of er een module geselecteerd is voor de methods aan te spreken.

Styles en Templates in XAML

Intro

Bij de introductie van WPF in 2006 was een grote wijziging ten opzichte van Winforms dat de grafische user interface wordt opgemaakt in XAML. Daarnaast kwamen er ook heel wat extra mogelijkheden deze grafische user interfaces op te bouwen. Waar je bij de traditionele Winforms enkel de mogelijkheid had om bijvoorbeeld de kleur of het font van een knop te wijzigen kan je in WPF via XAML de volledige look en feel van elke control wijzigen. In grote lijnen kan je styles in WPF vergelijken met CSS bij HTML.

Elke control in WPF erft over van de class FrameworkElement (<http://msdn.microsoft.com/en-us/library/system.windows.frameworkelement.aspx>). Zoals je in de documentatie kan lezen heeft elke instantie van een FrameworkElement – en dus ook elke control – een property Style (<http://msdn.microsoft.com/en-us/library/system.windows.frameworkelement.style.aspx>). Door de property Style van een control in te vullen zal bepaald worden hoe de control er uit ziet.

De class Style

De class Style wordt volledig beschreven in de MSDN (<http://msdn.microsoft.com/en-us/library/system.windows.style.aspx>). De belangrijkste properties om te onthouden zijn:

- TargetType: geeft aan voor welk type control (Label, TextBlock, Button, ...) de style werd ontworpen.
- Setters: bevat een collectie van alle properties die met deze style worden ingesteld. Elke setter zal een property en een value bevatten.
- BasedOn: deze property maakt het mogelijk om een nieuwe style te laten overerven van een bestaande style.

Eerste demo

Een eerste eenvoudig voorbeeld toont aan hoe deze styles in de praktijk kunnen gebruikt worden.



Figuur 3: eerste demo met styles

```
<Window.Resources>
    <Style x:Key="DemoStyle" TargetType="{x:Type TextBox}">
        <Setter Property="Background" Value="#FFA2E356"/>
```

```
<Setter Property="Foreground" Value="Azure"/>
<Setter Property="BorderThickness" Value="4"/>
<Setter Property="Margin" Value="8"/>
<Setter Property="BorderBrush" Value="GreenYellow"/>
</Style>
</Window.Resources>
<StackPanel>
    <TextBox Style="{StaticResource DemoStyle}" Text="Eerste"/>
    <TextBox Style="{StaticResource DemoStyle}" Text="Tweede"/>
    <TextBox Text="Derde"/>
</StackPanel>
```

Er wordt binnen de resources van het window (later meer over resources) een nieuwe style aangemaakt met als key (x:Key) DemoStyle. Deze style wordt ontworpen voor textboxen (TargetType).

Binnen de <Style> tag worden verschillende <Setter> elementen aangemaakt. Deze hebben als attributen allemaal een Property en een Value. Zoals je kan zien kan je voor kleuren een voor gedefinieerde naam gebruiken of een ARGB code opgeven.

Deze style kan nu op alle gewenste textboxen toegepast worden door de Style property van die Textbox in te vullen. Je verwijst hiervoor naar de Key van de aangemaakte Style. Indien het gewenst is dat alle Textboxen zonder uitzondering deze Style gebruiken kan je het attribuut x:Key weglaten. Zo moet niet telkens de Style property van elke control worden ingevuld.

Het is ook mogelijk om Expression Blend te gebruiken om styles aan te maken en toe te kennen. Zo moet niet alle XAML code manueel getypt worden. Je kan dit doen door een control te selecteren en dan in het menu te kiezen voor Object -> Edit Style.

Styles schrijven in C#

Alles wat je in XAML doet kan je in principe ook in C# doen, dus ook het aanmaken en toekennen van een bepaalde style. In de praktijk zal dit echter zelden voorkomen, maar toch een klein voorbeeld.

```
// aanmaken nieuwe style
Style s = new Style();
// aanmaken en instellen van setter
Setter s1 = new Setter();
s1.Property = TextBox.BackgroundProperty;
s1.Value = Brushes.Red;
// setter toekennen aan de style
s.Setters.Add(s1);
// style toekennen aan de control
txtDemo.Style = s;
```

Werken met resources

Een resource aanmaken

Het begrip resource werd al enkele keren aangehaald bij het eerste voorbeeld. Kort samengevat komt het er op neer dat een resource een stuk XAML code bevat die je op verschillende plaatsen kan hergebruiken. Styles zijn een goed voorbeeld hier van, maar ook colors en brushes. Het volgende voorbeeld zal een gradient brush aanmaken als resource:



Figuur 4: een LinearGradientBrush als resource

```
<Window.Resources>
    <LinearGradientBrush x:Key="MyBrush">
        <GradientStop Color="Yellow" Offset="0.0" />
        <GradientStop Color="Orange" Offset="0.5" />
        <GradientStop Color="Red" Offset="1.0" />
    </LinearGradientBrush>
</Window.Resources>
<StackPanel>
    <Button Width="200" Height="200" Background="{StaticResource MyBrush}" />
</StackPanel>
```

Binnen het element `<Window.Resources>` wordt een nieuw element aangemaakt met als attribuut een key die de unieke naam voor de resource zal bevatten. In het voorbeeld is dit een `LinearGradientBrush`, meer info en voorbeelden over de verschillende soorten brushes is terug te vinden op <http://msdn.microsoft.com/en-us/library/aa970904.aspx>.

Na het aanmaken van de resource kan deze toegekend worden aan een bepaalde property van een control. In dit voorbeeld is dit de property `Background` van een button, maar het kan ook de background of forecolor van een textbox zijn.

Het toekennen van een resource aan een property doe je door eerst accolades te plaatsen, daarna het woord `StaticResource` en tot slot de naam van de key. Naast een `StaticResource` bestaat er ook nog `DynamicResource`, maar dit valt buiten de scope van deze module.

Local versus global

Tot nu toe werden alle resources lokaal aangemaakt tussen de `<Window.Resources>` tags. Dit wil zeggen dat ze enkel gebruik kunnen worden binnen dit Window. Het is echter ook mogelijk om resources globaal voor het volledige project aan te maken.

Elk WPF project heeft een file `App.xaml` om globale resources in te plaatsen. De code voor de resource zal exact dezelfde zijn en ook in het aanroepen van de resource is er geen verschil. Het enige verschil is dat resources in `App.xaml` over het ganse project gekend zijn en niet enkel binnen één Window. Het is dan ook aan te raden om resources zoals `Styles` in de `App.xaml` te plaatsen.

Resource dictionary

Naast lokale en globale resources is het ook mogelijk om alles op te slaan in een Resource Dictionary. Dit is een XAML file die aan verschillende projecten kan toegekend worden. Zo is het mogelijk om een basis style te maken voor alle controls en deze op alle WPF projecten toe te passen. Zo kan je een volledig theme ontwikkelen.

Een Resource Dictionary maak je aan door een nieuwe file aan je project toe te voegen (Add -> Resource Dictionary...). Je plaatst de code voor de nodige styles in deze file.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="MyBrush">
        <GradientStop Color="Yellow" Offset="0.0" />
        <GradientStop Color="Orange" Offset="0.5" />
        <GradientStop Color="Red" Offset="1.0" />
    </LinearGradientBrush>
</ResourceDictionary>
```

Je kan deze Resource Dictionary koppelen aan je project door de link te maken in App.xaml. Het is ook mogelijk om deze link in je Window te maken indien je de styles enkel maar lokaal wenst te gebruiken.

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Dictionary1.xaml"/>
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

Via Codeplex (<http://wpfthemes.codeplex.com>) kan je een ganse boel voorbeelden vinden van themes die je in je project kan gebruiken.

Style inheritance

Het is mogelijk om een Style te laten overerven van een andere Style. Op die manier kan je kleine varianten laten maken zonder alles opnieuw te hoeven tikken. We kunnen bijvoorbeeld een variant maken op de Style uit de eerste demo waar we de dikte van de rand van de Textbox willen aanpassen en het lettertype:

```
<Style x:Key="..." BasedOn="{StaticResource DemoStyle}" TargetType="{x:Type TextBox}">
    <Setter Property="FontFamily" Value="Arial"/>
    <Setter Property="BorderThickness" Value="1"/>
</Style>
```

Zoals de demo aantoon is het mogelijk om zowel nieuwe properties toe te voegen als bestaande te overschrijven. Het attribuut BasedOn geeft aan van welke stijl we moeten overerven.

Het overerven van styles of het aanmaken van styles op verschillende locaties kan echter wel voor verwarring zorgen. Let dus goed op de naamgeving dat er geen conflicten kunnen opduiken. Algemene regel voor styles is dat de style die het "dichtst" bij het element staat voorrang krijgt. Concreet betekent dit dat local resources voorrang hebben op global resources en dat global resources voorrang hebben op resource dictionaries.

Triggers

Een trigger kan zorgen voor een kleine animatie of een effect in de style bij een bepaalde actie. Het grote voordeel van deze triggers is dat deze kleine effecten geen C# code meer nodig hebben in de code behind.

Er bestaan verschillende soorten triggers. Momenteel focussen we enkel op de Property Triggers. Deze triggers zullen een bepaalde actie uitvoeren als de property van een element wijzigt naar de waarde die we willen. Een voorbeeld met een knop toont aan hoe dit in de praktijk werkt:

```
<Style TargetType="{x:Type Button}">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="FontWeight" Value="Bold"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

Er wordt een style voor alle buttons aangemaakt met binnen deze style een sectie <Style.Triggers>. Binnen die sectie wordt er maar één trigger aangemaakt, namelijk voor de IsMouseOver property. De MSDN toont een overzicht van alle properties die mogelijk zijn voor een button (<http://msdn.microsoft.com/en-us/library/system.windows.controls.button.aspx>). Als de value van deze property wijzigt naar true dan wordt de tekst binnen de button op bold gezet. Als deze property terug false wordt zal de property aangepast worden naar de oorspronkelijke waarde.

Het is ook mogelijk om een korte animatie af te spelen via een storyboard in plaats van de waarde direct te wijzigen. Dit wordt gedaan aan de hand van EnterActions en ExitActions waarbinnen een storyboard wordt aangemaakt om de animatie af te spelen.

```
<Style TargetType="{x:Type Button}">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Trigger.EnterActions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetProperty="Opacity" To="1" Duration="0:0:1" />
                    </Storyboard>
                </BeginStoryboard>
            </Trigger.EnterActions>
            <Trigger.ExitActions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetProperty="Opacity" To="0.25" Duration="0:0:1" />
                    </Storyboard>
                </BeginStoryboard>
            </Trigger.ExitActions>
        </Trigger>
    </Style.Triggers>
</Style>
```

Het is wel aan te raden om Expression Blend te gebruiken om animaties aan te maken. Alle info over storyboards maken met Expression Blend kan je vinden op <http://msdn.microsoft.com/en-us/library/cc295346.aspx> en alle info over storyboards in het algemeen is terug te vinden op <http://msdn.microsoft.com/en-us/library/system.windows.media.animation.storyboard.aspx>.

ControlTemplate

Tot nu toe werd er bij een Style enkel properties aangepast om het uitzicht van de control een beetje aan te passen. Het is echter ook mogelijk om de volledige structuur van de control aan te passen zodat je bijvoorbeeld ronde buttons kan maken of een originele checkbox.

Het volgende voorbeeld toont hoe een rode knop wordt aangemaakt:

```
<ControlTemplate x:Key="TemplateDemo" TargetType="Button">
    <Grid>
        <Ellipse Fill="Red"/>
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </Grid>
</ControlTemplate>
```

In plaats van een `<Style>` tag wordt er gebruik gemaakt van een `<ControlTemplate>` tag hoewel het ook mogelijk is om de template property binnen een `<Style>` element te gebruiken. Binnen het `<ControlTemplate>` element kan de structuur van de knop worden opgegeven. Let wel dat er altijd een ContentPresenter element aanwezig moet zijn om het mogelijk te maken om de content van de knop weer te geven. Via `{StaticResource TemplateDemo}` kan je deze template dan toekennen aan het attribuut Template van de knop.

ControlTemplates kunnen enorm complex zijn en dit is dan ook een heel eenvoudig voorbeeld. We gaan dieper in op templates bij het behandelen van het concept databinding.

Styles in Windows Store Apps

Naast de gewone traditionele desktop applicaties kan je ook specifieke Windows 8 applicaties maken die zowel op desktop als op tablet beschikbaar zijn. Later in deze module meer over dit soort applicaties.

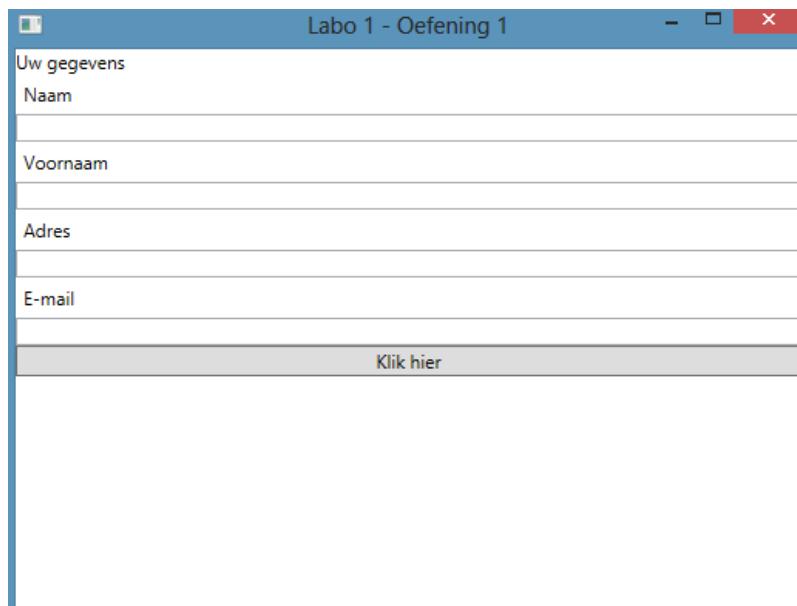
Momenteel is het enkel belangrijk om te onthouden dat deze applicaties een bepaalde style hebben die Metro wordt genoemd. Alle info over de Metro style is terug te vinden op <http://msdn.microsoft.com/en-us/library/windows/apps/hh465424>. Kort samengevat komt het neer op "Content over chrome" wat zoveel wil zeggen als: de inhoud staat centraal en de UI wordt zo sober mogelijk gehouden.

Daarom zijn er een aantal voorgemaakte Styles voor deze applicaties. Als je in Visual Studio een nieuw Windows Store project aanmaakt zie in in de map Common de file StandardStyles.xaml staan. Dit is een ResourceDictionary die de standaard layout voor de controls bevat. Als je dan een control op je page plaatst zal die automatisch de metro style aannemen.

Oefeningen Styles en templates

Oefening 1

- Maak een nieuw WPF project aan
- Maak van het grid een stackpanel en plaats er een textblock, 4 labels, 4 textboxen en een button op zoals in onderstaande screenshot.



Figuur 5: labo 1 - oefening 1 start

- Maak een style aan voor het textblock met volgende eigenschappen:
 - Stel de key in voor de style (bijvoorbeeld: Title)
 - De achtergrond is #FFE85E00
 - De tekstkleur is wit, met als lettertype Ubuntu en een size van 36 in Extrabold
 - De padding is overal 8px
- Maak een style aan voor alle labels met volgende eigenschappen:
 - Font is Ubuntu
 - Kleur is #FFE85E00
 - Marges zijn 8px links en boven, 0px onder
 - Padding is overal 2px
- Maak een style aan voor alle textboxen met volgende eigenschappen:
 - Font is Ubuntu
 - Fontsize is 14
 - De rand heeft als kleur #FF3785B2 maar de dikte van de rand staat op 0
 - De achtergrond is #FFE85E00

- Het tekstkleur is gewoon wit
- Er is langs alle zijden een padding van 2px
- De marges zijn 12 links, 48 rechts en 0 boven en onder
- Maak een style aan voor de button met de volgende eigenschappen:
 - De font is Ubuntu met een size van 18
 - De height van de knop is 48px
 - De marges zijn 12 links, 16 top, 48 rechts en 0 onder
 - De padding is overall 2px
 - De achtergrond is #FFD4D4D4
 - De rand heeft een blauwe kleur (#FF3785B2) en heeft een dikte van 4px
 - De kleur van de tekst is dezelfde als van de rand
- Nadat je al deze styles ontwerpen hebt en toegekend aan de juiste elementen zou het resultaat als in de onderstaande screenshot moeten zijn:



Figuur 6: labo 1 - oefening 1 eindresultaat

- Tijdens de volgende stappen zullen we de XAML code optimaliseren zodat we zoveel mogelijk kunnen hergebruiken.
 - Verplaats alle styles naar App.xaml
 - Maak van alle kleuren een global resource in App.xaml en test dit door de kleuren eens aan te passen.
 - Maak ook een LinearGradientBrush aan als resource voor de hoofding van het oranje naar een gele kleur en pas deze toe op de achtergrond
 - Kijk wat de verschillende styles gemeenschappelijk hebben en probeer zoveel mogelijk overerving te gebruiken. Welke properties kan je laten overerven? Zijn er problemen met de Textblock? Welke TargetType geef je mee bij de basis style?
- Voeg de nodige triggers toe:
 - De border van een textbox krijgt een dikte van 2px als hij de focus krijgt. Zoek welke property je hier voor nodig hebt.
 - De text van de knop wordt wit als je over de knop beweegt en de dikte van de rand wordt 8. Wat merk je als je over de knop beweegt? Hoe komt dit?
- Probeer die overgangen met een animatie te doen van 0.2 seconde. Maak het storyboard aan Expression Blend en bekijk de XAML die werd aangemaakt. Kopieer dan deze XAML in je style.
- Maak een nieuwe style aan voor de hoofding met enkele andere kleuren en opmaak en geef deze als titel bijvoorbeeld Title2. Je kan hier ook gebruik maken van overerving.
- Bij het klikken op de knop moet de style van de hoofding aangepast worden naar de nieuwe style. Nogmaals klikken wijzigt terug naar de vorige style. Kijk op

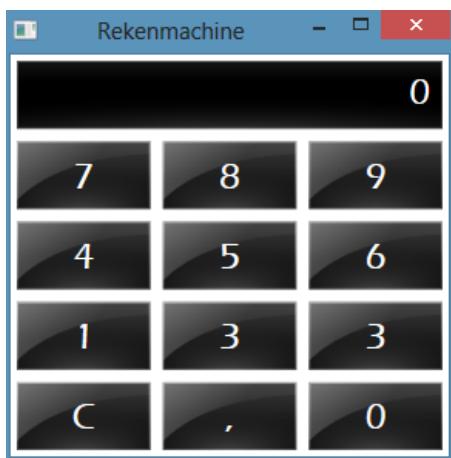
<http://msdn.microsoft.com/en-us/library/system.windows.application.findresource.aspx> om te zien hoe je dit kan aanpakken en schrijf de nodige code in C#

- Download een Style theme van <http://wpfthemes.codeplex.com/SourceControl/latest> en voeg dit toe aan je project. Wordt de style direct toegepast? Is de aanpassing voor alle elementen? Waarom niet? Pas de code aan zodat de style van de theme wordt gebruikt.

Oefening 2

Maak een nieuw WPF project aan en schrijf de nodige XAML code om de volgende user interface na te maken. Voor de opmaak maak je gebruik van een bestaand theme dat je via codeplex kan downloaden. Voeg wel nog je eigen opmaak toe om er voor te zorgen dat je kleine aanpassingen kan doen met het theme. Zorg er voor dat elk element zo weinig mogelijk attributen heeft door deze in de style te stoppen.

Om er voor te zorgen dat de style van het theme niet volledig verdwijnt laat je je nieuwe style overerven van het theme: `BasedOn="{StaticResource {x:Type Button}}}`



Figuur 7: labo 1 - oefening 2 eindresultaat

Databinding

Intro

Databinding is één van de belangrijkste concepten van WPF en maakt het mogelijk om het basisprincipe van MVVM – de scheiding tussen Business Logic en de GUI – na te streven. Via databinding kunnen we properties van objecten vastmaken aan elementen in de GUI en deze automatisch laten updaten. Een goede databinding zal er dus voor zorgen dat je veel minder C# code hoeft te schrijven. Let wel, databinding is geen uniek concept voor WPF en je zal dit principe nog in veel andere programmeertalen terugvinden.

Dependency property

Dependency properties zijn een redelijk complex gegeven in WPF. Waar bij gewone properties de waarde in een private field wordt opgeslagen, wordt dit bij een dependency property in een Dictionary gedaan met een key (de naam) en een value (de waarde). Alle verdere info over

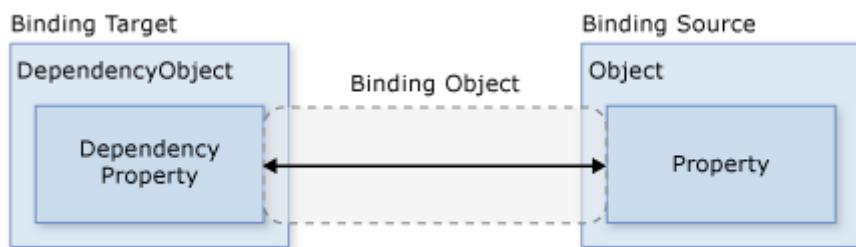
dependency properties is te vinden op <http://msdn.microsoft.com/en-us/library/system.windows.dependencyproperty.aspx>

Dependency properties hebben 2 grote voordelen:

- Als een dependency property geen waarde heeft wordt naar de parent elementen van die property gekeken in de Visual Tree. Bijvoorbeeld: als de FontSize (een dependency property) van een Grid op 12pt staat zullen alle childs van het grid deze size overnemen zelfs al is ze niet expliciet zo ingesteld voor die elementen.
- Een ander voordeel is dat dependency properties een mechanisme hebben voor change-notifications. Daarom zijn ze heel handig bij databinding.

Basis concept

Het basis concept van databinding is in feite heel eenvoudig en wordt getoond in onderstaande figuur.



Figuur 8: concept van databinding (<http://msdn.microsoft.com/en-us/library/ms752347.aspx>)

Bij databinding is er een source en een target. De source is een bepaald object. Dit kan een instantie van een class zoals Employee, Car, Person, ... zijn, een XML document of zelfs een bepaalde control. Het target moet een DependencyObject zijn, maar alle grafische controls in WPF zijn DependencyObjects.

Bij databinding wordt er een bepaalde property van de source vast gemaakt aan een dependency property van het target. Een concreet voorbeeld is de property Name van een object Employee vastmaken aan de dependency property Text van een dependency object Textbox.

De dubbele pijl geeft aan dat databinding mogelijk is in verschillende richtingen. Er zijn drie belangrijke modes en elke dependency property heeft zijn eigen default mode:

- **OneWay**: een wijziging in de source zal het target updaten, maar het target wijzigen zal de source niet updaten. Dit wordt gebruikt voor het weergeven van data.
- **TwoWay**: een wijziging in de source zal het target updaten en een wijziging in het target zal de source updaten. Dit wordt meestal gebruikt bij textboxen, checkboxen, ...
- **OneWaytoSource**: dit is het omgekeerde van OneWay binding. Een wijziging in het target zal de source updaten, maar niet omgekeerd.

Er zijn ook drie verschillende momenten wanneer een update kan gebeuren van het target naar de source. Hier heeft opnieuw elke dependency property zijn eigen default:

- **LostFocus**: wanneer de control zijn focus verliest
- **PropertyChanged**: bij elke wijziging in de control
- **Explicit**: bij een expliciete actie zoals het klikken op een knop

Aangezien elke dependency property zijn eigen default waarde heeft is het niet altijd nodig om de binding mode en het update moment mee te geven. Meestal zal de default waarde volstaan.

Eerste demo

Bij deze demo wordt er een class Person aangemaakt met één property, een string die de naam van de persoon zal bevatten.

```
namespace Testapp
{
    public class Person
    {
        private string _name = "Frederik";
        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }
}
```

De inhoud van deze property wordt gebind aan de text property van een textblock. Volgende elementen zijn dus nodig:

- Een source object: een instantie van de person class
- Een source property: de property Name van deze instantie
- Een target dependency object: in dit geval de textblock
- Een target dependency property: in dit geval de property Text van de textblock

```
<Window x:Class="Testapp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:p="clr-namespace:Testapp"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <p:Person x:Key="MyPerson"/>
    </Window.Resources>
    <StackPanel>
        <TextBlock Text="{Binding Source={StaticResource MyPerson}, Path=Name}"/>
    </StackPanel>
</Window>
```

Er wordt eerst een xmlns (xml namespace) aangemaakt die verwijst naar de namespace van de class. Binnen de resources – in dit geval resources van het Window, maar kan ook in global resources of resources van het StackPanel – wordt er een nieuwe instantie aangemaakt van de class Person met als key Myperson. Op die manier is er dus een source object en kan de property Name gebruikt worden als source property.

De lijn `Text="{Binding Source={StaticResource MyPerson}, Path=Name}"` zorgt voor de uiteindelijke binding. De source wordt ingesteld op de resource MyPerson en de gevraagde property (path) is Name. Voor de bind mode (OneWay, TwoWay of OneWayToSource) en de UpdateSourceTrigger worden de default waarden automatisch overgenomen.

Datacontext

Tijdens de eerste demo werd er slechts één property van een object gebind aan één control. In de praktijk zullen natuurlijk meerdere properties gebind moeten worden. Bijvoorbeeld: wanneer alle data zoals naam, voornaam, adres, van een bepaalde persoon moet worden weergegeven. In dergelijke gevallen is het aan te raden om de datacontext te gebruiken zoals in onderstaand voorbeeld:

```
<StackPanel>
    <StackPanel.DataContext>
```

```

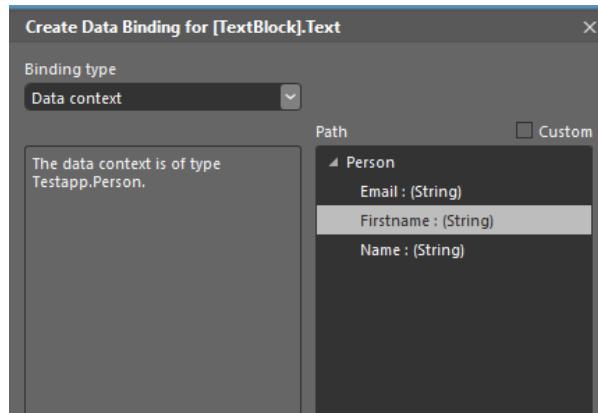
<Binding Source="{StaticResource MyPerson}"/>
</StackPanel.DataContext>
<TextBlock Text="{Binding Path=Name}"/>
<TextBlock Text="{Binding Path=Firstname}"/>
<TextBlock Text="{Binding Path=Email}"/>
</StackPanel>

```

In dit geval moeten we slechts één maal de source instellen, deze wordt dan automatisch overgenomen voor alle child elementen van het stackpanel.

Databinding in Expression Blend

Tot nu toe hebben we alle bindings in XAML ingetikt, maar Expression Blend geeft enorm veel ondersteuning voor databinding. Het grote voordeel hierbij is dat je niet alle namen van de properties moet kennen en het bespaart je ook enorm veel tikwerk.



Figuur 9: databinding in Expression Blend

Converteren van data

Tijdens de vorige demo's werden enkel string waarden gebind aan een text property die ook van het type string is. Maar in sommige gevallen is de binding niet altijd zo eenvoudig. Bijvoorbeeld: als je een kleur (rood, oranje, groen) wil toekennen afhankelijk van een bepaald getal of als je een checkbox aan of uit wil zetten afhankelijk van een leeftijd.

Voor dergelijke conversies bestaat er een interface `IValueConverter` (<http://msdn.microsoft.com/en-us/library/system.windows.data.ivalueconverter.aspx>). Deze interface bevat 2 methods, namelijk `Convert()` en `ConvertBack()`. Aangezien we tot nu toe enkel data in één richting binden van de source naar het target moet enkel de method `Convert()` worden ingevuld.

Onderstaand voorbeeld toont hoe een score kan omgezet worden naar een kleur. Afhankelijk van de score zal het getal omgezet worden naar een rode, oranje of groene brush. Deze brush kan dan gebruikt worden om een element op de user interface in te kleuren.

```

[ValueConversion(typeof(int), typeof(SolidColorBrush))]
public class ScoreToColorConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,...)
    {
        int score = (int)value;
        if (score < 10)
            return new SolidColorBrush(Colors.Red);
        else if (score >= 10 && score < 12)
            return new SolidColorBrush(Colors.Orange);
    }
}

```

```

        else return new SolidColorBrush(Colors.Green);
    }
}

```

Natuurlijk moet er in XAML ook worden aangegeven dat deze converter moet gebruikt worden voor de binding. Eerst wordt er xmlns aangemaakt voor de converters waarna in de resources een instantie wordt aangemaakt. Tot slot moet als attribuut van de binding aangegeven worden dat er een conversie nodig is.

```

<Window.Resources>
    <p:Person x:Key="MyPerson"/>
    <c:ScoreToColorConverter x:Key="ScoreToColorConverter"/>
</Window.Resources>
    <StackPanel DataContext="{Binding Source={StaticResource MyPerson}}">
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Firstname}"/>
        <TextBlock Text="{Binding Email}"/>
        <Rectangle Fill="{Binding Score, Converter={StaticResource ScoreToColorConverter}}" Height="20"/>
    </StackPanel>

```

Merk ook nog op dat Path de default property is voor de binding. Je mag het woord Path dus zelf gerust weglaten.

Binding van element naar element

Tot dus toe hebben we enkel een binding gemaakt tussen een Person object (source) en een UI element (target). Het is echter ook mogelijk om zowel voor de source als voor het target een UI element te gebruiken. Je kan bijvoorbeeld de waarde van een slider binden aan de waarde van een textbox en deze binding in twee richtingen laten verlopen.

```

<TextBox x:Name="MyTextbox" Text="20"/>
<Slider Value="{Binding ElementName=MyTextbox, Path=Text, Mode=TwoWay}"/>

```

De waarde van de slider wordt gebind aan de textbox via het attribuut ElementName. Daarom moet de textbox wel een naam krijgen. De property waar aan gebind wordt is Text en de binding verloopt in 2 richtingen. Dit houdt in dat het verschuiven van de slider zal zorgen voor het aanpassen van de tekst in de textbox en dat de waarde van de textbox aanpassen zal resulteren in het verschuiven van de slider.

Databinding met collections

Een volgende belangrijk onderdeel is het binden aan een collectie van data zoals aan een array. Tijdens alle vorige demo's werden ContentControls gebruikt zoals Textblocken, Textboxen, Buttons,... Om een collection te binden aan een UI element is er een ItemsControl nodig zoals een listbox, combobox, listview of datagrid.

Al deze ItemsControls hebben gemeenschappelijk dat ze een property ItemsSource hebben die de volledige collectie van data bevat. Momenteel ligt de focus op OneWay binding. Wanneer we de ObservableCollection en de INotifyPropertyChanged interface behandeld hebben is het ook mogelijk om TwoWay binding uit te voeren op een collectie. Voorlopig maken we echter gebruik van de List<T> collection (<http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx>).

Stel dat ons Person object een property Friends heeft die een lijst van personen bevat met alle vrienden van deze persoon, dan is het mogelijk om deze lijst te binden aan een listbox of combobox zoals in het volgende voorbeeld:

```

public List<Person> Friends
{
    get {
        _friends = new List<Person>();
        _friends.Add(new Person(){Name = "Bostyn", Firstname = "Henk", ...});
    }
}

```

```
_friends.Add(new Person() { Name = "Walcarius", Firstname=... });
    return _friends;
}
set { _friends = value; }
}
```

Merk op dat het codevoorbeeld hierboven gebruik maakt van een object initializer zodat de properties van een nieuwe instantie direct kunnen ingevuld worden. De XAML voor de binding ziet er dan als volgt uit:

```
<ListBox ItemsSource="{Binding Friends}" DisplayMemberPath="Name"/>
```

Er is niet momenteel veel verschil tussen het binden met een ContentControl, uitgezonderd dat er via DisplayMemberPath wordt opgegeven welke property er moet weergegeven worden in plaats van de default ToString() method van het object. Als we meer informatie willen weergeven kan dit aangepast worden via een DataTemplate (zie later) of door de ToString() method van de class te overschrijven.

Master-detail scenario

Een master-detail scenario komt enorm veel voor. In zo een scenario komt het er op neer dat een bepaalde master – meestal een listbox of combobox – een lijst van items bevat en dat bij het selecteren van een item alle details van dit item getoond worden in labels of textboxen. Bij het selecteren van een ander item in de master wordt de detail automatisch aangepast. Dit kan met databinding als volgt gedaan worden:

```
<ListBox x:Name="Master" ItemsSource="{Binding Friends}" DisplayMemberPath="Name"/>
<TextBox Text="{Binding ElementName=Master, Path=SelectedItem.Firstname}"/>
```

De listbox heeft een naam gekregen die gebruikt wordt bij het binden aan de detail elementen. De detail elementen verwijzen naar de SelectedItem property van de listbox en daarna naar de property van het object Person dat moet weergegeven worden.

Over databinding met collecties en master-detail scenario's gaan we nog dieper in nadat ObservableCollections en de INotifyPropertyChanged interface aan bod zijn gekomen omdat we dan ook direct updates kunnen terugsturen.

DataTemplates

Momenteel is de data die zichtbaar is in de listbox heel erg beperkt. Zo is het bijvoorbeeld niet mogelijk om afbeeldingen of andere controls te gebruiken binnen een listboxitem. Om deze beperking op te lossen zijn er DataTemplates.

Een DataTemplate is een stuk XAML code dat zal beschrijven hoe de opmaak van elk listboxitem er precies uit moet zien. Bij elke element van de DataTemplate is het mogelijk om op te geven aan welke property dit precies gebind moet worden.

Het volgende voorbeeld toont de datatemplate voor de Person objecten die zowel de naam als de voornaam tonen en deze in een kleur zet afhankelijk van de score.

```
<DataTemplate x:Key="PersonTemplate">
    <StackPanel Background="{Binding Score, Converter={StaticResource ScoreToColorConverter}}">
        <TextBlock Text="{Binding Name}"/>
        <TextBlock Text="{Binding Firstname}"/>
    </StackPanel>
</DataTemplate>
```

Er wordt een key meegegeven om later de template te kunnen aanspreken. Daarna worden de verschillende elementen opgebouwd en wordt telkens aangegeven aan welke property ze gebind moeten worden. In dit voorbeeld wordt alles heel sober gehouden, maar het is mogelijk om heel complexe datatemplates uit te werken door een combinatie van verschillende elementen.

Bij de listbox zelf moet tot slot nog opgegeven worden dat de template met als key PersonTemplate gebruikt moet worden om de items weer te geven:

```
<ListBox x:Name="Master" ItemsSource="{Binding Friends}" ItemTemplate="{StaticResource PersonTemplate}"/>
```

Oefeningen databinding

Oefening 1

In deze oefening wordt data uit een XML bestand met data over verkeerscontroles weergegeven in een Listbox via databinding met een object. De listbox krijgt echter wel een volledige style via DataTemplates zodat direct duidelijk is waar het meeste controles plaats vinden en waar het meeste auto's geflitst worden. De XML file is online beschikbaar via http://data.drk.be/kortrijk/veiligheid_bemande_snelheidsmetingen_2012_wgs84.xml maar voor de performantie wordt er gekozen voor de offline versie.

stationsstraat februari	rijksweg augustus	stationsstraat september	rijksweg mei	rijksweg juli
kortrijksstraat december	kortrijksstraat december	kortrijksstraat oktober	kortrijksstraat november	heulsestraat februari
rijksweg februari	rijksweg maart	rijksweg april	rijksweg mei	rijksweg augustus
hulstsestraat december	rijksweg augustus	rijksweg september	tombroekstraat januari	tombroekstraat januari
tombroekstraat september	tombroekstraat oktober	tombroekstraat november	overzetweg februari	overzetweg november
markebekestraat april	markebekestraat augustus	markebekestraat september	markebekestraat oktober	markebekestraat december
roggelaan oktober	roggelaan december	munkendoornstraat mei	munkendoornstraat oktober	munkendoornstraat november
erasmuslaan februari	erasmuslaan maart	erasmuslaan april	erasmuslaan mei	erasmuslaan september

Figuur 10: labo 2 - oefening 1 eindresultaat

- Maak een nieuw WPFproject aan
- Kopieer de file SpeedControls.xml naar de bin/Debug map van je project
- Maak de volgende global resources als SolidColorBrushes
 - Orange: #FFE8E500
 - Blue: #FF3785B2
 - Gray: #FFB1B1B1
- Maak een map met als naam model aan in je project en maak onder die map een nieuwe class met als naam SpeedControl. De reden dat er gekozen wordt om dit in een map model te stoppen is dat dit ook zal toegepast worden wanneer we later met MVVM werken.
- Maak de volgende public properties aan in deze class:
 - Month (int)
 - Street (string)
 - ZipCode (string)

- City (string)
- NumberOfChecks (int)
- NumberOfVehicles (double)
- NumberOfViolations (double)
- ViolationsRate(double): dit is het percentage voertuigen dat werd geflitst.
- DataList (List<SpeedControl>)
- Binnen de getter van de property DataList wordt het XML document ingeladen. Kijk naar het code voorbeeld op <http://msdn.microsoft.com/en-us/library/dc0c9ekk.aspx> om te zien hoe je dit kan doen. Let wel: het root element in dit document heeft dezelfde naam als de childnodes (wat in feite niet zou mogen, maar dat kan gebeuren met data van online services) dus moet je de for loop vanaf 1 laten starten ipv 0.
- Maak binnen de for loop telkens een instantie aan van SpeedControl en vul de verschillende properties in. Return dan op het einde de list
- Maak in de file MainWindow.xaml.cs eens een nieuwe instantie aan van SpeedControl om te testen of de lijst correct wordt opgevuld.

Nu de data in orde is kunnen we verder naar de volgende stap, namelijk het ontwerpen van de user interface en het maken van de nodige databindings.

- Pas de titel aan van het Window en zorg er voor dat de WindowState op Maximized staat
- Binnen het grid maak je slechts één element aan, zijnde een ItemsControl. Je kan ook met een listbox werken, maar ItemsControl is een meer algemeen type en zal vermijden dat je de chrome van de listbox ziet wanneer de muis over een element beweegt of een element selecteert.
- Zorg nu voor de binding van de property DataList aan je ItemsControl en bekijk het resultaat. Welke resources zal je hier voor moeten toevoegen?
- Om er voor te zorgen dat de items niet allemaal onder elkaar staan passen we de template van het ItemsPanel aan. Plaats in deze template enkel een WrapPanel. Kijk op <http://msdn.microsoft.com/en-us/library/system.windows.controls.itemspaneltemplate.aspx> om te zien hoe je deze template aan kan passen en <http://msdn.microsoft.com/en-us/library/system.windows.controls.wrappanel.aspx> geeft meer info over de functie van het WrapPanel. Het resultaat zou moeten zijn dat alle items nu naast elkaar staan en dat er automatisch een nieuwe lijn wordt genomen.

De volgende stap in de oefening is er voor zorgen dat de template voor elk item wordt aangepast zodat niet langer de default ToString() method van het object wordt getoond.

- Maak een nieuwe DataTemplate aan en plaats daarbinnen een StackPanel met 2 textblocken. In de bovenste textblock komt de naam van de straat en in het onderste textblock de naam van de maand. Aangezien dat het object enkel een getal bevat voor de naam van de maand moet er een ValueConverter aangemaakt worden. Maak een ValueConverter zodat het cijfer wordt omgezet naar de overeenkomstige maand.
- Geef het StackPanel een breedte van 145px, een hoogte van 45px en een marge van 4px langs elke kant. Zet de achtergrond voorlopig op zwart
- Maak binnen de resources van je datatemplate een style aan voor de TextBlocken. Deze worden gecentreerd en hebben een witte tekstkleur.
- De zwarte achtergrond wordt vervangen door een blauwe of oranje achtergrond afhankelijk van het aantal controles. Als er maar 1 controle was krijgt de template een blauwe achtergrond, anders wordt het een oranje achtergrond. Het aantal controles is terug te vinden in de property NumberOfChecks, je moet natuurlijk wel opnieuw een ValueConverter aanmaken om dit getal om te zetten naar een SolidColorBrush.
- De opacity van het StackPanel hangt af van de property ViolationRate. Maak deze binding.
- Als de muis over het StackPanel beweegt moet de opacity echter opnieuw 1 worden. Maak hiervoor een Style aan voor het StackPanel met een IsMouseOver trigger die er voor zorgt dat de Opacity 1 zal worden.

In de laatste stap zal er een tooltip getoond worden als de muiscursor over het StackPanel beweegt. Deze tooltip zal alle detailinfo van een snelheidscontrole bevatten. Op

<http://msdn.microsoft.com/en-us/library/ms754034.aspx> kan je verdere uitleg vinden over hoe je een tooltip moet toekennen aan een element.

- Ken een tooltip toe aan het StackPanel.
- Plaats in deze tooltip een grid die labels bevat voor het aantal controles, het aantal gecontroleerde voertuigen en het aantal overtredingen.
- Maak een style voor de labels zodat deze een witte tekstkleur hebben en overal een padding van 2px.
- Maak een style voor de tooltip zodat deze een grijze achtergrond heeft en geen border.

Oefening 2

Deze oefening is gelijkaardig aan de eerste oefening. We zullen opnieuw een class maken die data ophaalt. Deze maal wordt de API van de NMBS gebruikt (<http://project.irail.be/wiki/API/APIv1>). Eerst zal een listbox alle stations weergeven en bij het klikken op een station worden alle vertrekkende treinen van uit dit station getoond (het liveboard) in een listview.

Time	Delay	Station	Vehicle	Platform
10:48 AM		Oostende [NMBS/SNCB]	BE.NMBS.IC831	5
10:50 AM		Poperinge [NMBS/SNCB]	BE.NMBS.P7017	4
10:50 AM		Antwerpen-Centraal [NMBS/SNCB]	BE.NMBS.IC732	3
10:53 AM		Antwerpen-Centraal [NMBS/SNCB]	BE.NMBS.IR3110	7
11:00 AM	+6	Blankenberge [NMBS/SNCB]	BE.NMBS.L661	6
11:05 AM		Zottegem [NMBS/SNCB]	BE.NMBS.L1661	1
11:15 AM		Sint-Niklaas [NMBS/SNCB]	BE.NMBS.IC2310	3
11:18 AM		Leuven [NMBS/SNCB]	BE.NMBS.IR4111	2
11:19 AM		Lille Flandres(f) [NMBS/SNCB]	BE.NMBS.IC19714	4
11:48 AM		Oostende [NMBS/SNCB]	BE.NMBS.IC832	5

Figuur 11: labo 2 - oefening 2 eindresultaat

- We maken gebruik van een theme om de opmaak te verzorgen. Voeg het theme uit het bronmateriaal toe aan je project en maak in App.xaml de link naar dit theme.
- Maak een map model aan in je project met daarin een class Station. Deze class zal alle informatie over de stations bevatten en heeft de volgende properties:
 - ID (string)
 - Name (string)
 - StationList (List<Station>)
 - Liveboard (List<Departure>)
- Maak ook een class Departure met de volgende properties:
 - Station (string)
 - Delay (string) Er wordt gekozen voor een string omdat dit veld ook de waarde "cancel" kan bevatten
 - Vehicle (string)
 - Time (DateTime)

- NormalPlatform (bool)
- Platform (int)
- In de class station vullen we de property StationList op met de data uit het gegeven XML document. Dit is gelijkaardig aan de eerste oefening, het verschil is wel dat je het ID uit een attribuut moet halen.
- Ook de property liveboard wordt opgevuld in de class Station. Deze data is afhankelijk van de naam van het station, dus check eerst of er wel een naam van een station is ingevuld.
- Daarna moet het XML document vanaf het volgende url worden ingeladen:
<http://api.irail.be/liveboard/?station=NaamVanHetStation&fast=true> waarbij je NaamVanhetStation moet vervangen door de uiteindelijk naam. Test bijvoorbeeld eens uit in je browser met Kortrijk of Antwerpen-Centraal, of Brussel-Centraal,... als naam.
- De rest is simultaan aan de eerste oefening. Haal alle nodes departures op en overloop deze met een lus. Maak voor elke node een nieuwe instantie aan van de class Departure.
- De tijd bevat een timestamp die moet omgezet worden naar een DateTime object. Zoek op hoe je die conversie kan uitvoeren (tip: <http://stackoverflow.com/questions/249760/how-to-convert-unix-timestamp-to-datetime-and-vice-versa>)
- Als het attribuut normal van het element platform op 1 staat krijgt de property NormalPlatform de waarde true, in alle andere gevallen staat de waarde op false.

Het model is nu klaar om gebruikt te worden in de grafische user interface. Eerst worden de nodige elementen op de user interface geplaatst.

- Maak 2 rijen en 2 kolommen in de grid. De bovenste rij is 100px en bevat een textblock met daarin de titel "NMBS Liveboard". Kies zelf een kleur uit de resources van de theme en kies ook een mooie font. Je kan dit in een style stoppen of gewoon de attributen invullen.
- De onderste rij bevat ook 2 kolommen. De linkse kolom is 200px breed en bevat een listbox met een margin van 8px langs elke zijde.
- Aan de rechterzijde staat er een listview. Een listview is ook een ItemsControl net als een Listbox, maar heeft veel meer mogelijkheden naar groeperen, sorteren, filteren en input toe. Alle info over de listview kan je terugvinden op <http://msdn.microsoft.com/en-us/library/system.windows.controls.listview.aspx>. De listview zal later opgevuld worden.
- Test of het ophalen van de data werkt door een nieuwe instantie van de class Station aan te maken in XAML. Vul wel een naam van een station in om het Liveboard te testen.

Nu de user interface min of meer klaar is kan de data opgevuld worden.

- Eerst wordt de listbox opgevuld met de lijst van stations. Er wordt geen afzonderlijke template aangemaakt, aangezien enkel de naam van het station moet ingevuld worden. Gebruik hier het attribuut DisplayMemberPath.
- Bij het selecteren van een station moet de listview ingevuld worden met de data uit de property Liveboard. Maak hiervoor een binding tussen de ItemsSource van de listview en de property Liveboard van het geselecteerde item in de listbox (Tip: dit is binding tussen 2 UI elementen).

Test de applicatie uit. Normaal gezien moet er nu bij het selecteren van een station data in de listview komen. Indien dit een error geeft kan dit zijn omdat een platform niet is ingevuld in de XML file. Je kan dit opvangen in het model (tip: HasChildNodes), maar dit heeft geen prioriteit.

- De Listview zal een GridView bevatten. Kijk op <http://msdn.microsoft.com/en-us/library/system.windows.controls.gridview.aspx> om een voorbeeld te vinden hoe je dit kan toepassen. Er moet een kolom voorzien worden voor Time, Delay, Station, Vehicle en Platform.
- Zorg er voor dat de tijd wordt weergegeven als HH:MM. Je kan dit doen door het StringFormat attribuut van de binding op de waarde t te plaatsen. Dus: **StringFormat=t**
- De delay wordt weergegeven in seconden. Schrijf een converter om dit om te zetten naar minuten en plaats er een plusteken voor. In het geval dat een trein geannuleerd werd plaats je een hoofdletter C.
- Het station en vehicle worden gewoon als tekst in de listview getoond.

- Voor het platform moet je de CellTemplate aanpassen van de GridViewColumn. Je plaatst in deze template een Textblock en de tekst zal gewoon het nummer van het platform bevatten. Je maakt echter ook een ValueConverter om de achtergrond in het geel te plaatsen als de property NormalPlatform op false staat. Tip: plaats om te testen eens de property NormalPlatform default op false omdat het maar zelden voorkomt dat die op false staat in de API. Om een element geen achtergrondkleur te geven gebruik je: `new SolidColorBrush(Colors.Transparent)`
- Tot slot wordt nog de ItemContainerStyle aangepast van de listview. Deze style zal alle items 24px hoog maken, een lichte achtergrondkleur instellen (mag je zelf kiezen) en de tekstkleur laten afhangen van het feit of er al niet niet vertraging is. Als er geen vertraging is staat de tekst in het zwart, anders in het rood. Je zal hier opnieuw een ValueConverter voor moeten schrijven.

ObservableCollection

Intro

Tot nu toe werd een `List<T>` gebruikt om collections in op te slaan. Met de komst van WPF is er echter een type dat meer mogelijkheden biedt voor collections, namelijk de `ObservableCollection`. Zoals de naam doet vermoeden is het met dit type mogelijk om wijzigingen in de data van de collectie te observeren en daar op te anticiperen. Dit is dus een heel krachtig mechanisme bij databinding.

INotifyPropertyChanged

Deze interface zorgt er voor dat een property van een object een melding kan sturen dat de waarde werd gewijzigd. Bijvoorbeeld: een object Person kan laten weten dat de property JobRole werd aangepast. Deze aanpassing wordt dan automatisch in de user interface getoond.

Bij het gebruik van een `List<T>` zou deze melding ook naar het model gestuurd worden. Er zal echter geen update gebeuren in de user interface.

Alle info over deze interface is te vinden op: <http://msdn.microsoft.com/en-us/library/system.componentmodel.inotifypropertychanged.aspx>. In de praktijk komt het er op neer dat het implementeren van deze interface er voor zal zorgen dat er een event `PropertyChanged` wordt aangemaakt dat moet afgewuurd worden telkens een property aangepast is. Dit wordt met een method `OnPropertyChanged` gedaan:

```
private string _jobrole;
public string JobRole
{
    get { return _jobrole; }
    set {
        if (_jobrole != value)
        {
            _jobrole = value;
            OnPropertyChanged("JobRole");
        }
    }
}
```

Binnen de setter van de property `JobRole` wordt er eerst gecontroleerd of er wel een nieuwe value werd ingevuld om een overbodige uitvoering van de setter te voorkomen. Daarna komt de nieuwe waarde in het private field `_jobrole` en wordt een method `OnPropertyChanged` aangeroepen met de naam van de property als parameter.

De method `PropertyChanged` ziet er als volgt uit:

```
private void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Er wordt gecontroleerd of het event verschillend is van null en in dat geval wordt het event afgeweerd met de naam van de property als event argument.

Om te voorkomen dat elke klasse dit moet implementeren is het mogelijk om een base class aan te maken met deze functionaliteit. In het labo wordt dit principe in een oefening aangetoond.

INotifyCollectionChanged

Deze interface is gelijkaardig aan INotifyPropertyChanged. Het enige verschil is dat deze wordt toegepast op collecties en enkel een melding zal geven als een item aan de collectie werd toegevoegd of verwijderd. Er zal dus geen update van de user interface komen als een property van een element uit de collectie wijzigt. Daarvoor dient de INotifyPropertyChanged interface.

De INotifyCollectionChanged interface is automatisch geïmplementeerd in de ObservableCollection. Daarom geniet dit type collectie de voorkeur geniet op gelijk welk ander type.

Eerste demo

Deze demo toont een collectie van Employee objecten in een ObservableCollection. Op de JobRole property van de Employee class wordt de method OnPropertyChanged opgeroepen. Bij de Name property wordt dit niet gedaan.

```
public string Name { get; set; }

private string _jobrole;
public string JobRole
{
    get { return _jobrole; }
    set {
        if (_jobrole != value)
        {
            _jobrole = value;
            OnPropertyChanged("JobRole");
        }
    }
}
```

In MainWindow.xaml.cs wordt een ObservableCollection aangemaakt van Employee objecten en worden enkele elementen aan de collectie toegevoegd. Tot slot wordt de collection toegekend aan de ItemsSource property van een listbox (of gelijk welke andere ItemsControl).

```
Employees.Add(new Employee() { Name = "Jan Jansens", JobRole = "Manager" });
Employees.Add(new Employee() { Name = "Peter Peeters", JobRole = "Software ..." });
Employees.Add(new Employee() { Name = "Ann Dewaele", JobRole = "Software ... " });
EmployeeList.ItemsSource = Employees;
```

Op de user interface worden nog 2 buttons geplaatst. De ene button zal een item aan de collection toevoegen, de andere zal het geselecteerde item aanpassen.

```
// item toevoegen
Employees.Add(new Employee() { Name = "Stefaan Vandecasteele", JobRole = "Sales" });
```



```
// item aanpassen
Employee selected = EmployeeList.SelectedItem as Employee;
selected.JobRole = "Helpdesk";
```

Bij het testen van de demo zal bij het klikken op de add button direct een nieuwe lijn worden toegevoegd, bij het klikken op de update button zal het geselecteerde item aangepast worden in de UI.

Mocht de ObservableCollection vervangen worden door een List<Employee> zal het toevoegen van nieuwe records niet zichtbaar zijn. Als de INotifyPropertyChanged interface geïmplementeerd is zal de update wel werken.

Commands en Routed Events

Intro

WPF biedt veel meer mogelijkheden via routed events om events af te handelen, ook al zijn deze mogelijkheden niet direct zichtbaar. Je hebt immers al routed events gebruikt telkens je een eventhandler gekoppeld hebt aan bijvoorbeeld een Button.

Daarnaast zijn er in WPF ook nog commands die zorgen voor een betere scheiding tussen de user interface en de business logic. Daarom zijn ze ook onontbeerlijk bij het gebruik van het MVVM pattern. De volgende secties tonen de mogelijkheden van deze routed events en commands.

Logical tree en Visual tree

Deze begrippen zijn belangrijk om te kunnen werken met routed events. **De logical tree** is de boomstructuur die je in je XAML ziet staan. Als root element is er altijd een Window en dat window heeft altijd 1 child element – dat standaard een grid is. Dat grid kan ook child elementen bevatten en deze elementen zullen het grid als parent hebben. Kortom: net als HTML bestaat XAML uit een boomstructuur en die wordt de logical tree genoemd.

De visual tree is heel gelijkaardig aan de logical tree, uitgezonderd dat deze ook alle interne elementen van een control zal bevatten. Bijvoorbeeld: een button bestaat uit een Border element en een ContentPresenter. Deze verdere verfijning zal je niet terugvinden in de logical tree, maar enkel in de visual tree. Via de VisualTreeHelper (<http://msdn.microsoft.com/en-us/library/system.windows.media.visualtreehelper.aspx>) kan je door de volledige visual tree van een control lopen. De link bevat een code sample hoe dit precies kan doen.

Routed Events

Een routed event is een event dat navigeert door de visual tree. Het beste voorbeeld is een click event op een button. Bijvoorbeeld een button met de volgende structuur:

```
<Button Background="Blue" Margin="20" Click="Button_Click">
    <StackPanel Background="Red" Margin="20" MouseDown="StackPanel_MouseDown">
        <Rectangle ... Margin="20" MouseDown="Rectangle_MouseDown"/>
        <Rectangle ... Margin="20" MouseDown="Rectangle_MouseDown"/>
        <Rectangle ... Margin="20" MouseDown="Rectangle_MouseDown"/>
    </StackPanel>
</Button>
```

Het root element is hier de Button, met 1 child element zijnde een StackPanel. Dit stackpanel heeft 3 child elementen, namelijk de rechthoeken. Bij de button wordt het click event opgevangen, bij de andere elementen het MouseDown event. Bij het schrijven van een boodschap "geklikt op ..." in de console bij elk van deze eventhandlers zal een click op een rechthoek het volgende resultaat opleveren:

klik op rectangle
klik op stackpanel
klik op button

Bij een klik op het blauwe vlak van de knop zelf krijg je het volgende resultaat:

klik op button

Het komt er dus op neer dat een mousedown op een child element van de button automatisch ook het click event van de button zal oproepen. Dit noemt event **bubbling**. Dit is een goede zaak, anders zou je als developer enorm veel werk hebben om alle mogelijke events van alle mogelijke child elementen op te vangen. Als je dus enkel het click event van de button wil opvangen hoef je niet te luisteren naar dit event van de child elementen.

Dit principe kan ook zijn nadelen hebben. Bijvoorbeeld als er actie moet uitgevoerd worden bij het klikken op een rechthoek, maar als op dat ogenblik de click van de button niet mag uitgevoerd worden. Je kan het bubbelen van events stoppen door e.Handled = `true`; aan te roepen.

Naast event bubbling is er ook nog **tunneling**. Dit principe volgt de omgekeerde weg. In plaats van een event van een child element naar de parents te sturen, worden events van de parents naar de child elementen gestuurd. Hiervoor heeft WPF de verschillende previewevents zoals PreviewMouseDown. Bij het luisteren naar dit event in het voorbeeld krijgen we volgend resultaat bij het klikken op een rechthoek:

```
Tunnel: klik op button
Tunnel: klik op stackpanel
Tunnel: klik op rectangle
Bubble: klik op rectangle
Bubble: klik op stackpanel
Bubble: klik op button
```

Eerst worden de preview events uitgevoerd van de parent naar het child. Daarna wordt event bubbling uitgevoerd van het child naar de parent. Ook hier kan via e.Handled = `true`; de navigatie gestopt worden. Let wel: dit zal zowel het tunneling als bubbling proces stoppen.

Tot slot zijn er ook nog de **direct events**. Deze worden enkel uitgevoerd op het element zelf en hebben geen tunneling of bubbling proces. Verdere informatie over alles wat met routed events te maken heeft is terug te vinden op <http://msdn.microsoft.com/en-us/library/ms742806.aspx>.

Commands

Naast de routed events zijn er in WPF ook nog commands. (Routed) commands hebben de volgende voordelen ten opzichte van (routed) events:

- Er is minder koppeling tussen de control die de event oproept en de code die wordt uitgevoerd.
- Commands kunnen automatisch UI elementen enablen en disablen
- Een command kan aan verschillende acties gekoppeld worden zoals het gebruik van een shortcut en een klik op een element.

WPF kent een ganse boel ingebouwde commands die gebruikt kunnen worden. Deze zijn verdeeld in verschillende categorieën zoals ApplicationCommands, MediaCommands, NavigationCommands, ... Het volledige overzicht is terug te vinden op <http://msdn.microsoft.com/en-us/library/ms752308.aspx>. Daarnaast is het ook nog mogelijk om eigen commands aan te maken, maar dit komt momenteel niet aan bod in deze cursus.

Terminologie

Er zijn vier belangrijke termen als het op het gebruik van commands aankomt in WPF. Dit zijn:

- Command: de actie die uitgevoerd moet worden
- Command source: het object dat de actie zal oproepen
- Command target: het object waar de actie zal op uitgevoerd worden
- Command binding: het binden van een bepaalde method aan een command

Demo

De volgende demo toont hoe het command ApplicationCommands.Save kan gebruikt worden om vast te maken aan een button. Er wordt vertrokken vanaf het volgende stuk XAML:

```
<StackPanel>
    <Label>Naam</Label>
    <TextBox x:Name="Naam"/>
```

```
<Label>Voornaam:</Label>
<TextBox x:Name="Voornaam"/>
<Button Content="Save" Command="ApplicationCommands.Save"/>
</StackPanel>
```

Merk dat de button standaard disabled is. Dit komt omdat er een voorwaarde moet opgegeven worden wanneer de button al dan niet enabled mag zijn. Dit wordt gedaan in de command binding:

```
<Window.CommandBindings>
    <CommandBinding Command="...Save" CanExecute="..." Executed="..."/>
</Window.CommandBindings>
```

Deze binding geeft aan voor welk command de binding is en welke methods moeten uitgevoerd worden om te testen of de source (in dit geval de button) al dan niet enabled mag zijn en welke method moet uitgevoerd worden. In de code behind worden deze methods dan ingevuld:

```
private void CommandBinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = EnableDisableControls();
}

private void CommandBinding_Executed(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("saved the data");
}
```

De property CanExecute zal via true of false aangeven of de control al dan niet enabled is. Bij het klikken op de knop zal de messagebox getoond worden. Merk op dat de shortcut Ctrl+S ook zal werken. Mocht er een menuitem zijn met hetzelfde command zal dat item automatisch de werking van de button overnemen.

```
<Menu IsMainMenu="True">
    <MenuItem Header="Save" Command="ApplicationCommands.Save" />
</Menu>
```

Dit hoofdstuk was slechts een korte inleiding op commands, maar een kennis van het begrip command is wel noodzakelijk voor het vervolg van de cursus.

Oefeningen ObservableCollection, Commands en Routed Events

Oefening 1

Deze oefening geeft alle winkels uit Kortrijk terug in een Datagrid. De uitleg over hoe je met eendatagrid kan werken is te vinden op <http://msdn.microsoft.com/en-us/library/system.windows.controls.datagrid.aspx>. Je kan styling uitvoeren op dedatagrid, maar in deze oefening beperken we ons tot het weergeven en bewerken van data.

Boven dedatagrid is er een klein menu voorzien. Via dit menu is het mogelijk om een item uit de lijst te verwijderen of om de ganse lijst weg te schrijven in het output venster. De mogelijkheid is voorzien om te controleren of de data source wel degelijk werd aangepast.

Naam	Adres	Postcode	Stad
REDDAHI	Loodwitstraat 2	8500	Kortrijk
EUROCITY	Waterpoort 2	8500	Kortrijk
Firma Antoon Dewulf en zoon	Meensesteenweg 98	8501	Bissegem
Arub Indian Food Store	Overleiestraat 33	8500	Kortrijk
WAHAB SUPERMARKET	Zwevegemsestraat 14	8500	Kortrijk
OKAY BISSEGEM	Meensesteenweg 210 A	8501	Bissegem
Kruidvat Kuurne	Ringlaan 34 330	8500	Kortrijk
Proxy Delhaize	Steenpoort 2	8500	Kortrijk
ALDI MARKT	Gullegemsestraat 22	8501	Heule
Vanhoutteghem, Patrick	Abdijmolenweg 44	8510	Marke
Vanhalst, Micheline	Beheerstraat 2 A	8500	Kortrijk
Biotheek Vanneste	Hektor Casteleinstraat 43	8510	Marke
Schoutetens, Luc	Parkietenlaan 27	8500	Kortrijk

Figuur 12: labo 3 - oefening 1 eindresultaat

- Maak een map model aan met daarin een class BaseModel.cs. Implementeer de INotifyPropertyChanged interface in deze class en schrijf een method OnPropertyChanged die het event van de interface zal oproepen (zie theorie). Deze class zal gebruikt worden zodat niet voor elke andere class de interface moet geïmplementeerd worden. We kunnen immers gewoon overerven van BaseModel.cs.
- Maak een class Shop.cs aan. Deze class erft over van BaseModel en heeft de volgende properties:
 - Name (string)
 - Address (string)
 - ZipCode (string)
 - City (string)
 - Shops (ObservableCollection<Shop>)
- Vergeet niet om in elke setter de method OnPropertyChanged voor de property op te roepen.
- Maak in de class Shop een method GetShops() die de property Shops zal opvullen aan de hand van het gegeven XML document.
- Overschrijf de ToString() method zodat alle properties mooi naast elkaar getoond worden.
- Tot slot schrijf je een method PrintShops() die alle elementen uit de property Shops overloopt en in de console afdrukt. Hiervoor gebruik je de overschreven ToString() method.

Nu het model klaar is kan de grafische user interface opgebouwd worden. Deze beperkt zich tot een menu en een datagrid.

- Maak 2 rijen in het grid. De eerste rij is 32px hoog en zal het menu bevatten. Hetdatagrid komt in het resterende gedeelte.
- Zie theorie hoe je een menu kan aanmaken
- Hetdatagrid maak je als volgt aan:

```
<DataGrid AutoGenerateColumns="False" Grid.Row="1">
    <DataGrid.Columns>
        <DataGridTextColumn Header="Naam" />
        <DataGridTextColumn Header="Adres" />
        <DataGridTextColumn Header="Postcode"/>
        <DataGridTextColumn Header="Stad"/>
    </DataGrid.Columns>
</DataGrid>
```

- De property AutoGenerateColumns staat op false omdat anders ook een kolom voorzien zal worden voor de property Shop wat hier niet nodig is. We beperken ons hier tot DataGridViewTextBoxColumn elementen, maar ook andere elementen zijn mogelijk.
- Geef het parent grid een naam, bijvoorbeeld Root.
- Maak in je code behind file een nieuwe instantie aan van de Shop class in de constructor. Roep daarna de method GetShops() op en stel de datacontext van je root grid in op de instantie van de class Shop. Op die manier kunnen we aan databinding doen en hoeft er verder geen code hiervoor in de code behind geschreven te worden.
- Stel de property Shops in als ItemsSource voor het datagrid. In principe moet je hier als mode TwoWay binding instellen, maar dit mag weggeleggen worden aangezien dit de default waarde is voor de property ItemsSource van een datagrid.
- Stel nu per kolom ook de property binding in voor de juiste property en test of je alle data kan weergeven.

Bij het testen van de applicatie zie je normaal alle data staan en kan je ook onderaan items toevoegen en de bestaande items wijzigen door er in te typen. Als je naar een andere rij navigeert wordt dit item in de property Shops automatisch aangepast. Je kan dit testen door een breakpoint in te stellen op de setter van de property die je aanpast.

In de laatste stap wordt er code voorzien om een item te kunnen verwijderen en om de volledige lijst in de console af te drukken.

- Stel het ApplicationCommands Delete in op het menuitem om te verwijderen en het ApplicationCommands Save op het item om de lijst af te printen.
- Na het instellen van het Delete command zal je merken dat het menuitem beschikbaar wordt zodra er een rij geselecteerd is en dat de rij wordt verwijderd bij het klikken op het element. Dit toont de grote kracht van commands aan.
- Voor het Save command moet je wel nog de commandbinding maken. Vul een method name in bij de Executed property. Het menuitem mag altijd enabled zijn, je hoeft dus de property CanExecute niet in te vullen.
- Roep tot slot in de aangemaakte method de method PrintShops() op van de datacontext.

Oefening 2

In deze oefening worden alle vuilnisbakjes die langs de straat te vinden zijn in Kortrijk op een kaart weergegeven en is het mogelijk om deze te bewerken, te verwijderen of er toe te voegen. In feite is de kaart één grote listbox met verschillende listboxitems.

Er wordt in deze oefening gebruikt gemaakt van Bing Maps – wat de Microsoft variant is van Google maps. Er bestaat een control om deze functionaliteit te gebruiken in WPF. Alle informatie over deze control en de bijhorende SDK is te vinden op: <http://msdn.microsoft.com/en-us/library/hh750210.aspx>.

Je zal echter wel een developer key moeten aanvragen om deze functionaliteit te kunnen gebruiken. Een key aanvragen kan via <https://www.bingmapsportal.com/>.



Figuur 13: labo 3 - oefening 2 eindresultaat

- Er is een start project gegeven voor deze oefening. Dit startproject bevat de volgende zaken:
 - Enkele brushes in App.xaml
 - Een referentie naar de Bing maps control
 - Het volledige model (3 classes)
- Bekijk het model eens aandachtig. Er is opnieuw de class BaseModel zoals in de eerste oefening. Daarnaast is er ook een class Wastebin die verschillende properties bevat. Een van de properties is LocationType waar ook een class voor aangemaakt is in het model.
- Er is een method GetWastebins() die alle data uit een csv file zal ophalen en verwerken. Het verwerken vraagt nog redelijk veel controles aangezien de structuur van de csv file niet overal even consequent is bij het invullen van de GPS-coördinaten.
- Er wordt ook gebruik gemaakt van een type Location. Dit type komt mee met de Map control en bevat de longitude en latitude als properties. Alle info is te vinden op <http://msdn.microsoft.com/en-us/library/microsoft.maps.mapcontrol.wpf.location.aspx>
- Plaats een map control op je MainWindow. Deze control mag het volledige Window voor zijn rekening nemen. Kijk op <http://msdn.microsoft.com/en-us/library/hh830433.aspx> hoe je dit kan doen. Stel je key, Center en ZoomLevel in. Het center plaats je op Kortrijk en voor het ZoomLevel zijn 13,14 of 15 goede waarden.

Test je applicatie. Je zal de map control zien waar je kan zoomen en over de kaart kan navigeren. In de volgende stap worden alle vuilnisbakjes op de kaart getekend.

- Maak in de constructor van je code behind file een instantie aan van de class Wastebin, roep de method GetWastebins() op en stel de instantie in als DataContext van je root element.
- Voeg een MapItemsControl toe binnen je map control. Deze zal als een ItemsControl dienen waar een ItemsSource en een ItemTemplate kan op ingesteld worden. Alle info over de MapItemsControl is te vinden op <http://msdn.microsoft.com/en-us/library/microsoft.maps.mapcontrol.wpf.mapitemscontrol.aspx>
- Bind de ItemsSource property aan de property Wastebins van de datacontext van het root element.

Maak een ItemTemplate aan voor de MapItemsControl. Je begint met een DataTemplate element en daar binnen plaats je een ContentControl element. De positie stel je in op de GeoLocation

property van de Wastebin instantie. Gebruik hiervoor `map:MapLayer.Position="{Binding GeoLocation}"` als attribuut bij de ContentControl.

- Plaats binnen de ContentControl een button en bind deze aan de barcode property. Test de applicatie eens uit en normaal gezien moet er voor elke vuilnisbak een button op de map getekend worden.

In de volgende stap wordt de button vervangen door een formulier dat alle data weergeeft. Bouw de volgende template na:

Figuur 14: labo 3 oefening2 datatemplate

- Het root element is een stackpanel met daarin een textblock dat de barcode bevat. Daaronder zit opnieuw een stackpanel met vervolgens:
 - Een button
 - Label voor adres
 - Textbox voor adres
 - Label voor location type
 - Combobox
 - Label voor capaciteit
 - Slider (Minimum=0, Maximum=100, stap per10 en label laten tonen)
- Stel de nodige bindings in. Voor een combobox zijn die nogal complex. De ItemsSource zal een collectie van alle LocationTypes zijn. Deze zijn terug te vinden in het model
- Naast de property SelectedItem moet ook nog de SelectedValue en SelectedValuePath ingesteld worden. Kijk op <http://msdn.microsoft.com/en-us/library/system.windows.controls.combobox.aspx> wat deze properties precies doen en hoe je ze moet gebruiken.
- Om een item te verwijderen is het voldoende om een click event te koppelen aan de button. Het is ook mogelijk om met een command te werken, maar dit heeft hier in feite geen enkele meerwaarde. In de eventhandler verwijder je het item uit de lijst van de datacontext van het root element.

Het probleem is momenteel wel dat de elementen enorm over elkaar hangen en dat dit moeilijk werken is. Daarom wordt het binnenste stackpanel verborgen tot de muis over de ContentControl beweegt. Hier wordt een trigger voor gebruikt.

- Zet de Visibility van het binnenste stackpanel op Collapsed
- Maak een trigger aan voor IsMouseOver op de template en plaats binnen die trigger de visibility van het stackpanel opnieuw op Visible. Je zal wel de property TargetName moeten gebruiken. Een voorbeeld hiervan is te vinden op <http://msdn.microsoft.com/en-us/library/system.windows.setter.targetname.aspx>
- Zet ook de Zindex van de template op 9999 zodat het geselecteerde item altijd volledig bovenaan zit.
- Tot slot moet een nieuw element aan de collectie toegevoegd worden bij het dubbel klikken op de map. Kijk op <http://msdn.microsoft.com/en-us/library/hh709044.aspx> hoe je dit kan

doen en voeg dan een nieuwe instantie van WasteBin toe aan de collectie. Je hoeft enkel een timestamp toe te voegen voor de property barcode en de coördinaten in de property GeoLocation.

Databases ontwerpen

Intro

Het ontwerp van een database is een belangrijke stap bij elk project en dient als de basis voor je uiteindelijk object model waarop je de applicatie zal bouwen. Een wijziging in het database model kan dus serieuze implicaties hebben in alle andere lagen van je applicatie.

Er zijn 3 verschillende manieren hoe een database ontworpen kan worden:

- De requirements van een project bekijken om een volledig nieuwe database te ontwerpen
- Analyseren van bestaande gegevens uit Excel, PDF, XML, ... documenten
- Het hervormen van een bestaande database

Relationele database

Definitie

Een database is niets meer of minder dan een verzameling van gegevens. In dat opzicht is een collectie van XML documenten of Excel files ook een database. Binnen deze cursus ligt de focus echter op het gebruik van relationele databases die ondervraagd kunnen worden aan de hand van SQL statements.

Volgens Wikipedia (http://en.wikipedia.org/wiki/Relational_database) is een relationele database een verzameling van gegevens die gestructureerd zijn in tabellen. Tussen de verschillende tabellen is er een relatie aanwezig.

Een typisch voorbeeld is het opslaan van gegevens over producten. Er is een tabel ProductCategorie die algemene info over de categorie bevat en een tabel Product met specifieke info over dat product. Tussen beide tabellen is er een relatie.

Een relationele database zal gebruik maken van een Relational Database Management System (RDBMS). Binnen deze cursus maken we gebruik van MS SQL Server 2012, maar er zijn ook nog andere systemen zoals MySQL.

Terminologie

Een **entity** is een uniek identificeerbaar iets waarover we informatie wensen bij te houden (bijvoorbeeld: een klant, een product, een bestelling,...) en bestaat uit verschillende unieke **attributes**. Elk attribute bevat een stuk specifieke informatie over de entity. Een entity persoon kan dus de attributes naam, voornaam, rijksregisternummer,... bevatten. Een **record** bevat dan de uiteindelijke waarde van een bepaald attribute binnen een bepaalde entity.

Kortweg:

- Entity = tabel
- Attribute = kolom
- Record = rij

Bij een relationele database is er een **relatie** tussen de verschillende tabellen. Deze relatie wordt gevormd door **de primary key** en foreign key van de betrokken tabellen. De primary key van een entity is een attribute of een verzameling van attributes dat er voor zorgt dat elk record binnen de entity uniek is. De **foreign key** is dan een attribute of een verzameling van attributes die verwijst naar de primary key van een andere entity.

Database normalisatie

Het organiseren van de data in entiteiten en attributen met een minimum aan redundantie wordt normalisatie genoemd. Dit is een belangrijk concept binnen het ontwerpen van databases. Een voorbeeld is volgende situatie waar gegevens over studenten en hun cursus wordt bijgehouden:

Tabel: Student				
Naam	Voornaam	Cursus	Studiepunten	Campus
Devolder	Jochen	Frans	6	GKG
Desmet	Stijn	Frans	6	GKG
Vanneste	Thomas	Engels	9	RDR

Als Thomas Vanneste zich uitschrijft voor de cursus Engels gaat niet enkel de informatie van de student verloren, maar ook de informatie over de cursus. Dergelijke situatie is een anomalie en mag nooit voorkomen in een database model. Via normalisatie kan dit vermeden worden.

Er zijn 5 normaalvormen (NF: normal form) die elkaar opvolgen. 2NF is dus 1NF + extra vereisten, 3NF is dan 2NF + extra vereisten,... Binnen deze cursus gaan we tot de 3^{de} normaalvorm aangezien deze volstaat voor de meeste situaties. Elke normaalvorm bestaat uit een set van regels.

1NF: eerste normaalvorm

Duplicateer geen data in dezelfde record van een entiteit (atomicity).

Een klassiek voorbeeld is een lijst met contactgegevens waarbij een persoon meerdere e-mailadressen heeft.

Tabel: Persoon		
Naam	Email1	Email2
Paul Peeters	Paul.peeters@howest.be	Paul.peeters@telenet.be
Chris Devos	Chris007@hotmail.com	
Karel Vandevelde	karl@gmail.com	Karl.vdv@skynet.be

Als een nieuwe persoon 3 e-mailadressen heeft moet er een extra veld aangemaakt worden in de tabel. Dit heeft serieuze implicaties en moet te allen tijde vermeden worden. In dit geval is het een oplossing om een nieuw record te maken voor elk e-mailadres.

Tabel: Persoon	
Naam	Email
Paul Peeters	Paul.peeters@howest.be
Paul Peeters	Paul.peeters@telenet.be
Chris Devos	Chris007@hotmail.com
...	

Elk record moet een attribute of verzameling van attributes bevatten die het record uniek maken.

Kort samengevat komt het er op neer dat elke entiteit een primary key moet hebben. Dit kan een bestaand veld zijn zoals het riksregisternummer van een persoon of de barcode van een product. Dit kan ook een autonummer zijn die door het RDBMS wordt toegekend.

De tabel persoon kan er dan als volgt uitzien:

Tabel: Persoon		
ID (PK)	Naam	Email
1	Paul Peeters	Paul.peeters@howest.be
2	Paul Peeters	Paul.peeters@telenet.be

2NF: tweede normaalvorm

Plaats subsets van data die veel voorkomen in meerdere rijen naar een afzonderlijke tabel

Een typisch voorbeeld is het bijhouden van adresinformatie. Op meerdere rijen zal je telkens dezelfde combinatie van postcode en gemeente terugvinden

Tabel: Adres				
ID (PK)	Straat	Nummer	Postcode	Gemeente
1	Graaf Karel de Goedelaan	5	8500	Kortrijk
2	Molenstraat	36	8501	Heule
3	Renaat de Rudderlaan	7	8500	Kortrijk
4	Sint Jorisstraat	25	8000	Brugge
5	Rijselstraat	26	8000	Brugge

Zoals je kan zien bevat de tabel bij verschillende rijen dezelfde combinatie van postcode en gemeente. Dit is ook logisch aangezien deze aan elkaar gerelateerd zijn. De tweede normaalvorm eist dus dat deze in een afzonderlijke tabel komen.

Opmerking: je ziet dat het nummer in een afzonderlijke kolom is opgeslagen. Dit is om te voldoen aan atomicity van de eerste normaalvorm die stelt dat elk attribuut slechts 1 waarde mag bevatten. Punt van discussie kan hier uiteraard zijn of het huisnummer en de straatnaam al dan niet één geheel vormen.

Een ander voorbeeld om dit principe aan te tonen is het opslaan van gegevens over producten en hun categorie.

Tabel: Product			
ID (PK)	Naam	Prijs	Categorie
1	Xbox one	399	Spelconsoles
2	HDMI kabel	50	Bekabeling
3	Playstation 4	450	Spelconsoles
4	VGA kabel	25	Bekabeling
5	Toestenbord	25	Accessoires

De kolom categorie bevat ook voor verschillende rijen dezelfde waarde. Deze moet volgens de tweede normaalvorm dus ook in een afzonderlijke tabel komen.

Maak een relatie tussen deze tabellen door gebruik te maken van foreign keys

Deze regel is een gevolg van de eerste regel. Door gebruik te maken van een foreign key zorg je er voor dat je later gemakkelijk relaties tussen de beide tabellen kan leggen.

Het resultaat van het eerste voorbeeld zal zijn

Tabel: Adres			
ID (PK)	Straat	Nummer	GemeenteID
1	Graaf Karel de Goedelaan	5	1

Tabel: Gemeente		
GemeenteID (PK)	Postcode	Gemeentenaam
1	8500	Kortrijk

Opmerking: de postcode kan hier niet als primary key gekozen worden aangezien bijvoorbeeld Bavikhove en Hulste allebei postcode 8531 hebben. Ook Marke en Bissegem hebben allebei postcode 8501.

Het resultaat van het tweede voorbeeld wordt dan:

Tabel: Product			
ID (PK)	Naam	Prijs	CategorieID
1	Xbox one	399	1

Tabel: Productcategorie	
CategorieID (PK)	Omschrijving
1	Spelconsoles

3NF: derde normaalvorm

Verwijder attributen die niet volledig afhankelijk zijn van de primary key

Deze regel zorgt er voor dat elke entiteit enkel informatie over de entiteit zelf kan bevatten. Als we het eerste voorbeeld van de normalisatie opnieuw bekijken merken we dat deze entiteit informatie bevat over zowel studenten als cursussen. Dit kon al opgelost worden in de tweede normaalvorm aangezien er veel informatie in verschillende rijen wordt herhaald.

Tabel: Student					
ID (PK)	Naam	Voornaam	Cursus	Studiepunten	Campus
1	Devolder	Jochen	Frans	6	GKG
2	Desmet	Stijn	Frans	6	GKG
3	Vanneste	Thomas	Engels	9	RDR

Mocht dit nog niet ontdekt zijn in de tweede normaalvorm moet dit opgelost worden bij de regels in de derde normaalvorm. De attributen cursus, studiepunten en campus hebben immers niets te maken met de entiteit student en moeten dus naar een afzonderlijke tabel. Het resultaat wordt dus

Tabel: Student

ID (PK)	Naam	Voornaam	CursusID
1	Devolder	Jochen	1

Tabel: Cursus

CursusID (PK)	Naam	Studiepunten	CampusID
1	Frans	6	1

Tabel: Campus

CampusID (PK)	Omschrijving
1	GKG

Er wordt dus een afzonderlijke tabel gemaakt met alle info voor de student, maar de overige velden worden ook nog gesplitst in 2 afzonderlijke tabellen. De informatie over de campus heeft immers niets te maken met de informatie over een cursus.

Een ander voorbeeld zijn attributen die berekende informatie bevatten. Een typisch voorbeeld is hier het berekenen van de BTW.

Tabel: Product

ID (PK)	Product	Netto	BTW	Bruto
1	Xbox One	300	21	363
2	Kabel	100	21	121

Het veld bruto heeft niets te maken met het product en hoeft dus niet opgeslagen te worden in de database. Merk wel op dat sommige berekeningen zodanig complex zijn dat het performanter is om de resultaten op te slaan in de database in plaats van ze telkens opnieuw te laten berekenen.

Conclusie van normalisatie is dan ook dat dit duidelijke richtlijnen zijn om je database model te ontwerpen, maar ze zijn echter geen exacte wetenschap.

Soorten relaties

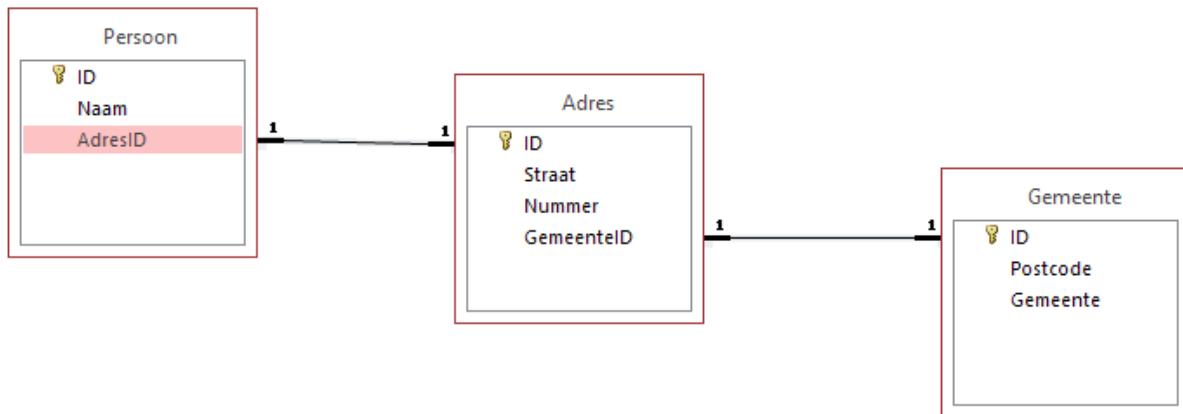
In grote lijnen zijn er 3 verschillende soorten relaties die kunnen bestaan tussen 2 tabellen. Dit is de 1 op 1 relatie, 1 op N relatie en de N op N relatie waarbij de 1 of N (oneindig) aangeeft hoeveel records uit de ene tabel kunnen overeen komen met records uit de andere tabel. De indicatie met 1 of N wordt ook de kardinaliteit genoemd.

De relaties tussen de tabellen worden gevisualiseerd binnen een Entiteit Relatie Diagram of ER-Diagram. Er bestaan veel verschillende tools om dergelijke diagrammen aan te maken, maar MS Access voldoet hier ruimschoots voor.

1 op 1 relatie

Bij deze relatie zal 1 record uit tabel A overeenkomen met 1 record uit tabel B. Een variant is de 1 op 0 relatie waar één record uit tabel A maximaal kan – maar niet moet – overeenkomen met 1 record uit tabel B. 1 op 1 relaties zijn echter eerder zeldzaam en zullen eerder afgedwongen worden door business logica dan door normalisatie.

Een voorbeeld is het loskoppelen van adresinformatie van een persoon uit de tabel persoon indien een persoon altijd maar maximum 1 adres kan hebben. We krijgen dan het volgende ER-diagram.

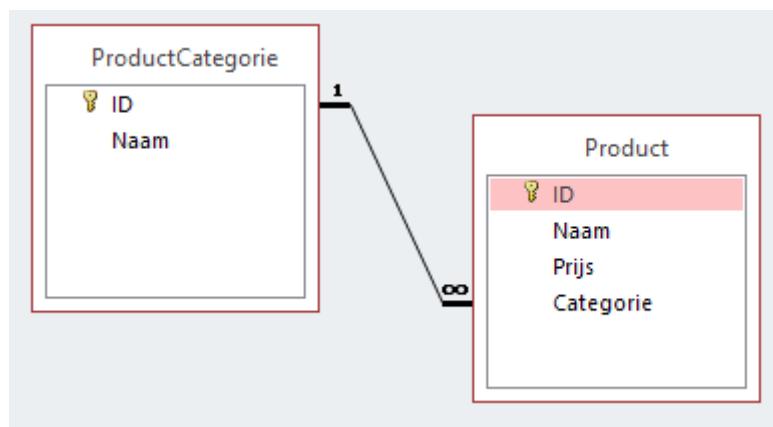


Dit schema toont aan dat een persoon altijd over exact 1 adres zal beschikken. Naar normalisatie toe kan dit perfect in dezelfde tabel, maar in bepaalde gevallen kan het wenselijk zijn dit in een afzonderlijke tabel op te slaan.

De tabel Adres heeft ook een 1 op 1 relatie met de tabel gemeente in de veronderstelling dat elk adres in exact 1 gemeente zal liggen.

1 op N relatie

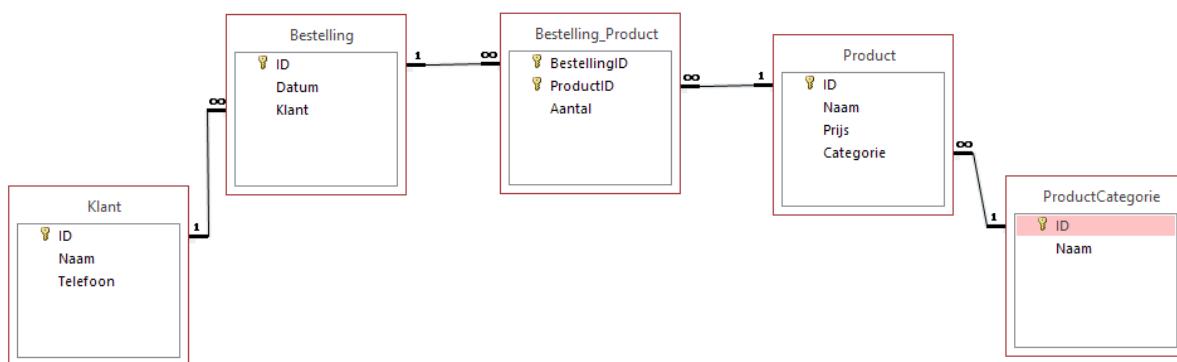
Bij deze relatie zal er exact 1 record uit tabel A overeenkomen met 0, 1 of meerdere records uit tabel B. Een typisch voorbeeld is het opsplitsen van producten in bepaalde categorieën. Elk product (tabel B) zal overeenkomen met exact 1 categorie (tabel A), maar een categorie (tabel A) zal gebruikt worden bij meerdere producten (tabel B). Dit wordt weergegeven in het onderstaande schema.



N op N relatie

Bij deze relatie zal een record uit tabel A overeenkomen met 0, 1 of meerdere records uit tabel B en zal een record uit tabel B overeenkomen met 0, 1 of meerdere records uit tabel A. Om een dergelijke relatie te kunnen afdwingen zal er gebruik gemaakt worden van een tussentabel die als primary key een combinatie bevat van de primary key van tabel A en B en eventueel wordt aangevuld met extra velden.

Een typisch voorbeeld hier is het bijhouden van bestellingen.



Eerst en vooral zijn er twee 1 of N relaties. Namelijk deze tussen Klant en Bestelling en tussen ProductCategorie en Product. Daarnaast is er ook een N op N relatie tussen Bestelling en Product. Een product kan immers voorkomen op meerdere bestelbonnen en een bestelbon kan meerdere producten bevatten. Daarom wordt er een tussentabel Bestelling_Product gemaakt met als extra informatie hoeveel er van elk product werd besteld.

Praktijkvoordeel

Voor de studentenadministratie is er een Excel document dat een overzicht biedt van alle studenten, de modules die ze volgen en de groep waarmee ze de modules volgen.

				SEMESTER 1						SEMESTER 3				
1				Design	ProjSkills	AppMaths	SSS	CNW		Webdev	BusApp	ProdMan	SSA	AdvNetw
2	naam	jaar	extra info											
4	Ivy Breyne	1NMCT1		1NMCT1	1NMCT1	1NMCT1	1NMCT1	1NMCT1						
5	Alexis	3NMCT												
6	Alice Anne	1NMCT8		1NMCT8	1NMCT8	1NMCT8	1NMCT8	1NMCT8						
7	Alvies Valentin	1NMCT4		1NMCT4	1NMCT4	1NMCT4	1NMCT4	1NMCT4		2NMCT2	2NMCT2	2NMCT2	2NMCT2	2NMCT2
8	Ariane Gobbe	2NMCT2												
9	Alice	1NMCT3		1NMCT3	1NMCT3	1NMCT3	1NMCT3	1NMCT3						
10	Alain Thomas	IOT2-3												

Als we het Excel document bekijken merken we dat een student niet elke module binnen dezelfde groep moet volgen en dat modules opgesplitst worden in semesters. Er is ook een kolom met het jaar voorzien en voor extra info. De kolom jaar moet eigenlijk de waarden 1NMCT, 2NMCT, 3 NMCT, IOT1-2, IOT2-3 of IOT1-2-3 bevatten. De kolom extra info kan informatie over de student bevatten.

Identificeren entiteiten

Om deze situatie om te zetten naar een databasemodel beginnen we met de entiteiten te zoeken. Je kan deze meestal detecteren door gewoon de situatie te omschrijven en de zelfstandige naamwoorden uit deze omschrijving te halen. In ons geval is dit:

"Studenten volgen een aantal **modules in een **semester** en volgen deze modules binnen een bepaalde **groep**".**

De entiteiten in dit geval zijn dus Student, Module, Semester en Groep. Merk op dat entiteiten altijd geformuleerd worden als enkelvoud.

Identificeren attributen

De volgende stap is om voor elke entiteit de attributen te identificeren. Bij dit voorbeeld zullen we alles eenvoudig houden en de meest noodzakelijk attributen gebruiken. Dit levert de volgende zaken op:

- Student
 - Naam
- Module
 - Naam
- Semester
 - Naam
- Groep
 - Naam

Toepassen eerste normaalvorm

Om aan de regels van de eerste normaalvorm te voldoen moet er enkel een primary key toegevoegd worden aan elke entiteit. Dit levert het volgende resultaat:

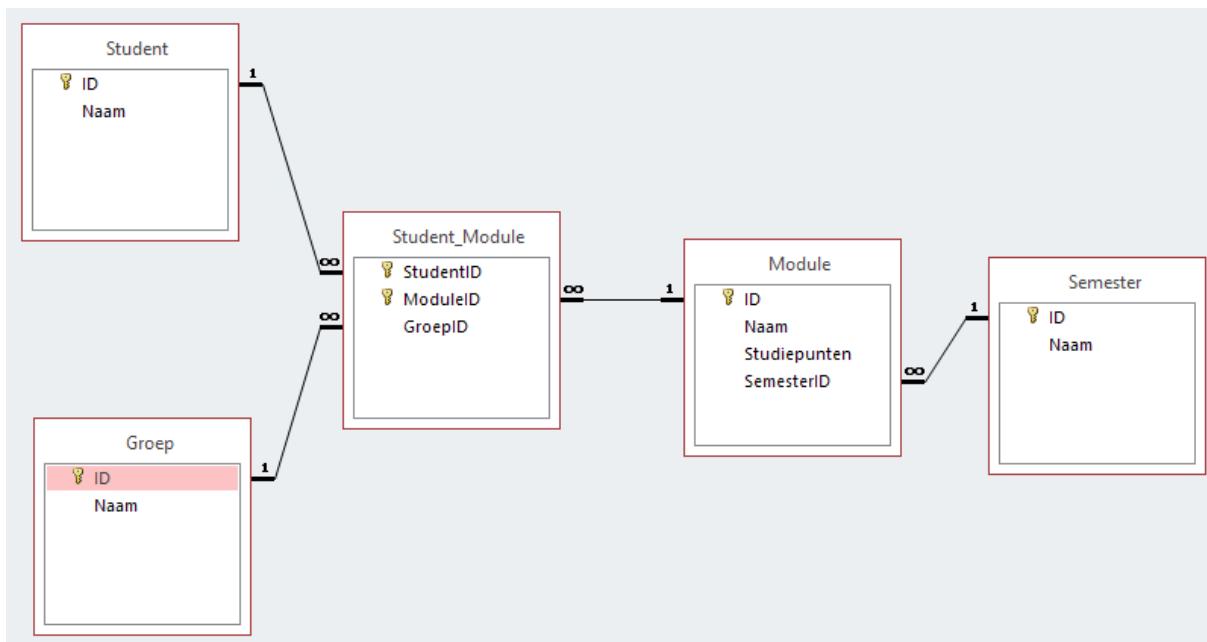
- Student
 - ID (PK)
 - Naam
- Module
 - ID (PK)
 - Naam
- Semester
 - ID (PK)
 - Naam
- Groep
 - ID (PK)
 - Naam

Toepassen tweede normaalvorm

De entiteiten zijn al opgesplitst en voor dit voorbeeld is er geen verdere opsplitsing meer nodig. Wat er wel nog moet gebeuren is een foreign key veld toevoegen voor de relaties. Dit voorbeeld bevat de volgende relaties:

- Er is een **N op N relatie** tussen **Student** en **Module** aangezien een student meerdere modules kan volgen en een module door meerder studenten wordt gevuld. Dit zal dus resulteren in het gebruik van een tussentabel **Student_Module**
- Er is een **1 op N relatie** tussen **Student_Module** en **Groep** aangezien een student de module maar met 1 groep zal volgen, maar een groep uit meerder studenten zal bestaan.
- Er is een **1 op N relatie** tussen **Module** en **Semester** aangezien een module maar in 1 semester zal plaatsvinden, maar een semester bevat meerdere modules.

Dit resulteert in het volgende schema:



Toepassen derde normaalvorm

Normaal gezien moeten er nu attributen die niet volledig afhankelijk zijn van de primary key verwijderd worden. Aangezien we het aantal attributen heel beperkt gehouden hebben stelt dit probleem zich hier niet en is het schema af. Mochten er bij module nog attributen zoals campus gezet hebben zouden deze moeten worden afgesplitst naar een afzonderlijke tabel.

Van database model naar object model

Na het ontwerpen van de database moet deze omgezet worden naar een model in C# code. Er bestaan tools zoals het Entity Framework die dit automatisch doen, maar binnen deze cursus wordt de code manueel geschreven.

De algemene regels om een database model om te zetten naar een object model zijn:

- Elke entity (tabel) wordt een class
- Elk attribute (kolom) wordt een property
- Een foreign key attribute krijgt als type de entity waar het naar verwijst.

Als we het database model van het vorige voorbeeld vertalen naar een object model krijgen we volgend resultaat:

```

class Student
{
    public int ID { get; set; }
    public string Naam { get; set; }
}

class Groep
{
    public int ID { get; set; }
    public string Naam { get; set; }
}

class Module
{
    public int ID { get; set; }
}
  
```

```
public string Naam { get; set; }
public int Studiepunten { get; set; }
public Semester Semester { get; set; }
}

class StudentModuleGroep
{
    public Student Student { get; set; }
    public Module Module { get; set; }
    public Groep Groep { get; set; }
}

class Semester
{
    public int ID { get; set; }
    public string Naam { get; set; }
}
```

Hou er wel opnieuw rekening mee dat het vertalen van een database model naar een object model geen exacte wetenschap is. Je kan van de regels afwijken indien daar een goed gemotiveerde reden toe is.

Oefeningen: databases ontwerpen

Oefening 1

De Kinepolis groep wenst hun voorstellingen bij te houden in een database. Een voorstelling bestaat uit de volgende zaken:

- De voorstelling gaat door in een bepaalde zaal
- Een zaal maakt deel uit van een bioscoop
- De voorstelling bevat een bepaalde film
- De voorstelling gaat door op een bepaald tijdstip
- Een Film heeft een bepaald genre

Lees enkele keren aandachtig de requirements hierboven en bepaal de verschillende entiteiten met hun attributen en de relaties tussen de entiteiten. Maak daarna het database model op papier.

Oefening 2

Bij deze oefening wordt er een database model ontworpen voor een gegeven situatie. Werk het model uit op papier volgens de stappen die gezien werden in de theorie. De situatie is als volgt:

Radio Quindo wil een systeem om een playlist op te stellen voor elke uitzending. Deze playlist moet de volgende informatie bevatten:

- Titel van het nummer
- Duur van het nummer
- Naam van de artiest
- Titel van het album + het jaar van publicatie
- Genre van het nummer
- Tijdstip waarop het nummer moet afgespeeld worden

Daarnaast moet er nog informatie over de uitzending worden bijgehouden zodat de software die bovenop dit datamodel komt vlot de verschillende playlists kan terugvinden. Er moet dus informatie bijgehouden worden over:

- De verschillende programma's van Quindo. Een programma heeft ook exact één bepaald genre (nieuws, talkshow, verzoekprogramma...)
- De verschillende uitzendingen van het programma (datum en tijd)
- De verschillende presentatoren en de link met hun programma(s).

Lees de bovenstaande situatie enkele keren aandachtig en probeer alle entiteiten en attributen te detecteren. Leg dan de relaties tussen de verschillende entiteiten en controleer de normaalvormen. Er zijn meerdere oplossingen mogelijk, check jouw oplossing af met de labo docent.

Oefening 3

Het database model uit oefening 1 of 2 wordt omgezet naar een MS SQL Server 2012 database. Volg de instructies van de docent om de eerste tabellen aan te maken en werk daarna zelfstandig de database af. Je kan de informatie om een nieuwe database aan te maken terug vinden op <http://msdn.microsoft.com/en-us/library/ms186312.aspx>. Je kan de informatie om tabellen aan te maken terugvinden op <http://msdn.microsoft.com/en-us/library/ms188264.aspx>. Meer informatie over het leggen van relaties tussen tabellen is te vinden op <http://msdn.microsoft.com/en-us/library/ms189049.aspx>

ADO.NET

Intro

ADO.NET is het deel van het .NET framework dat instaat voor de communicatie met databronnen. Deze databron hoeft geen relationele database te zijn, maar dit is wel de situatie die we in deze module het meest zullen gebruiken. Enkele voorbeelden van producten die het werken met relationele databases toelaten zijn MS Access, MS SQL Server en MySQL. ADO.NET laat toe om met elk van deze types te communiceren. Hou echter wel in het achterhoofd dat ADO.NET ook gebruikt kan worden om data uit CSV, Excel, HTML,... files te halen.

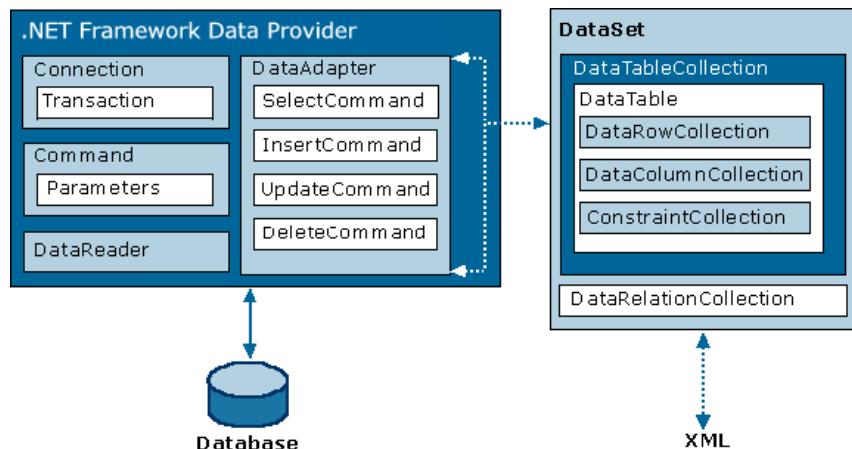
Naast ADO.NET is er ook nog LINQ to SQL en het Entity Framework om te communiceren met een database, maar binnen deze module wordt enkel ADO.NET gebruikt.

Web Api

Alle data access binnen deze module zal afgehandeld worden via een Web Api. Deze Api zal dan XML of JSON genereren als response. Voor verdere info rond Web API wordt verwezen naar de module Server Side Applications.

Structuur van ADO.NET

ADO.NET bestaat uit verschillende onderdelen. Deze worden in de volgende secties stuk per stuk besproken.



Figuur 15: structuur van ADO.NET (<http://msdn.microsoft.com/en-us/library/27y4ybxw.aspx>)

ADO.NET kent verschillende dataproviders. Voor MS Access en MS SQL Server databases is dit bijvoorbeeld OLEDB. Er is voor MS SQL Server echter ook de SQL data provider die een hogere performantie biedt. Voor MySQL kan een afzonderlijke data provider geïnstalleerd worden via <http://dev.mysql.com/downloads/connector/net/5.0.html>. De dataproviders erven echter over van dezelfde abstracte classes zodat de verschillen voor de developer zo klein mogelijk kunnen gehouden worden.

Elke dataprovider heeft een Connection class die overerft van de abstracte **DbConnection** class (<http://msdn.microsoft.com/en-us/library/system.data.common.dbconnection.aspx>). Zowel de OleDbConnection class (<http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbconnection.aspx>) als de SqlConnection class (<http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx>) erven over van deze class. De belangrijkste onderdelen van de connectie zijn de connectionstring en het openen en sluiten van de connectie. Ook het werken met transacties is mogelijk, maar deze komen straks aan bod.

Vervolgens is er de **DbCommand** class (<http://msdn.microsoft.com/en-us/library/system.data.common.dbcommand.aspx>) die ook door zowel OLEDB (<http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbcommand.aspx>) als SQL (<http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx>) wordt gebruikt. Hier wordt het SQL commando opgesteld dat naar de database zal gestuurd worden en is het mogelijk om parameters toe te voegen aan het commando.

Tot slot is er de **DbDataReader** class (<http://msdn.microsoft.com/en-us/library/system.data.common.dbdatareader.aspx>) die het mogelijk maakt om het resultaat van een SELECT statement te verwerken op een gelijkaardige manier als het inlezen van tekstbestanden via de StreamReader. Er is ook hier een versie beschikbaar voor OLEDB en SQL.

Een alternatief voor de DataReader is het gebruik van een DataSet met één of meerdere DataTables(s), hoewel het belangrijk is om te weten dat een DataSet ook een DataReader zal gebruiken op de achtergrond. Daarom zal de focus in deze cursus liggen op het verwerken van de data via de DataReader. Wie met een DataSet wenst te werken kan een voorbeeld vinden op <http://msdn.microsoft.com/en-us/library/ss7fbaez.aspx>.

DbProviderFactories

Een probleem bij ADO.NET is dat er verschillende code moet geschreven worden afhankelijk van de data provider. Zo zal een MS SQL Server database een SqlConnection gebruiken terwijl een MS Access database een OleDbConnection zal nodig hebben.

Dit probleem wordt opgelost door gebruik te maken van het Factory pattern. Verdere informatie en voorbeelden over het Factory pattern zijn te vinden op <http://msdn.microsoft.com/en-us/library/ee817667.aspx> maar valt buiten de scope van de cursus.

Het is echter wel belangrijk om te weten dat dit principe gebruikt wordt binnen ADO.NET via de class **DbProviderFactories** (<http://msdn.microsoft.com/en-us/library/system.data.common.dbproviderfactories.aspx>). Deze class bevat een method **GetFactory()** die een instantie van de **DbConnection** of **DbCommand** class kan aanmaken op basis van een opgegeven provider.

Op die manier moet het exacte type van een connectie (OleDbConnection of SqlConnection) niet opgegeven worden in de code en kan deze gewijzigd worden zonder enige aanpassing in de broncode. Dit principe wordt in de volgende secties in de praktijk getoond.

Connectie maken

De eerste stap om te kunnen communiceren met een database is het maken van een connectie met die database via een connectionstring. De connectionstring zal verschillen van databron tot databron. Bijvoorbeeld: de connectionstring om verbinding te maken met een database op een MS SQL Server zal anders zijn dan die om te connecteren met een MS Access database of een MySQL database. De website <http://www.connectionstrings.com/> toont een mooi overzicht van alle mogelijke verschillende connectionstrings.

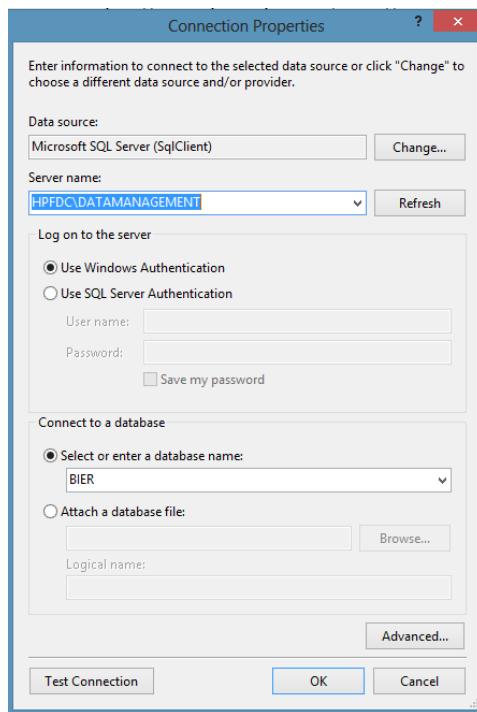
Het volgende stuk code toont hoe connectie kan gemaakt worden met een database:

```
DbConnection con =  
DbProviderFactories.GetFactory("System.Data.SqlClient").CreateConnection();  
con.ConnectionString = "@Data Source=... Security=SSPI;Initial Catalog=BIER";  
con.Open();
```

Eerst wordt er een variabele aangemaakt die de connection zal bijhouden. De property **ConnectionString** wordt ingevuld en tot slot wordt de connectie geopend. Als parameter bij de method **GetFactory** wordt de provider opgegeven zodat ADO.NET weet welk soort connectie er moet gemaakt worden.

Het is echter niet aan te raden om de connectionstring en de provider hard te coderen in de broncode. Als de connectionstring of de provider wijzigt zal de code moeten aangepast worden wat best te vermijden is. Een Web Api project bevat een file **Web.config** die in XML formaat een ganse boel settings kan bevatten. Het grote voordeel is dat deze file na het deployen van de applicatie nog kan aangepast worden zodat de broncode zelf niet meer moet gewijzigd worden.

In Visual Studio zit er een handige tool om de connectionstring in de **Web.config** file in te vullen. Ga naar de properties van je project en kies voor **Settings**. Daar kan je een **ConnectionString** als nieuwe setting toevoegen en krijg je een wizard om je connectionstring in te stellen. Je kan ook testen of je connectionstring correct is.



Figuur 16: aanmaken van een connectionstring

Bij het openen van de file Web.config werd de volgende XML code aangemaakt:

```
<connectionStrings>
    <add name="DefaultConnection" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-nmct.ba.ado.api-
20141015115331.mdf;Initial Catalog=aspnet-nmct.ba.ado.api-20141015115331;Integrated
Security=True"
        providerName="System.Data.SqlClient" />
    <add name="nmct.ba.ado.api.Properties.Settings.ConnectionString"
        connectionString="Data Source=xxx;Initial Catalog=BIER;Integrated Security=True"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

Je vervangt beste het name attribuut naar een meer duidelijke naam. In je broncode kan je dan verwijzen naar deze setting en ook de provider laten invullen op basis van de data uit Web.Config. Op die manier is het nu mogelijk om te wisselen van data provider zonder iets aan de broncode te wijzigen.

```
string provider =
ConfigurationManager.ConnectionStrings["ConnectionString"].ProviderName;
string connectionstring =
ConfigurationManager.ConnectionStrings["ConnectionString"].ConnectionString;

DbConnection con = DbProviderFactories.GetFactory(provider).CreateConnection();
con.ConnectionString = connectionstring;
con.Open();

con.Close();
```

Zorg er wel voor dat System.Configuration toegevoegd is bij de references, anders zal het niet lukken om de ConfigurationManager aan te spreken.

Command opstellen

Nadat de connectie met de database gemaakt werd kan een commando op de database uitgevoerd worden op basis van een SQL statement. Het opstellen van het commando zal opnieuw gebeuren aan de hand van de DbProviderFactories class. Als type zal altijd voor Text gekozen worden en er moet ook opgegeven worden via welke connectie dit commando moet worden uitgevoerd.

```
DbCommand command = DbProviderFactories.GetFactory(provider).CreateCommand();
command.CommandType = System.Data.CommandType.Text;
command.Connection = con;
command.CommandText = "SELECT * FROM bier";
```

Parameters meegeven

Het is ook mogelijk om parameters mee te geven met een SQL statement. Dit wordt echter **NIET** gedaan door deze in de SQL statement als variabele mee te geven om SQL injection te voorkomen.

Daarom wordt een nieuwe **DbParameter** aangemaakt waarbij een naam en waarde wordt opgegeven. In het onderstaande voorbeeld is de naam van de parameter "filter" en de waarde "bruin". Bij het SQL statement wordt de naam dan vooraf gegaan door het @ - teken.

```
DbParameter par = DbProviderFactories.GetFactory(provider).CreateParameter();
par.ParameterName = "filter";
par.Value = "bruin";
command.Parameters.Add(par);
command.CommandText = "SELECT * FROM bier WHERE Kleur=@filter";
```

Commando uitvoeren

Bij het manipuleren van data wordt de method ExecuteNonQuery() van het commando gebruikt. Deze method zal een integer returnen die het aantal rijen bevat die door de statement werden aangepast.

```
command.CommandText = "INSERT INTO...";
int affected = command.ExecuteNonQuery();
Console.WriteLine(affected);
```

Resultaat verwerken

Indien het commando een SELECT statement bevat zal deze een resultaat weergeven dat verwerkt moet worden door de applicatie. Zoals bij het deel "Structuur van ADO.NET" besproken is er hier de keuze tussen een DataReader en een DataSet. Binnen deze module wordt er gekozen voor een DataReader aangezien de extra functionaliteiten van een DataSet voor deze module niet relevant zijn.

De DataReader is goed te vergelijken met een StreamReader. Het resultaat zal lijn per lijn ingelezen worden via een loop. Binnen de loop kan dan de waarde van elke kolom opgehaald worden. In de meeste situaties zal een nieuw object aangemaakt worden binnen de loop en worden alle objecten aan een collection toegevoegd.

```
DbDataReader reader = command.ExecuteReader();
while (reader.Read())
{
    Bier b = new Bier() { Biernaam = reader["Biernaam"].ToString(), ... };
}
reader.Close();
```

Indien er slechts maar één enkele waarde moet opgehaald worden is het ook mogelijk om gebruik te maken van de method ExecuteScalar() die exact één waarde als object zal teruggeven (<http://msdn.microsoft.com/en-us/library/system.data.common.dbcommand.executescalar.aspx>).

DBNull

Het gebruik van een datareader kan problemen opleveren als een veld in een bepaalde rij de waarde null bevat. Bij string waarden zal dit automatisch omgezet worden naar een lege string, maar bij getallen en andere types zal er een error optreden omdat het niet mogelijk is om de null waarde om te zetten. Dit kan opgelost worden door te controleren op een DBNull waarde.

```
int score = 0;
if(!DBNull.Value.Equals(reader["Score"])){
    score = (int)reader["score"];
}
```

Indien de waarde null is krijgt de variabele een standaard waarde, in andere gevallen wordt de waarde uit de database gebruikt. Je hoeft dit niet voor alle velden te doen, enkel de velden die niet van het type string zijn en die toelaten om een null waarde te krijgen door de database.

Transacties

In sommige gevallen zal een actie uit meerdere SQL statements bestaan. Een voorbeeld dat veel gebruikt wordt is het overschrijven van geld tussen twee bankrekeningen. Het saldo van de ene gebruiker moet verminderd worden waar het saldo van de andere gebruiker moet vermeerderd worden. In dit geval moeten beide statements correct uitgevoerd worden of totaal niet uitgevoerd in het geval van problemen. Anders zal de ene gebruiker een stuk armer zijn en toch nog zijn factuur niet betaald hebben.

Transacties kunnen dergelijke problemen oplossen door het gebruik van een Commit (voer alle acties uit) of een Rollback (hou alle acties tegen).

```
DbTransaction trans = con.BeginTransaction();

try
{
    DbCommand command = DbProviderFactories.GetFactory(provider).CreateCommand();
    command.CommandType = System.Data.CommandType.Text;
    command.Transaction = trans;
    command.Connection = con;
    command.CommandText = "UPDATE...";

    DbCommand command2 = DbProviderFactories.GetFactory(provider).CreateCommand();
    command2.CommandType = System.Data.CommandType.Text;
    command2.Transaction = trans;
    command2.Connection = con;
    command2.CommandText = "UPDATE...";

    command.ExecuteNonQuery();
    command2.ExecuteNonQuery();
    trans.Commit();
}
catch (Exception)
{
    trans.Rollback();
    throw;
}
```

Er wordt eerst een nieuwe transactie aangemaakt voor de connectie en deze transactie wordt ingesteld op elk command dat binnen de transactie valt. Na het uitvoeren van beide commands wordt de method Commit() van de transactie opgeroepen en worden beide commands definitief uitgevoerd. Als er iets fout zou lopen bij het uitvoeren van één van de commands worden de wijzigingen van alle vorige commands ongedaan gemaakt via de method Rollback().

Error handling

Acties van en naar de database kunnen er gevoelig zijn aan exceptions. Een database kan bijvoorbeeld tijdelijk niet beschikbaar zijn of een query kan een verkeerde parameter bevatten. Daarom is het belangrijk om altijd try en catch statements te gebruiken bij het ophalen of wegschrijven van data in een database.

Oefeningen ADO.NET

Oefening 1

Tijdens deze oefening wordt er een class gemaakt die zal functioneren als helper class voor alle acties naar de database. Deze class zal gebruik maken van de abstracte classes van ADO.NET zodat ze bruikbaar is voor zowel OLEDB, SQL en MySQL.

Eerst maken we een database aan zodat de gemaakte functionaliteit getest kan worden:

- Open de SQL Server Management console en maak een database aan met de naam BankDemo
- Maak een tabel Rekening aan met de volgende velden:
 - ID (autonummer)
 - Rekeninghouder (nvarchar(50))
 - Rekeningnummer (nvarchar(14))
 - Saldo (numeric(18,0))
- Voeg enkele records in bij deze tabel

Tijdens de volgende stap kan de class Database aangemaakt worden.

- Maak een nieuwe Visual Studio solution aan met als naam: nmct.ba.week6.oefening1
- Maak een Web API aan met als naam nmct.ba.week6.oefening1.api
- Maak een connectionstring aan voor de MS SQL Server database met de System.Data.SqlClient dataprovider in Web.config. Geef deze de naam "ConnectionString".
- Maak een map Helper aan met daarbinnen een class Database
- Maak een static private method GetConnection aan die een DbConnection zal teruggeven op basis van een parameter connectionstring waar je de naam van de ConnectionString zal meegeven. Zorg er ook voor dat deze connectie geopend wordt voor je ze teruggeeft.
- Maak een public static method ReleaseConnection die een connectie als parameter binnen krijgt en deze connectie zal afsluiten. De reden dat deze method public moet staan is omdat ze later gebruikt zal worden bij transacties.

Voorlopig hebben we voldoende functionaliteit om een connectie met de database te kunnen openen en sluiten. In de volgende stap wordt een method geschreven om een commando te kunnen opbouwen.

- Maak een private static method BuildCommand die de naam van de connectionstring heeft als parameter en ook een SQL statement. Deze parameter zal ook van het type string zijn. Daarnaast kan er ook een array van DbParameter worden meegegeven met een variabel aantal elementen, maar dit mag echter niet verplicht zijn. Maak daarom gebruik van het keyword params (<http://msdn.microsoft.com/en-us/library/w5zay9db.aspx>).
- Maak een nieuw SqlCommand aan, stel het type en text in en voeg de parameters toe indien nodig. Return daarna het command. Om de connection in te stellen roep je de method CreateCommand() op via de method GetConnection().

Nu het command is opgebouwd kan het uitgevoerd worden. Hier zijn er twee mogelijkheden: ofwel moet er data worden terug gegeven (SELECT statement) of wordt het aantal bewerkte rijen terug gegeven (UPDATE en DELETE), ofwel het ID van een nieuwe rij (INSERT). Hiervoor worden twee methods aangemaakt: GetData() en ModifyData().

- Maak een public static method GetData die een DbDataReader zal terug geven en een connectionstring, een sql statement en een array van parameters krijgt als parameters.
- Gebruik de method BuildCommand() om het commando op te bouwen.
- Gebruik de method ExecuteReader() om het commando uit te voeren. De reader moet wel afgesloten worden na het ophalen van de data. Return tot slot de reader
- Vergeet niet om de nodige exception handling te voorzien
- Maak een method ModifyData() aan die dezelfde parameters krijgt als GetData() maar een integer zal returnen. Deze method is gelijkaardig aan GetData() maar zal de method ExecuteNonQuery() van het command uitvoeren en het resultaat (zijnde het aantal bewerkte rijen) teruggeven.
- Vergeet ook hier niet de error handling
- Maak tot slot nog een method InsertData die het ID van het nieuw aangemaakte record zal teruggeven. Dit kan je doen via de volgende code:

```
command.Parameters.Clear();
command.CommandText = "SELECT @@IDENTITY";

int identity = Convert.ToInt32(command.ExecuteScalar());
```

Er moet nog een extra method aangemaakt worden die toestaat om parameters toe te voegen van uit de model classes

- Maak een public static method aan die een DbParameter terug geeft op basis van een connectionstring (string), een naam (string) en een value (object)
- Maak gebruik van de DbProviderFactories class om een nieuwe parameter aan te maken.
- Vul de naam en waarde in en geef de DbParameter terug.

Momenteel bevat deze class al de nodige functionaliteit zolang er niet met transacties moet gewerkt worden. Daarom wordt nog een method toegevoegd om het gebruik van transacties mogelijk te maken en wordt overloading gebruikt bij BuildCommand, GetData() en ModifyData() zodat een transactie kan worden meegegeven.

- Maak een public static method die een DbTransaction zal terug geven. Maak binnen deze method een nieuwe connectie aan en roep de method BeginTransaction() aan op deze connectie.
- Vergeet hier de error handling niet
- Maak een nieuwe method BuildCommand() die gelijkaardig is aan de huidige, maar als extra parameter een transaction krijgt.
- Maak een nieuw command op basis van de connection van de transaction. De overige code is dezelfde als in de originele implementatie
- Maak een nieuwe method GetData() die gelijkaardig is aan de huidige, maar als extra parameter een transaction krijgt. Het enige verschil met de originele implementatie is dat de transactie wordt doorgegeven naar de method BuildCommand en wordt ingesteld op het command.
- Doe tot slot hetzelfde voor de methods ModifyData() en InsertData(). Ook hier wordt de transactie als extra parameter mee gegeven en ingesteld op het commando.

Oefening 2

Nu deze class afgewerkt is kan ze getest worden in een oefening. Maak hiervoor een nieuw Class Library project aan met de naam nmct.ba.week6.oefening1.model en maak een class Bankaccount aan met de volgende properties:

- ID (string)
- AccountHolder (string)
- AccountNumber (string)
- Balance (double)

Voeg deze class library als reference toe aan het Web API project.

Maak in de map Models van het Web API project een class BankaccountDA aan. In deze class zullen we de nodige code schrijven voor de data access. Maak de volgende methods aan:

- GetBankaccounts() die een List<Bankaccount> zal teruggeven
- GetBankaccount(int id) die een instantie van Bankaccount zal teruggeven
- InsertAccount(Bankaccount ba) die een integer zal teruggeven met het ID
- UpdateAccounts(int sender, int receiver, double amount) die een int zal teruggeven met het aantal bewerkte rijen

De volgende code zou moeten volstaan om de data op te halen door gebruik te maken van de Database helper class: `DbDataReader reader = Database.GetData("SELECT * FROM Rekening");`

- Deze code zal een DbDataReader teruggeven. Via een loop kunnen alle records van deze reader overlopen worden en kan er telkens een nieuwe instantie aan de collection worden toegevoegd.
- Het is echter ook mogelijk om nog een helper method aan te maken die telkens één rij zal verwerken. Dit kan heel handig zijn en de code hiervoor ziet er als volgt uit:

```
private static Account Create(IDataRecord record)
{
    return new Account()
    {
        ID = record["ID"].ToString(),
        AccountHolder = record["Rekeninghouder"].ToString(),
        AccountNumber = record["Rekeningnummer"].ToString(),
        Balance = Double.Parse(record["Saldo"].ToString())
    };
}
```

Maak nu een nieuwe Controller aan in je Web API met als naam BankaccountController en gebruik de GetBankAccounts() binnen de Get() method van de controller. Run daarna je service en test of je alle records in de browser ziet.

Indien alles werkt kan je de method UpdataAccount aanmaken die geld zal transfereren van de ene account naar de andere. Dit is een mooi voorbeeld waar een transactie moet gebruikt worden.

- Maak een method UpdateAccount die een integer zal teruggeven en het id binnen krijgt voor de verzender en voor de ontvanger. De laatste parameter is het bedrag dat moet overgeschreven worden.
- Start een nieuwe transactie binnen deze method
- Maak de nodige sql statements om de rekening van de verzender te verminderen en deze van de ontvanger te vermeerderen. Je zal hier de nodige parameters moeten toevoegen:

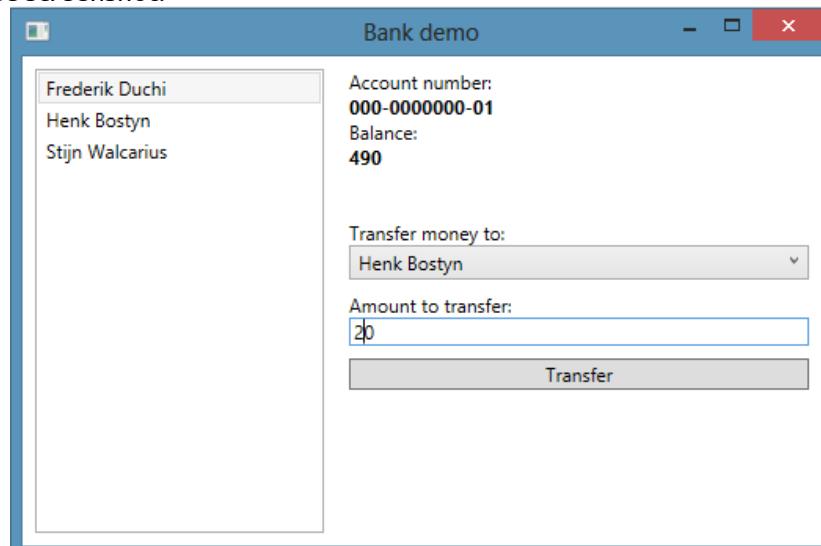
```
string sql = "UPDATE Rekeningen SET Saldo=Saldo-@Amount WHERE ID=@ID";
DbParameter par1 = Database.AddParameter("@Amount", amount);
DbParameter par2 = Database.AddParameter("@ID", sender);
rowsaffected += Database.ModifyData(trans, sql, par1, par2);
```

- Vergeet niet om Commit() uit te voeren op de transactie en voorzie ook de nodige error handling zodat je in een catch statement een Rollback() kan uitvoeren. Sowieso moet de connectie nog afgesloten worden via de method ReleaseConnection() uit de database class.

Vervolledig tot slot de methods zoals gevraagd en maak ook de nodige Post en Put methods aan in de controller. Je kan de update en insert methods niet direct in de browser testen, maar Fiddler is een goede tool om dit te doen.

Nu de API klaar is kan de WPF applicatie aangemaakt worden. Je zal ook de HttpClient en JSON package van NuGet moeten installeren via de package manager.

Bij het maken van deze applicatie zal er al een ViewModel gebruikt worden. In het volgende hoofdstuk over MVVM wordt er meer uitleg over ViewModel gegeven. De applicatie ziet er uit als bij onderstaande screenshot:



Figuur 17: labo 5 - oefening 2 eindresultaat

Aan de linkerzijde is er een listbox, langs de rechterzijde is er een stackpanel voorzien met enkele textblokken, een combobox, een textbox en een button. Leg ook een referentie naar de class library zodat het model gekend is.

Maak een map ViewModel aan met daarbinnen een class BankAccountVM. Stel deze class in als DataContext op de Grid van de View.

Maak ook een class ObservableObject aan die de INotifyPropertyChanged interface zal implementeren zoals gezien in vorige oefeningen. Laat de class BankAccountVM overerven van deze class.

Haal alle bankaccounts op in de constructor van BankAccountVM. Gebruik hier voor de volgende code:

```
public BankAccountVM()
{
    GetBankAccounts();
}

private ObservableCollection<Bankaccount> _bankaccounts;
public ObservableCollection<Bankaccount> BankAccounts
{
    get { return _bankaccounts; }
    set { _bankaccounts = value; OnPropertyChanged("BankAccounts"); }
}

private async void GetBankAccounts()
{
    using (HttpClient client = new HttpClient())
    {
        HttpResponseMessage response = await
client.GetAsync("http://localhost:55313/api/bankaccount");

        if (response.IsSuccessStatusCode)
        {
            string json = await response.Content.ReadAsStringAsync();
        }
    }
}
```

```
        BankAccounts =
JsonConvert.DeserializeObject<ObservableCollection<Bankaccount>>(json);
    }
}
}
```

Voor meer uitleg over het gebruik van await / async en de HttpClient library wordt verwezen naar de module Server Side Applications.

Bind nu de property BankAccounts aan de listbox en de namen zouden moeten verschijnen.

Werk nu de applicatie verder af zodat het mogelijk is om geld over te schrijven van de ene rekening naar de andere. De code om de Put() method aan te spreken is als volgt:

```
public async void DoTransaction()
{
    using (HttpClient client = new HttpClient())
    {
        object[] data = { SelectedBankAccount.ID, TransferToAccount.ID, Amount
};
        string json = JsonConvert.SerializeObject(data);

        HttpResponseMessage response = await
client.PutAsync("http://localhost:55313/api/bankaccount", new StringContent(json,
Encoding.UTF8, "application/json"));
        if (response.IsSuccessStatusCode)
        {
            string jsonresponse = await response.Content.ReadAsStringAsync();
            int result = JsonConvert.DeserializeObject<int>(jsonresponse);
            SelectedBankAccount.Balance -= Amount;
            OnPropertyChanged("SelectedBankAccount");
            TransferToAccount.Balance += Amount;
        }
    }
}
```

Oefening 3

Test nu ook of je applicatie werkt met MS Acces en MySQL. Je kan hiervoor nog extra Connectionstrings aanmaken. Maak ook de database aan voor deze platformen

Merk op dat je voor het gebruik van MySQL de connector moet downloaden en toevoegen. Je moet ook het gedeelte binnen <system.data> toevoegen aan de Web.config zodat deze provider kan gebruikt worden.

```
<system.data>
  <DbProviderFactories>
    <add name="MySQL Data Provider" invariant=" MySql.Data.MySqlClient"
description=".Net Framework Data Provider for MySQL"
type=" MySql.Data.MySqlClient.MySqlClientFactory, MySql.Data" />
  </DbProviderFactories>
</system.data>
```

Model View ViewModel (MVVM)

Intro

Tijdens de introductie van de cursus werd het begrip MVVM, het voordeel en de verschillende onderdelen al eens aangehaald. Met de kennis van Databinding, ObservableCollection en Commands is het mogelijk om dit nu volledig in de praktijk te implementeren aangezien dit de kern elementen van MVVM zijn.

Nog eens een korte samenvatting van de verschillende onderdelen:

- **Model:** dit is alle code die je data model beschrijft, aangevuld met alle code om data op te halen of te manipuleren in de database. In ons geval is dit C# code.
- **View:** dit is de grafische user interface van je applicatie. In ons geval is dit XAML code.
- **ViewModel:** is de laag die tussen de model en de view zit en in feite het model van de view. Deze laag zorgt er voor dat het model en de view niet rechtstreeks met elkaar moeten communiceren.

Het is voor de communicatie tussen de View en het ViewModel dat er gebruik wordt gemaakt van Commands, Notifications en Databinding.

Eerste demo

Deze demo zal de oefening met de info uit de shops uit het vorige hoofdstuk maken volgens het MVVM pattern en zo de verschillende stappen illustreren. Momenteel wordt er gewerkt zonder framework zodat alle verschillende begrippen duidelijk zijn.

Algemene structuur

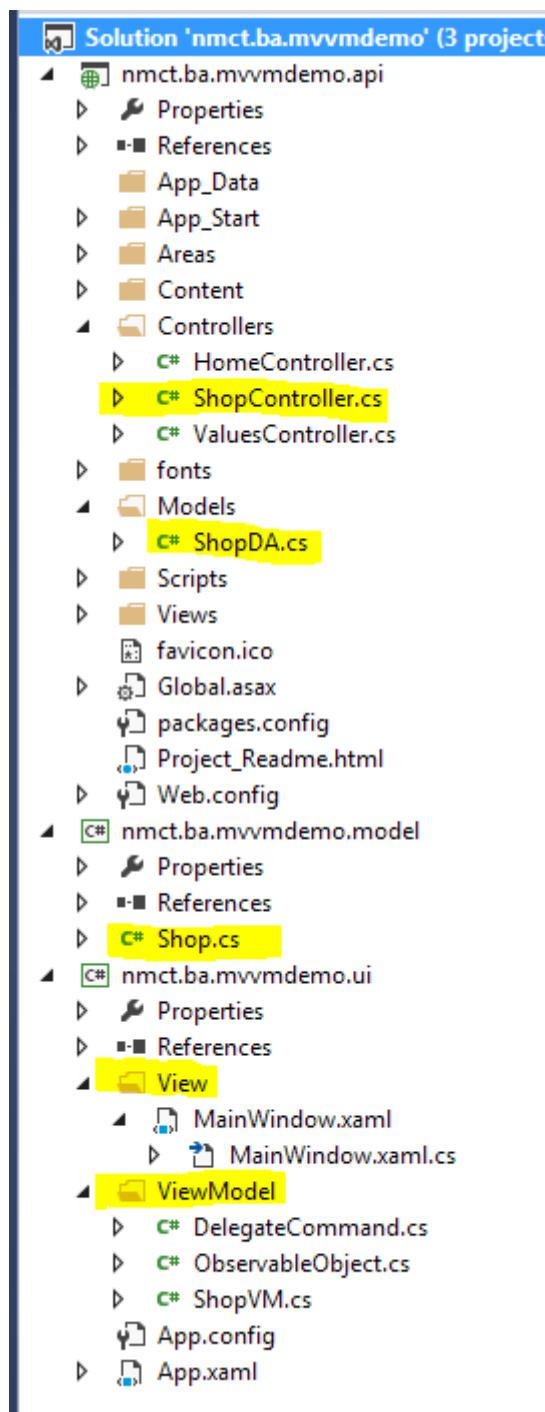
De structuur van het project ziet er als volgt uit. Er zijn 3 verschillende projecten binnen de solution:

- Nmct.ba.mvvmdemo.model: die van elke class de properties zal beschrijven
- Nmct.ba.mvvmdemo.api: die data access zal regelen
 - Models: bevat een class ShopDA voor de data access
 - Controllers: bevat ShopController voor Get, Post, Put en Delete operaties
- Nmct.ba.mvvmdemo.ui: is een WPF project voor de user interface
 - View: een map met alle XAML code
 - ViewModel: een map met classes die binding tussen View en Model mogelijk maakt

Er is ook een referentie gelegd tussen het API project en het model project en tussen het UI project en het model project.

Bij het UI project werden via de Package Manager de libraries voor HttpClient en JSON.NET toegevoegd om te kunnen communiceren met de API.

Tot slot moet nog ingesteld worden dat bij het runnen van de applicatie zowel de API als de UI moet opgestart worden. Dit kan je doen door rechts te klikken op de Solution en te kiezen voor Properties. Bij Common Properties → Startup Projects kan je kiezen voor Multiple startup projects.



Figuur 18: MVVM structuur

Er werd voor elk onderdeel een afzonderlijk mapje aangemaakt. Dit is niet verplicht, maar helpt wel om het overzicht te behouden in grotere projecten. Een ander voordeel is dat elke class automatisch onder de juiste namespace komt.

Er zijn twee helper classes aangemaakt in het ViewModel. Deze worden straks in detail besproken.

Model

Het model bevat alle properties die een bepaalde class kan hebben. In dit geval is er maar één class nodig, namelijk Shop. Deze zal verschillende properties bevatten. De methods om de data op te halen of te manipuleren worden geschreven in het API project.

```
public class Shop
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
}
```

In de class ShopDA van het API project zal dan de volgende code geschreven worden:

```
public static List<Shop> GetShops()
{
    List <Shop> list = new List <Shop>();
    ...
    return list;
}

public static void AddShop(Shop s){ Console.WriteLine(...); }
public static void Remove(Shop s) { Console.WriteLine(...); }
public static void EditShop(Shop s) { Console.WriteLine(...); }
```

Vervolgens moet in de ShopController de juiste method aangeroepen worden om zo de methods beschikbaar te stellen via de API.

```
public class ShopController : ApiController
{
    public List<Shop> Get()
    {
        return ShopDA.GetShops();
    }

    public HttpResponseMessage Post(Shop s)
    {
        ShopDA.AddShop(s);
        return new HttpResponseMessage(HttpStatusCode.OK);
    }

    public HttpResponseMessage Put(Shop s)
    {
        ShopDA.EditShop(s);
        return new HttpResponseMessage(HttpStatusCode.OK);
    }

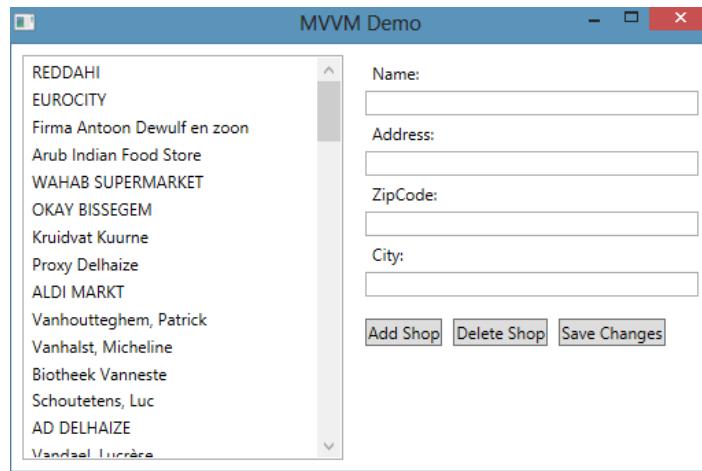
    public HttpResponseMessage Delete(int id)
    {
        ShopDA.Remove(id);
        return new HttpResponseMessage(HttpStatusCode.OK);
    }
}
```

Zoals bovenstaande code toont wordt de INotifyPropertyChanged interface niet meer geïmplementeerd in het model zelf. Dit zal verschuiven naar het ViewModel omdat het ViewModel verantwoordelijk is voor de notifications. Het enige doel van het Model is en blijft het beschrijven van de data structuur en het uitvoeren van manipulaties op deze data.

View

De view bevat de grafische user interface. In dit geval is er een ItemsControl nodig om een overzicht van alle shops in weer te geven en ContentControls om de details weer te geven. Er kan gekozen worden voor een listbox met textboxen voor het detail, eendatagrid of een listview. Dit toont de grote kracht van MVVM, namelijk dat de view aangepast kan worden zonder dat dit veel wijzigingen met zich meebrengt in andere delen van de code.

Voor dit voorbeeld werd er gekozen voor een listbox, textboxen en dan nog verschillende buttons om shops toe te voegen, te verwijderen en te bewerken. De user interface ziet er als volgt uit:



Figuur 19: MVVM de View

De XAML om tot deze interface te komen is als volgt:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <ListBox Margin="8" DisplayMemberPath="Name"/>
    <StackPanel Margin="8" Grid.Column="1">
        <Label Content="Name:"/>
        <TextBox Text="" />
        <Label Content="Address:"/>
        <TextBox Text="" />
        <Label Content="ZipCode:"/>
        <TextBox Text="" />
        <Label Content="City:"/>
        <TextBox Text="" />
        <StackPanel Orientation="Horizontal" Margin="0,16,0,0">
            <Button Content="Add Shop"/>
            <Button Content="Delete Shop" Margin="8,0,0,0"/>
            <Button Content="Save Changes" Margin="8,0,0,0"/>
        </StackPanel>
    </StackPanel>
</Grid>
```

Het binden van de data wordt pas gedaan nadat het ViewModel werd gemaakt.

ViewModel

Het ViewModel zorgt voor het converteren van het Model in bruikbare gegevens voor de View. Eerst en vooral worden 2 classes aangemaakt die in elk project bruikbaar zijn binnen de ViewModel

class. Dit is een class voor de implementatie van de `INotifyPropertyChanged` interface en een class voor de implementatie van de `ICommand` interface.

De class `ObservableObject` is een exacte copy van de class `BaseModel` die tijdens het vorige labo werd gebruikt. De class implementeert de `INotifyPropertyChanged` interface en zorgt voor een public method die het event zal afduren:

```
public class ObservableObject : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

De volgende class is `RelayCommand`. Deze class zal de `ICommand` interface implementeren en een Action uitvoeren. Alle informatie over een Action is te vinden op <http://msdn.microsoft.com/en-us/library/018hxwa8.aspx>. Kort samengevat komt het er op neer dat een Action een bepaalde method kan oproepen om uit te voeren. Voorwaarde is wel dat de method geen waarde moet teruggeven en het aantal parameters is beperkt tot 1.

De Action komt binnen in de constructor als parameter en wordt lokaal in de class opgeslagen. Bij het aanroepen van de method `Execute` wordt de action dan uitgevoerd. Er is ook een method `CanExecute()` voorzien om de controls te laten enablen of disablen. Deze geeft voorlopig in alle gevallen true terug.

Met het event `CanExecuteChanged` wordt er voorlopig ook niets gedaan, maar deze code moet er staan om de interface te implementeren. De code ziet er als volgt uit:

```
private Action _action;

public RelayCommand(Action a)
{
    _action = a;
}

public bool CanExecute(object parameter)
{
    return true;
}

public event EventHandler CanExecuteChanged;

public void Execute(object parameter)
{
    _action();
}
```

De classes kunnen nu in elke ViewModel gebruikt worden en zullen het opzetten van de ViewModel een stuk eenvoudiger maken.

Het ViewModel voor de Shop noemt `ShopVM.cs`. Deze class moet overerven van `ObservableObject` om notifications te kunnen implementeren. Daarnaast moet er een property zijn om alle shops in een lijst op te slaan en een property om de geslecteerde shop uit de lijst in op te slaan. Tot slot wordt de collectie van shops opgehaald in de constructor. Dit wordt in code als volgt gedaan:

```
public ShopVM()
{
```

```
        GetShops();  
    }  
  
    private async void GetShops()  
    {  
        using (HttpClient client = new HttpClient())  
        {  
            HttpResponseMessage response = await  
client.GetAsync("http://localhost:15269/api/shop");  
            if (response.IsSuccessStatusCode)  
            {  
                string json = await response.Content.ReadAsStringAsync();  
                Shops = JsonConvert.DeserializeObject<ObservableCollection<Shop>>(json);  
            }  
        }  
    }  
  
    private ObservableCollection<Shop> _shops;  
    public ObservableCollection<Shop> Shops  
    {  
        get { return _shops; }  
        set { _shops = value; OnPropertyChanged("Shops"); }  
    }  
  
    private Shop _selectedShop;  
    public Shop SelectedShop  
    {  
        get { return _selectedShop; }  
        set { _selectedShop = value; OnPropertyChanged("SelectedShop"); }  
    }
```

Het ophalen van de shops lijkt op het eerste zicht complex omdat er een `async` method moet aangemaakt worden. De method `GetShops()` zal via de `HttpClient` class een `Get` request doen naar de service en de data die wordt teruggegeven in JSON formaat verwerken tot instanties van de `Shop` class.

In de volgende stap moet er functionaliteit voor de knoppen geschreven worden. Dit zal gedaan worden via `Commands`. Er wordt voor elke actie een property van het type `ICommand` aangemaakt die een nieuwe instantie van de class `RelayCommand` zal teruggeven in de getter. Als parameter voor deze nieuwe instantie wordt de naam van de uit te voeren method opgegeven.

Binnen de methods zelf wordt de nodige code geschreven. In dit geval wordt er een actie op de collection uitgevoerd en wordt een method uit het model aangeroepen:

```
public ICommand AddShopCommand  
{  
    get { return new RelayCommand(AddShop); }  
}  
  
public ICommand DeleteShopCommand  
{  
    get { return new RelayCommand(DeleteShop); }  
}  
  
public ICommand SaveShopCommand  
{  
    get { return new RelayCommand(SaveShop); }  
}  
  
public async void AddShop()  
{
```

```
Shop newShop = new Shop();
Shops.Add(newShop);

using (HttpClient client = new HttpClient())
{
    string shop = JsonConvert.SerializeObject(newShop);
    HttpResponseMessage response = await
client.PostAsync("http://localhost:15269/api/shop", new StringContent(shop,
Encoding.UTF8, "application/json"));
    if (response.IsSuccessStatusCode){
        SelectedShop = newShop;
    }
}

public async void DeleteShop()
{
    using (HttpClient client = new HttpClient())
    {
        HttpResponseMessage response = await
client.DeleteAsync("http://localhost:15269/api/shop/1");
        if (response.IsSuccessStatusCode){
            Shops.Remove(SelectedShop);
        }
    }
}

public async void SaveShop()
{
    using (HttpClient client = new HttpClient())
    {
        string shop = JsonConvert.SerializeObject(SelectedShop);
        HttpResponseMessage response = await
client.PutAsync("http://localhost:15269/api/shop", new StringContent(shop,
Encoding.UTF8, "application/json"));
    }
}
```

View: Deel 2

Nu het ViewModel klaar is kunnen de nodige bindings gemaakt worden in de view. Eerst en vooral moet er voor gezorgd worden dat er een instantie van het ViewModel wordt aangemaakt als DataContext voor de View. Dit kan als volgt gedaan worden:

```
xmlns:vm="clr-namespace:mvvmtest.ViewModel"
Title="MVVM Demo" Height="350" Width="525">
<Window.DataContext>
    <vm:ShopVM/>
</Window.DataContext>
```

Tot slot worden de properties uit het ViewModel gebind aan de properties van de UI elementen. Expression Blend kan hier een grote hulp zijn.

```
<ListBox ... ItemsSource="{Binding Shops}" SelectedItem="{Binding SelectedShop}"/>
<StackPanel Margin="8" Grid.Column="1">
    <Label Content="Name:"/>
    <TextBox Text="{Binding SelectedShop.Name}"/>
    <Label Content="Address:"/>
    <TextBox Text="{Binding SelectedShop.Address}"/>
```

```
<Label Content="ZipCode:"/>
<TextBox Text="{Binding SelectedShop.ZipCode}" />
<Label Content="City:"/>
<TextBox Text="{Binding SelectedShop.City}" />
```

Ook voor de knoppen worden de nodige bindings gemaakt met de commands:

```
<StackPanel Orientation="Horizontal" Margin="0,16,0,0">
    <Button Content="Add Shop" Command="{Binding AddShopCommand}" />
    <Button Content="Delete Shop" Command="{Binding DeleteShopCommand}" />
    <Button Content="Save Changes" Command="{Binding SaveShopCommand}" />
</StackPanel>
```

Conclusie demo

Deze eenvoudige demo toont de werking van MVVM. Er werd een model gemaakt met de nodige properties en methods. Daarna een view en tot slot een.viewmodel om deze aan elkaar vast te maken. Hierdoor is er geen enkele rechtstreekse koppeling tussen de view en het model en kunnen beide los van elkaar aangepast worden zonder al te veel implicaties.

Een gouden regel binnen MVVM is dat er niets van code in de code behind file van de view mag staan, maar deze regel moet met een korrel zout genomen worden. Code om een animatie te starten of gelijk welke code die niet verbonden is met het model of de.viewmodel kan perfect in de code behind komen.

MVVM frameworks

Tijdens de vorige demo werden twee classes aangemaakt om support te bieden voor het ViewModel. Er bestaan echter frameworks voor .NET die een ganse boel support bieden om het MVVM pattern te gebruiken. Prism (<http://msdn.microsoft.com/en-us/library/gg406140.aspx>) en MVVM Light (<http://qalasoft.ch/mvvm/>) zijn hier twee voorbeelden van. Er bestaan natuurlijk nog vele andere en in feite kan je zelf je eigen MVVM framework maken.

Binnen deze cursus zal een deel van het MVVM light framework gebruikt worden om het gemakkelijker te maken events naar commands om te zetten en om parameters mee te geven met commands. De Visual Studio templates zullen echter niet gebruikt worden. In plaats van deze templates wordt er voor gekozen om zelf een template samen te stellen.

Het installeren van het MVVM Light framework kan je opnieuw via de Package Manager doen.

Advanced commands

Meegeven van een parameter

In de eerste demo werden verschillende commands aangemaakt, maar de methods die werden opgeroepen hadden geen parameter nodig. Soms zal het voorkomen dat dit wel het geval is.

Om deze situatie af te handelen wordt er gebruik gemaakt van de RelayCommand class uit het MVVM Light framework. Onze eigen class mag dus uit het project worden verwijderd.

De volgende code toont aan hoe een parameter van het type string kan worden meegegeven:

```
public ICommand DemoCommand
{
    get { return new RelayCommand<string>(Demo); }
}
```

```
public void Demo(string something)
{
```

```
        Console.WriteLine(something);
    }
```

Het verschil met de vorige demo is dat er bij het RelayCommand een type wordt meegegeven tussen <> net zoals dat bij een ObservableCollection wordt gedaan. Bij de method zelf wordt de parameters gewoon tussen de haakjes mee gegeven zoals bij gelijk welke andere method.

In XAML is het mogelijk om bij een command een CommandParameter mee te geven. Deze zal dan ingevuld worden in de parameter bij de method:

```
<Button Command="{Binding DemoCommand}" CommandParameter="Een voorbeeld" ... />
```

In dit voorbeeld is de parameter nu een stuk letterlijke tekst, maar dat kan natuurlijk ook een waarde zijn die via databinding wordt ingevuld.

Event omzetten naar command

Niet alle events die moeten afgehandeld worden kunnen rechtstreeks via een command worden gedaan zoals een click op een button of een menuitem. Stel bijvoorbeeld dat een bepaalde method maar mag uitgevoerd worden bij een dubbele klik op een button of op een MouseEnter van een rechthoek.

Voor deze gevallen bevat het MVVM light framework het EventToCommand behavior. Behaviors zijn kleine stukjes code die een bepaalde actie kunnen uitvoeren zoals het starten van een animatie of implementeren van drag & drop functionaliteit.

```
xmlns:cmd="clr-  
namespace:GalaSoft.MvvmLight.Command;assembly=GalaSoft.MvvmLight.Platform"  
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity""
```

Deze namespaces verwijzen naar de DLL's en zijn nodig zodat de functionaliteit er van in XAML kan worden gebruikt.

Deze demo toont de code hoe het MouseEnter event van een Rectangle kan worden opgevangen. Een Rectangle heeft standaard geen attribuut Command, dus is de behavior EventToCommand nodig:

```
<Rectangle Width="100" Height="100" Fill="Green">  
    <i:Interaction.Triggers>  
        <i:EventTrigger EventName="MouseEnter">  
            <cmd:EventToCommand CommandParameter="over" Command="{Binding DemoCommand}"/>  
        </i:EventTrigger>  
    </i:Interaction.Triggers>  
</Rectangle>
```

In dit geval is er maar 1 EventTrigger, namelijk voor het MouseEnter event, maar dit kan ook een collectie van triggers zijn. Binnen de trigger wordt het MouseEnter event uit het attribuut EventName gekoppeld aan het Command DemoCommand. Indien nodig is het hier ook mogelijk om een parameter mee te geven.

Navigatie binnen MVVM

Tot nu toe bestonden alle demo's en oefeningen uit een project met één Window en alle controls werden op dat Window geplaatst. Dit is natuurlijk niet volledig realistisch. De meeste applicaties zullen bestaan uit meerdere onderdelen die niet allemaal binnen hetzelfde Window passen. Het is echter wel een guideline om binnen een WPF applicatie slechts 1 Window te hebben en de content telkens dynamisch aan te passen via User Controls.

Een User Control is heel gelijkaardig aan een Window met als enig verschil dat er geen rand rond getekend staat en dat het niet mogelijk is om een title in te stellen. Een WPF applicatie zal dus normaal meestal bestaan uit een Window dat een menu bevat om te navigeren naar de verschillende onderdelen en een container waar de user control in geplaatst kan worden.

De volgende demo illustreert dit concept.

Navigatie binnen MVVM: demo

Eerst en vooral worden binnen de map View enkele nieuwe user controls aangemaakt. Dit kan gedaan worden door rechts te klikken op de map en te kiezen voor Add -> New Item... en te kiezen voor een User Control. Zoals je kan zien is dit een gewone XAML file die heel gelijkaardig is aan een Window.

```
<UserControl x:Class="NavigationApp.View.PartOne"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        </Grid>
</UserControl>
```

Voor elke view moet er nu ook een ViewModel aangemaakt worden binnen de map van de ViewModels. Het is echter aan te raden dat alle ViewModels van hetzelfde type zijn zodat er een lijst van de verschillende user controls kan aangemaakt worden. Dit is een goed voorbeeld waar het werken met een interface een meerwaarde biedt. Daarom wordt er eerst een interface IPage aangemaakt met slechts één property, zijnde een naam. Alle ViewModels zullen deze interface implementeren. In code ziet dit er als volgt uit.

De interface IPage:

```
public interface IPage
{
    string Name { get; }
}
```

De ViewModel voor een bepaalde View:

```
public class PartOneVM : ObservableObject, IPage
{
    public string Name
    {
        get { return "Page One"; }
    }
}
```

De class PartOneVM erft dus over van de class ObservableObject en implementeert de interface IPage. C# laat immers maar toe om van slechts één class over te erven, maar je kan meerdere interfaces implementeren.

De kern van de navigatie is echter een class die zorgt dat het mogelijk is om in een container van het Window een nieuwe user control te plaatsen die een collectie van alle user controls bevat.

Deze class zal een property CurrentPage hebben die de user control bevat die getoond moet worden. Daarnaast zal er ook een property zijn die een collection bevat van alle user controls in het project. Tot slot is er ook nog een command nodig om een andere user control als de huidig zichtbare user control aan te geven. De code voor deze class ziet er als volgt uit:

```
public class ApplicationVM : ObservableObject
{
    public ApplicationVM()
    {
        Pages.Add(new PartOneVM());
        Pages.Add(new PartTwoVM());
    }
}
```

```
Pages.Add(new PartThreeVM());  
  
        CurrentPage = Pages[0];  
    }  
  
    private IPAGE currentPage;  
    public IPAGE CurrentPage  
{  
        get { return currentPage; }  
        set { currentPage = value; OnPropertyChanged("CurrentPage"); }  
    }  
  
    private List<IPAGE> pages;  
    public List<IPAGE> Pages  
{  
        get {  
            if (pages == null)  
                pages = new List<IPAGE>();  
            return pages;  
        }  
    }  
  
    public ICommand ChangePageCommand  
{  
        get { return new RelayCommand<IPAGE>(ChangePage); }  
    }  
  
    private void ChangePage(IPAGE page)  
{  
        CurrentPage = page;  
    }  
}
```

Tot slot moet deze class gebind worden aan de DataContext van het Window. Dan is het mogelijk om een overzicht van alle user controls in een ItemsControl weer te geven en de inhoud van de geselecteerde user control in een ContentControl. Let ook op de namespaces v en vm die werden aangemaakt om te verwijzen naar de Views en ViewModels.

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
xmlns:vm="clr-namespace:NavigationApp.ViewModel"  
xmlns:v="clr-namespace:NavigationApp.View"  
Title="MainWindow" Height="350" Width="525">  
<Window.DataContext>  
    <vm:ApplicationVM />  
</Window.DataContext>
```

De binding voor de ItemsControl zal eerst de property Pages binden aan de ItemsSource property van de ItemsControl. Daarna wordt er een ItemTemplate aangemaakt om aan te geven hoe alle elementen in het menu er uit moeten zien. In dit geval werd er gekozen voor buttons.

Aan de content property van de button wordt de Name property gebind die voor elk ViewModel werd ingevuld. Deze zal dan ook weergegeven worden in de button. De binding van het command is iets complexer. Aangezien het command ChangePageCommand een onderdeel is van de ApplicationVM class en niet van het ViewModel van een user control moet er verwezen worden naar de DataContext property van het Window. Dit wordt hier gedaan door de RelativeSource in te vullen. De parameter voor het command zal de het geselecteerde element zijn.

De content van de ContentControl wordt tot slot gebind aan het geselecteerde ViewModel.

```
<ItemsControl ItemsSource="{Binding Pages}">
```

```
<ItemsControl.ItemTemplate>
    <DataTemplate>
        <Button Content="{Binding Name}" Command="{Binding
DataContext.ChangePageCommand, RelativeSource={RelativeSource AncestorType={x:Type
Window}}}" CommandParameter="{Binding}"/>
    </DataTemplate>
</ItemsControl.ItemTemplate>
</ItemsControl>
<ContentControl Grid.Column="1" Content="{Binding CurrentPage}"/>
```

Bij het testen van deze demo zal voorlopig enkel de `ToString()` method van elk ViewModel worden getoond ipv de inhoud van de View. Om dit te voorkomen moet er nog voor elke user control een DataTemplate aangemaakt worden in de resources van het Window.

```
<Window.Resources>
    <DataTemplate DataType="{x:Type vm:PartOneVM}">
        <v:PartOne/>
    </DataTemplate>
    <DataTemplate DataType="{x:Type vm:PartTwoVM}">
        <v:PartTwo/>
    </DataTemplate>
    <DataTemplate DataType="{x:Type vm:PartThreeVM}">
        <v:PartThree/>
    </DataTemplate>
</Window.Resources>
```

Bij elke DataTemplate wordt het DataType ingesteld op dat van het ViewModel. Binnen de DataTemplate wordt dan een instantie weergegeven van de overeenkomstige view. Hierdoor wordt nu wel de XAML van de user control getoond bij het selecteren van een item uit het menu.

Oefeningen MVVM

Oefening 1

In deze oefening worden 2 project templates aangemaakt die voor alle resterende oefeningen kunnen gebruikt worden. De eerste project template is voor de API en zal als extra enkel de class Database bevatten.

De tweede template is voor het WPF project en deze zal de volgende extra functionaliteiten bevatten:

- De folders View en ViewModel zijn aangemaakt
- De nodige Nuget packages voor HttpClient, JSON en MvvmLight zijn toegevoegd
- De nodige functionaliteit voor navigatie is aanwezig
- Plaats alvast een usercontrol op het window
- De class ObservableObject is aanwezig

Maak een project aan die al deze functionaliteiten bevat, maar ook niets meer of minder. Je kan de code uit de theorie hierboven gebruiken om te controleren of je alle functionaliteiten hebt.

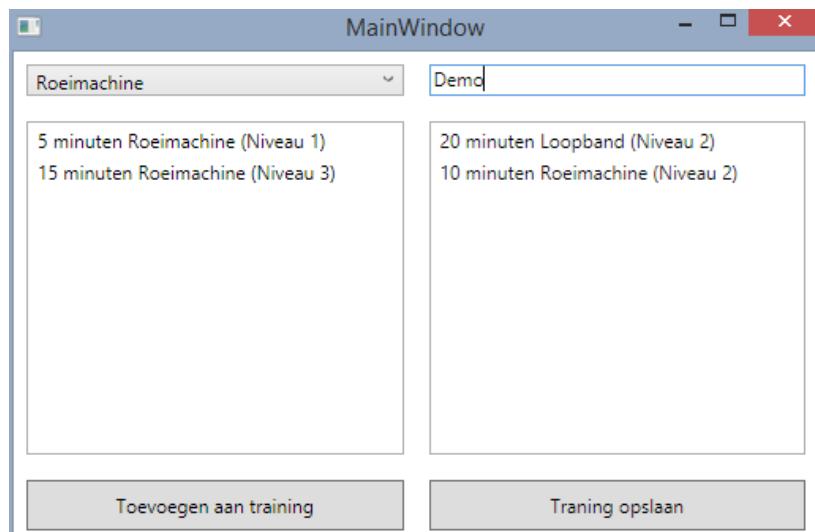
Als de projecten klaar zijn, lees je de verschillende stappen op <http://msdn.microsoft.com/en-us/library/xkh1wdx8.aspx> en maak je 2 project templates aan. De eerste voor de API en de tweede voor het WPF project.

Test de templates door een nieuwe solution aan te maken op basis van de templates. Op die manier hebben we in feite ons eigen MVVM framework aangemaakt met als basis MVVM light.

Oefening 2

Bij deze oefening maken we een applicatie om trainingssessies in de fitness bij te houden. Bij het bronmateriaal is er een database die volgende info bevat:

- Equipment: de naam en ID van fitnessstoestellen
- Workout: het gebruik van een fitnessstoestel voor een bepaalde tijd op een bepaald niveau
- Training: de naam van de gebruiker en een lijst van workouts. Voor het gemak worden deze gescheiden door een punt-komma in 1 veld opgeslagen.



Als eerste stap worden de nodige projecten aangemaakt:

- Maak eerst een blanco solution aan
- Maak vervolgens een class library aan
- Maak daarna een Web API aan op basis van de template
- Maak tot slot een WPF project aan op basis van de template
- Leg de nodige referenties en stel de startup projecten in zodat zowel de Web API als het WPF project standaard gestart worden.
- Start tot slot het project eens op.

De volgende stap is het aanmaken van het model. Maak in de class library de nodige classes aan:

- Equipment
- Workout: de property Equipment is van het type Equipment
- Trainingsession: de lijst met workouts is van het type IList<Workout>

Maak daarna de nodige data accessors aan. Je doet dit in de map Models van je Web API. Vergeet ook niet om je connectionstring in te stellen en maak gebruik van de class Database. Je hebt de volgende methods nodig:

- EquipmentDA:
 - GetEquipments() die alle toestellen uit de database haalt
 - GetEquipment(int id) die een toestel op basis van het ID uit de database haalt
- WorkoutDA:
 - GetWorkoutsByEquipment(int id): haalt op basis van het ID van een toestel alle workouts voor dit toestel op. Gebruik de method GetEquipment(int id) om het toestel van de workout in te vullen.
- TrainingsessionDA:
 - SaveTrainingSession(Trainingsession ts) slaat een training op in de database.

Maak nu ook de nodige controllers aan om je methods uit de data accessors beschikbaar te stellen via de API. Daarna kan je in de browser al eens testen of het lukt om de toestellen en hun workouts op te halen.

Het volgende deel van de oefening is het maken van view in de user control. De XAML code voor de view ziet er als volgt uit:

```
<UserControl x:Class="nmct.ba.week7.oefening2.ui.View.PageOne"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    xmlns:vm="clr-namespace:nmct.ba.week7.oefening2.ui.ViewModel"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.DataContext>
        <vm:PageOneVM/>
    </UserControl.DataContext>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="36"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="48"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <ComboBox Margin="8"/>
        <ListBox Grid.Row="1" Margin="8"/>
        <Button Margin="8" Grid.Row="2" Content="Toevoegen aan training"/>
        <TextBox Margin="8" Grid.Column="1"/>
        <ListBox Margin="8" Grid.Column="1" Grid.Row="1"/>
        <Button Margin="8" Grid.Row="2" Grid.Column="1" Content="Training opslaan"/>
    </Grid>
</UserControl>
```

Schrijf nu in PageOneVM de nodige code om:

- De toestellen op te halen
- Het geselecteerde toestel bij te houden
- De workouts per toestel op te halen
- De geselecteerde workout bij te houden
- Een training bij te houden
- Een command om een workout bij een training te plaatsen
- Een command om de training op te slaan in de database

Data validatie

Intro

Het is best om er van uit te gaan dat de gebruiker stommiteiten zal uithalen bij het gebruiken van een applicatie. Een goede user interface probeert dit te voorkomen en moet dit zeker kunnen opvangen. Een belangrijk onderdeel hier is het valideren van de ingegeven data van een gebruiker via data validation. WPF voorziet hier verschillende mogelijkheden die in de volgende secties worden besproken.

Voorkomen is echter altijd beter dan genezen. Kies daarom de juiste controls en zorg er voor dat de gebruiker zo weinig mogelijk tekst hoeft in te tikken. Voorbeelden hiervan zijn: gebruiken van radiobuttons om te kiezen tussen 2 opties, gebruiken van een combobox of autoaanvulling om een land of stad te selecteren, gebruik van datepicker om een datum in te vullen,...

Demo: conversie van integer naar string

Een eerste demo zal de Text property van een TextBox binden aan een de leeftijd van een object Person. Bij het invullen van een niet-numerieke waarde zal dit een error opleveren:

```
<TextBox Margin="8" Text="{Binding Person.Age, UpdateSourceTrigger=PropertyChanged}"/>
```



Figuur 20: automatische validatie door WPF

Dit wordt standaard in WPF weergegeven door een rode rand rond de textbox te plaatsen. In het output panel verschijnt de volgende mededeling:

```
System.Windows.Data Error: 7 : ConvertBack cannot convert value '31a' (type 'String'). BindingExpression:Path=Person.Age;
at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolean parse)
at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
at System.String.System.IConvertible.ToInt32(IFormatProvider provider)
at System.Convert.ChangeType(Object value, Type conversionType, IFormatProvider provider)
at MS.Internal.Data.SystemConvertConverter.ConvertBack(Object o, Type type, Object parameter, CultureInfo culture)
at System.Windows.Data.BindingExpression.ConvertBackHelper(IValueConverter converter, Object value, Type sourceType, C
The thread '<No Name>' (0x1b50) has exited with code 0 (0x0).
The thread '<No Name>' (0x12ac) has exited with code 0 (0x0).
```

Figuur 21: foutmelding bij binding error

Deze foutmelding toont aan dat er geprobeerd wordt om de string om te zetten naar een integer, maar dat deze conversie fout loopt. Het feit dat er automatisch een rode rand rond de textbox wordt getoond is natuurlijk mooi meegenomen, maar op dit punt krijgt de gebruiker geen enkele feedback over wat hij precies fout heeft gedaan. Daarnaast is er ook nood aan de mogelijkheid om eigen validatieregels te schrijven zoals bijvoorbeeld voor het valideren van een e-mailadres.

ErrorTemplate

WPF voorziet de mogelijkheid om een eigen template te ontwikkelen die getoond moet worden bij foutieve input. Elke control kan gebruik maken van de attached property Validation.ErrorTemplate waarbij een ControlTemplate kan aangemaakt worden.

Deze ControlTemplate zal in bijna alle gevallen een AdornedElementPlaceholder bevatten (<http://msdn.microsoft.com/en-us/library/system.windows.controls.adornedelementplaceholder.aspx>). Dit stelt het element voor in de ControlTemplate waar de validatie van op toepassing is en dat de nodige opmaak moet krijgen (adorned = opgesmukt, opgesierd). De ControlTemplate kan er dus bijvoorbeeld als volgt uit zien:

```
<ControlTemplate x:Key="ErrorTemplate">
<StackPanel>
    <Border BorderBrush="Red" BorderThickness="2">
        <AdornedElementPlaceholder/>
    </Border>
    <TextBlock Foreground="Red" Text="{Binding [0].ErrorContent}"/>
</StackPanel>
</ControlTemplate>
```

Dit voorbeeld zal er voor zorgen dat er een dikke rode rand rond de textbox komt en dat er een foutomschrijving onder de textbox staat. De text van deze textblock wordt gebind aan het eerste element uit Validation.Errors property (<http://msdn.microsoft.com/en-us/library/system.windows.controls.validation.errors.aspx>) die een collectie van errors bevat. De property ErrorContent geeft dan een beschrijving van de error.

Een andere optie is om via de Trigger Validation.HasError een Style te voorzien voor het element met een error. Op die manier kan je enkele properties zoals de tekstkleur van de textbox aanpassen, maar is het niet mogelijk om elementen toe te voegen aan de template.

```
<Style x:Key="TextboxError" TargetType="{x:Type TextBox}">
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="Foreground" Value="Red"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

Het is natuurlijk ook mogelijk om beide te combineren. Het toepassen van de template en de style verloopt dan als volgt:

```
<TextBox Margin="8" Text="{Binding Person.Age, UpdateSourceTrigger=PropertyChanged}"
    Validation.ErrorTemplate="{StaticResource ErrorTemplate }"
    Style="{StaticResource TextboxError}"/>
```

Validation rules

Tijdens de vorige demo werd de foutmelding automatisch getoond indien het niet mogelijk was om de input om te zetten naar een getal. Validatie is echter veel meer dan dat. Het moet dus mogelijk zijn om validation rules uit te schrijven voor verschillende cases. Een mooi voorbeeld hier is controleren of een leeftijd tussen bepaalde grenzen ligt.

Een validation rule zal overerven van de abstracte class ValidationRule (<http://msdn.microsoft.com/en-us/library/system.windows.controls.validationrule.aspx>) en zal de method Validate() overschrijven. Deze method zal een ValidationResult teruggeven.

```
class AgeRule : ValidationRule
{
    public override ValidationResult Validate(object value,
System.Globalization.CultureInfo cultureInfo)
    {
        ValidationResult result;
        int i;

        if(Int32.TryParse((string)value,out i))
            result = new ValidationResult(true,null);
        else
            result = new ValidationResult(false, "De leeftijd moet een getal zijn");

        return result;
    }
}
```

Een nieuwe instantie van ValidationResult bevat 2 parameters. De eerste is een boolean om aan te geven of het resultaat valid is of niet. De tweede parameter bevat dan de ErrorContent of foutbericht. In de volgende stap moet de validationrule toegevoegd worden aan de binding.

```
<TextBox Margin="8"
    Validation.ErrorTemplate="{StaticResource ErrorTemplate }"
    Style="{StaticResource TextboxError}">
<TextBox.Text>
    <Binding Path="Person.Age" UpdateSourceTrigger="PropertyChanged">
```

```
<Binding.ValidationRules>
    <rules:AgeRule/>
</Binding.ValidationRules>
</Binding>
</TextBox.Text>
</TextBox>
```

Er wordt een nieuwe regel toegevoegd aan de binding en de foutmelding zal overgenomen worden. Op die manier is het mogelijk om een eigen foutbericht door te geven naar de View.



Figuur 22: een eigen foutbericht weergeven

Daarnaast is het ook mogelijk om nog extra parameters mee te geven met een ValidationRule en meerdere testen uit te voeren op een bepaald veld. Zo kan bijvoorbeeld gecontroleerd worden of een leeftijd tussen de 3 en 18 jaar valt:

Eerst worden er binnen de ValidationRule class 2 properties aangemaakt

```
public int Minimum { get; set; }
public int Maximum { get; set; }
```

Daarna wordt in de methode Validate() gecontroleerd of de leeftijd wel binnen de correcte grenzen valt en wordt er een foutbericht op maat terug gestuurd.

```
int i;

if(Int32.TryParse((string)value,out i))
{
    if (i < Minimum)
        return new ValidationResult(false, "De leeftijd is te laag");
    else if (i > Maximum)
        return new ValidationResult(false, "De leeftijd is te hoog");
    else if (i >= Minimum && i <= Maximum)
        return new ValidationResult(true, null);
    else return new ValidationResult(false, "De leeftijd is niet correct");
}
else
    return new ValidationResult(false, "De leeftijd moet een getal zijn");
```

Tot slot worden de parameters ook ingevuld bij de binding:

```
<Binding.ValidationRules>
    <rules:AgeRule Minimum="3" Maximum="18"/>
</Binding.ValidationRules>
```

De foutbericht zal nu wijzigen afhankelijk van de foute waarde dit werd ingegeven.

Deze ValidationRules zijn een heel krachtig principe, maar hebben hun beperkingen. Zo moet de binding voor elk element voluit geschreven worden wat heel langdradige en moeilijk te onderhouden XAML code tot gevolg kan hebben. Een ander nadeel is dat er een nieuwe ValidationRule moet aangemaakt worden voor elk type van validatie. Dit kan ook tot moeilijk te onderhouden code leiden.

IDataErrorInfo

De IDataErrorInfo interface (<http://msdn.microsoft.com/en-us/library/system.componentmodel.idataerrorinfo.aspx>) biedt een alternatief voor de

ValidationRules. Het implementeren van deze interface op het model zal resulteren in 2 extra properties.

```
class Person : IDataErrorInfo
{
    public int Age { get; set; }

    public string Error
    {
        get { throw new NotImplementedException(); }
    }

    public string this[string columnName]
    {
        get { throw new NotImplementedException(); }
    }
}
```

De eerste property heeft als naam Error en zal een string teruggeven. Deze string bevat een foutbericht om aan te geven wat er niet correct is aan het volledige object. Binnen deze cursus zal deze property gewoon null terug geven ofwel een algemene boodschap.

De tweede property zal afhankelijk van de columnName (of in ons geval property name) controleren of een bepaalde waarde al dan niet valid is. Indien dit niet het geval is wordt een string teruggegeven met een specifieke foutbericht, in het andere geval wordt gewoon een lege string teruggegeven om aan te geven dat er geen fouten werden gevonden.

Een voorbeeld kan een object Person zijn met een naam, voornaam, leeftijd en gebruikersnaam. De leeftijd moet groter of gelijk zijn dan 18 jaar terwijl de gebruikersnaam steeds een @ teken en een . moet bevatten. De naam en de voornaam moeten minstens 2 karakters bevatten. Het model ziet er dan als volgt uit:

```
class Person : IDataErrorInfo
{
    public string Name { get; set; }
    public string FirstName { get; set; }
    public int Age { get; set; }
    public string UserName { get; set; }

    public string Error
    {
        get { return "Het object is niet valid"; }
    }

    public string this[string columnName]
    {
        get
        {
            switch(columnName)
            {
                case "Name":
                    if(Name != null && Name.Length <= 2)
                        return "De naam moet minstens 2 karakters bevatten";
                    break;
                case "FirstName":
                    if(FirstName != null && FirstName.Length <= 2)
                        return "De voornaam moet minstens 2 karakters bevatten";
                    break;
                case "Age":
                    if (Age < 18)
                        return "De leeftijd moet minstens 18 jaar zijn";
            }
        }
    }
}
```

```

        break;
    case "UserName":
        if (UserName != null && UserName.IndexOf('@') == -1)
            return "De username moet een @ teken bevatten";
        if (UserName != null && UserName.IndexOf('.') == -1)
            return "De username moet een . bevatten";
        break;
    }

    return String.Empty;
}
}
}
}
```

De properties worden gewoon aangemaakt en door het implementeren van de interface worden 2 properties toegevoegd. Binnen de property `this[string columnName]` wordt gecontroleerd over welke property het precies gaat.

Binnen elke case wordt er een controle gedaan die specifiek is voor een bepaalde property en telkens wordt een aangepaste foutbericht weergegeven. Daarnaast wordt ook gecontroleerd of een element verschillend is van null aangezien dit anders een error zou geven. Als geen enkele case een foutbericht heeft terug te geven dan geven we een lege string terug om te laten weten dat alle elementen perfect valid zijn.

De XAML ziet er dan als volgt uit voor alle properties, enkel de binding zal wijzigen afhankelijk van de property:

```
<TextBox Text="{Binding Person.Name, UpdateSourceTrigger=PropertyChanged,
ValidatesOnDataErrors=True}"
Margin="16"
Validation.ErrorTemplate="{StaticResource ErrorTemplate }"
Style="{StaticResource TextboxError}"/>
```

Belangrijk hier is om te weten dat het attribuut `ValidatesOnDataErrors` op true moet staan, anders zal de validatie niet worden uitgevoerd. Ook opmerkelijk is dat bij het runnen van de applicatie er standaard geen plaats wordt gereserveerd om eventuele foutmeldingen weer te geven. Je moet hier dus rekening mee houden bij het ontwerpen van je design. Vandaar dat de margin hier op 16 pixels staat. Zo heeft de `ErrorTemplate` voldoende plaats om een foutmelding te plaatsen onder elke textbox.



Figuur 23: valideren van properties

Data annotations

Het is net als in ASP.NET MVC mogelijk om data annotations te gebruiken bij de validatie. Data annotations zijn attributes die boven de properties worden geplaatst met daarbij de regels om een property valid te krijgen. Bijvoorbeeld voor de property Name

```
[Required(ErrorMessage = "De naam is verplicht")]
[RegularExpression(@"^a-zA-Z'-\s]{1,50}$", ErrorMessage = "Er zijn geen speciale
tekens toegelaten")]
[StringLength(50, MinimumLength = 3, ErrorMessage = "De naam moet tussen de 3 en 50
karakters bevatten")]
public string Name { get; set; }
```

Deze data annotations geven aan dat het veld verplicht moet ingevuld worden, dat er geen speciale tekens worden toegelaten en dat de lengte tussen de 3 en de 50 karakters moet liggen. Een volledig overzicht van alle mogelijke data annotations is te vinden op <http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx>. In feite is het mogelijk om met deze data annotations alle mogelijke validaties te doen, vooral omdat het mogelijk is om Regular Expressions te gebruiken. Het gebruik van Regular Expressions maakt niet echt deel uit van de cursus, maar een handig overzicht is te vinden op <http://www.regular-expressions.info/>

Het verschil met ASP.NET MVC is echter dat het plaatsen van data annotations niet automatisch er voor zal zorgen voor validatie. Er zijn veel verschillende mogelijkheden binnen WPF, maar de eenvoudigste is het implementeren van de IDataErrorInfo interface zoals in de vorige sectie.

Binnen de this[string columnName] property kan dan binnen een try catch blok gevraagd worden om de property te valideren:

```
public string this[string columnName]
{
    get
    {
        try
        {
            object value = this.GetType().GetProperty(columnName).GetValue(this);
            Validator.ValidateProperty(value, new ValidationContext(this, null, null) {
                MemberName = columnName });
        }
        catch (ValidationException ex)
        {
            return ex.Message;
        }
        return String.Empty;
    }
}
```

De regel `object value = this.GetType().GetProperty(columnName).GetValue(this);` haalt de waarde op voor de property die moet gevalideerd worden. Verdere info over de method GetProperty() is terug te vinden op <http://msdn.microsoft.com/en-us/library/kz0a8sxy.aspx>

Vervolgens wordt de method ValidateProperty van de class Validator opgeroepen (<http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.validator.validateproperty.aspx>). Deze method neemt de huidige waarde van een property als eerste argument en een ValidationContext als tweede argument. Bij de ValidationContext moet het huidige object (dus this) worden meegegeven, de andere parameters gebruiken we hier niet en mogen dus als waarde null krijgen. Tot slot wordt aangegeven over welke property de validatie zal gaan. Verdere info over de ValidationContext is te vinden op <http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.validationcontext.aspx>.

Vanaf hier werkt de validatie op exact dezelfde manier als bij het gebruik van de IDataErrorInfo interface zonder de data annotations.

Enablen en disablen van controls

Het zou logisch zijn dat je niet op een "save" knop kan klikken op het moment dat een model niet valid is. Als we nog een knop toevoegen aan onze demo zie je dat dit nog een stuk extra code vergt.

Eerst en vooral voeg je een method IsValid() toe op het model die een boolean zal teruggeven

```
public bool IsValid()
{
    return Validator.TryValidateObject(this, new ValidationContext(this, null, null),
null, true);
}
```

Binnen deze method wordt gebruik gemaakt van de method TryValidateObject() die zal bepalen of alle validatie regels werden gevuld en op die manier true of false zal teruggeven.

In het Viewmodel kan deze method nu gebruikt worden bij het RelayCommand. Zoals gezien bij de theorie over commands is het mogelijk om controls automatisch te enablen en disablen. Dit is ook mogelijk bij het RelayCommand door een extra parameter mee te geven, zijnde de method IsValid(). Op die manier zal de button die het command oproept enkel maar beschikbaar zijn als het volledige model valid is.

```
public ICommand AddPersonCommand
{
    get { return new RelayCommand(AddPerson, Person.IsValid); }
}
```