# MVVM architectuur

Contents

[MVVM architectuur 1](#_Toc433745360)

[MVVM architectuur: duiding 2](#_Toc433745361)

[Visual Studio Solution architectuur 2](#_Toc433745362)

[Models project invulling 4](#_Toc433745363)

[ModelBaseClass 4](#_Toc433745364)

[ModelBaseClass en het EntityFramework 5](#_Toc433745365)

[ModelBaseClass: beschouwingen 5](#_Toc433745366)

[Models.Core folder 5](#_Toc433745367)

[Ontwikkeling van de Model klasses 6](#_Toc433745368)

[Extra Models folders 6](#_Toc433745369)

[Flickr Models 6](#_Toc433745370)

[Data project invulling 6](#_Toc433745371)

[FlickrRepository 7](#_Toc433745372)

[FlickrRepository: voorbeeld implementatie 7](#_Toc433745373)

[FlickrRepository: testen van de implementatie 8](#_Toc433745374)

[World info 10](#_Toc433745375)

[World Models 10](#_Toc433745376)

[World Repositories 10](#_Toc433745377)

[ContinentRepository 10](#_Toc433745378)

[CountryRepository 11](#_Toc433745379)

[CityRepository 11](#_Toc433745380)

[RepositoryBaseClass<T> 11](#_Toc433745381)

[ContinentRepository Bis 12](#_Toc433745382)

[CountryRepository Bis 13](#_Toc433745383)

[CityRepository 15](#_Toc433745384)

[ViewModel en View uitwerking 15](#_Toc433745385)

[Navigatie 15](#_Toc433745386)

[Grote foto’s tonen 15](#_Toc433745387)

[Navigatie in MVVM: ApplicationViewModel 17](#_Toc433745388)

[ApplicationViewModel 17](#_Toc433745389)

[MainPage.xaml aanpassingen 18](#_Toc433745390)

[PageViewModel aanpassingen 18](#_Toc433745391)

[Services 18](#_Toc433745392)

[ViewModelLocator 18](#_Toc433745393)

[Dependency injection (DI) 20](#_Toc433745394)

[Inversion Of Control (IoC) 21](#_Toc433745395)

[Uiteindelijke opkuis 23](#_Toc433745396)

[Conclusie 23](#_Toc433745397)

[ArcGis webservice (appendix) 23](#_Toc433745398)

# MVVM architectuur: duiding

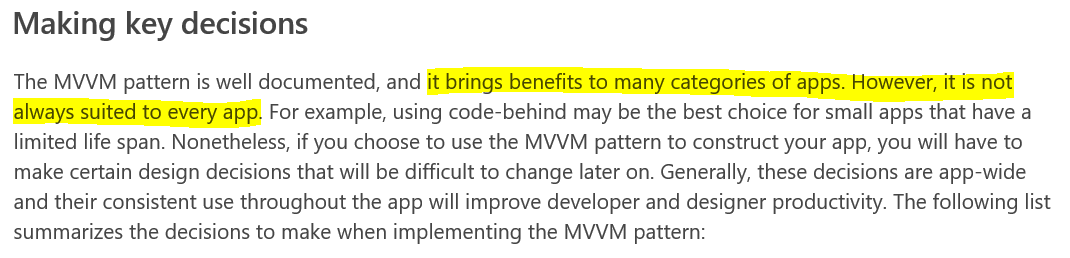
De voorbije lessen werden technische topics geïntroduceerd: Resources, Styles, DataBinding en CommandBinding. Deze leiden in een MVVM structuur tot een oplossing die momenteel echter niet super gestructureerd werd aangebracht. Het is de bedoeling van deze tekst om een object georiënteerde oplossing te presenteren in een goed gestructureerde Visual Studio Solution.

De structuur die we voorstellen in deze tekst is slechts een voorbeeld. Op een aantal punten zal hier afhankelijk van de ontwikkelaar en het project zeker vanaf geweken worden.

In de loop van deze tekst komen ook de termen Dependency Injection (DI) en Inversion of Control (IOC) uitgebreid aan bod.

De kapstok voor deze tekst is een oefening die vanuit twee webservices ([flickr](https://www.flickr.com/) en de eigen WorldWS) informatie en foto’s presenteert aan de eindgebruiker. Uiteraard maken we terug gebruik van een UWP oplossing.

Tot slot: MVVM is niet voor elke toepassing de meest geschikte oplossing: uit [Using the Model-View-ViewModel (MVVM) pattern in a Windows Store business app using C#, XAML, and Prism](https://msdn.microsoft.com/en-us/library/windows/apps/xx130657.aspx" \l "UIInteraction):



# Visual Studio Solution architectuur

De voorbije lessen werkten we in één project waarbinnen verschillende folders werden aangemaakt. Het kost weinig moeite om een meer gestructureerde oplossing op te zetten waarbinnen de Model- en Repository klasses in een apart project worden geplaatst. Het voordeel hiervan is dat deze beide in een ander project kunnen gerecupereerd worden.

Voor de oefening volgen we volgende stappen:

* Maak een blank solution met naam ContinentLandStadFotos
  + We zullen de WorldWS webservice gebruiken om continenten, landen en steden te selecteren, en voor deze foto’s op te zoekeno p flickr.
* Voeg 3 projecten toe aan deze blank solution:
  + ContinentLandStadFotos.UWP: een universal blank app, welke de UI zal bevatten
    - Voorzie de folders Pages en ViewModels
  + ContinentLandStadFotos.Data: een class library waarbinnen de Repositories zullen geprogrammeerd worden
    - Verwijder Class1
  + ContinentLandStadFotos.Models: een class library waarbinnen de Repositories zullen geprogrammeerd worden
    - Verwijder Class1

Bemerk dat de Data en Models projecten door hun naamgeving een namespace hebben waarin twee qualifiers gescheiden zijn door middel van een punt, net zoals we dit hebben bij het gebruik van een folder.

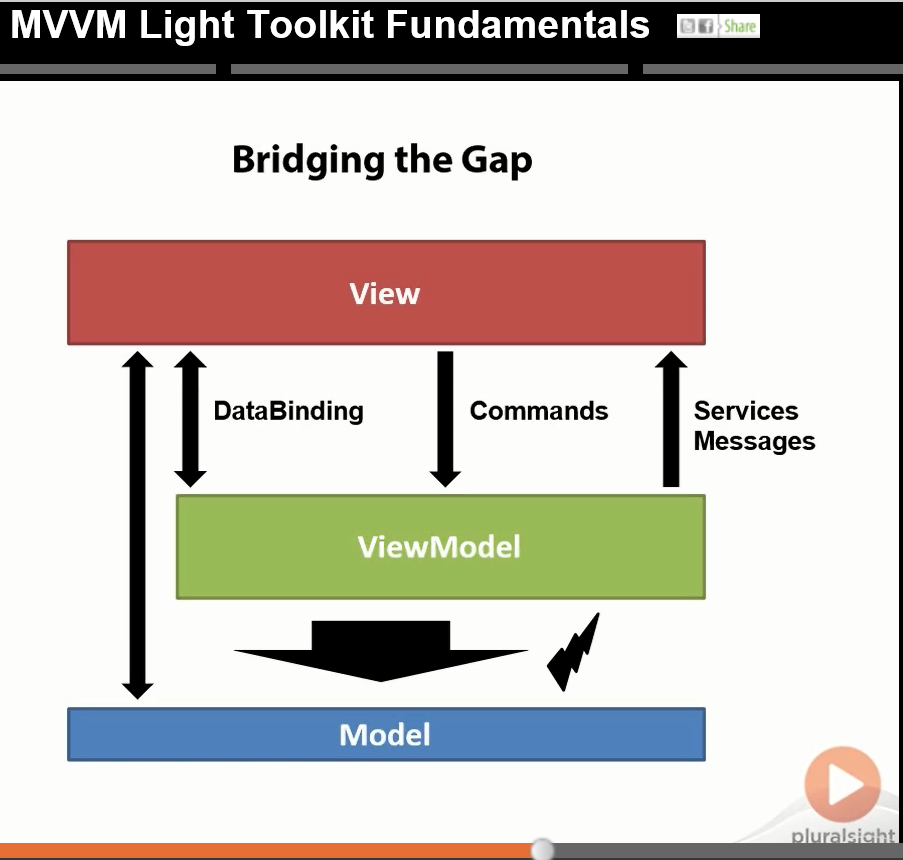
Vergeet niet de nodige referenties aan het project toe te voegen:

* Het UWP project refereert beide andere projecten
  + Omdat vanuit het UWP project zowel Repository als Model klasses gebruikt zullen worden
* Het Data project refereert het Models project
  + Omdat vanuit het Data project Model klasses gebruikt zullen worden

Ter informatie: circulaire referenties zijn niet toegelaten!

# Models project invulling

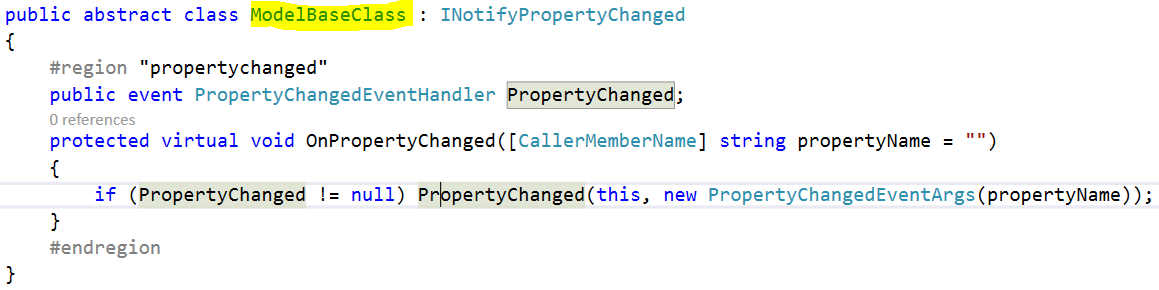
Hoewel veel MVVM beschrijvingen directe communicatie tussen View en Model vermijden (beide communiceren wel met het ViewModel) leunt volgende figuur uit [de PluralSight cursus ‘MVVM light toolkit fundamentals’](http://www.pluralsight.com/training/player?author=laurent-bugnion&name=mvvm-light-m1-introduction&mode=live&clip=0&course=mvvm-light-toolkit-fundamentals) dichter aan bij de de realiteit:



Bemerk dat de Model klasses via events wijzigingen kunnen mededelen aan het ViewModel en DataBinding ondersteunen via de INotifyPropertyChanged interface. Dit vermijdt de nood aan een specifieke ViewModel klasse voor elke Model klasse in de meeste scenario’s.

## ModelBaseClass

Omdat we elke Model klasse de INotifyPropertyChanged interface zullen opleggen en we deze niet elke keer opnieuw wensen te coderen zullen al onze Model klasses ervan van een parent klasse die deze logica voor ons voorziet:



Bemerk volgende zaken:

* De klasse is abstract gedefinieerd. Zie [Abstract (C# reference)](https://msdn.microsoft.com/en-us/library/sf985hc5.aspx) voor meer uitleg hieromtrent.
* De hier getoonde implementatie implementeert enkel de interface INotifyPropertyChanged. In veel gevallen kunnen we in de base class meer logica terugvinden.

### ModelBaseClass en het EntityFramework

Het [EntityFramework](https://entityframework.codeplex.com/) is een populaire ORM (Object Relational Mapper: software die objects mapt op tabelinhouden in relationele systemen). In het EntityFramework is het gebruikelijk om op alle entities een ID veld te definiëren. In dit geval zou ik sterk aanraden om:

* Hiervoor een interface IEntity aan te maken die deze property definieert
* Door met de ModelBaseClass deze interface te implementeren zullen we dit veld niet in elke Model klasse moeten aanmaken (het wordt geërfd).

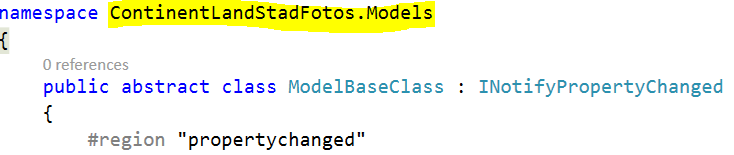
### ModelBaseClass: beschouwingen

Zoals de INotifyPropertyChanged implementatie en het EntityFramework voorbeeld hopelijk aangeven kunnen we volgende vuistregel formuleren:

* De klasse ModelBaseClass (ook deze naam is slechts een suggestie) bevat de code die gemeenschappelijk is aan (of niet storend is voor☺) alle Model klasses die we zullen ontwikkelen
* Indien na verloop van tijd blijkt dat deze klasse kan uitgebreid worden zullen we dit uiteraard niet nalaten
* Het definiëren vann één of meerdere interfaces voor de aanwezige elementen is geenszins verplicht maar heeft voglens mij volgende voordelen:
  + Een interface bundelt methodes tot een gewenst gedrag. Het definiëren van een interface verhoogt de kans op samenhang tussen de verschillende elementen van die interface!
    - Het is niet de bedoeling om elke methode een eigen interface te geven: elementen die samenhoren horen in dezelfde interface thuis.

## Models.Core folder

In het Models (en Data) project mogen natuurlijk ook folders gedefinieerd worden. In mijn voorbeeld implementatie worden ModelBaseClass en zijn interfaces gedefinieerd in de Core folder.

* Dit betekent by default dat de namespace van de ModelBaseClass en zijn interfaces in de namespace ContinentLandStadFotos.Models.Core gedefinieerd zijn. Om in mijn code niet teveel namespace te hebben verwijder ik voor alle Core elementen de laatste qualifier (Core) in de namespace definitie (C# file):

## Ontwikkeling van de Model klasses

### Extra Models folders

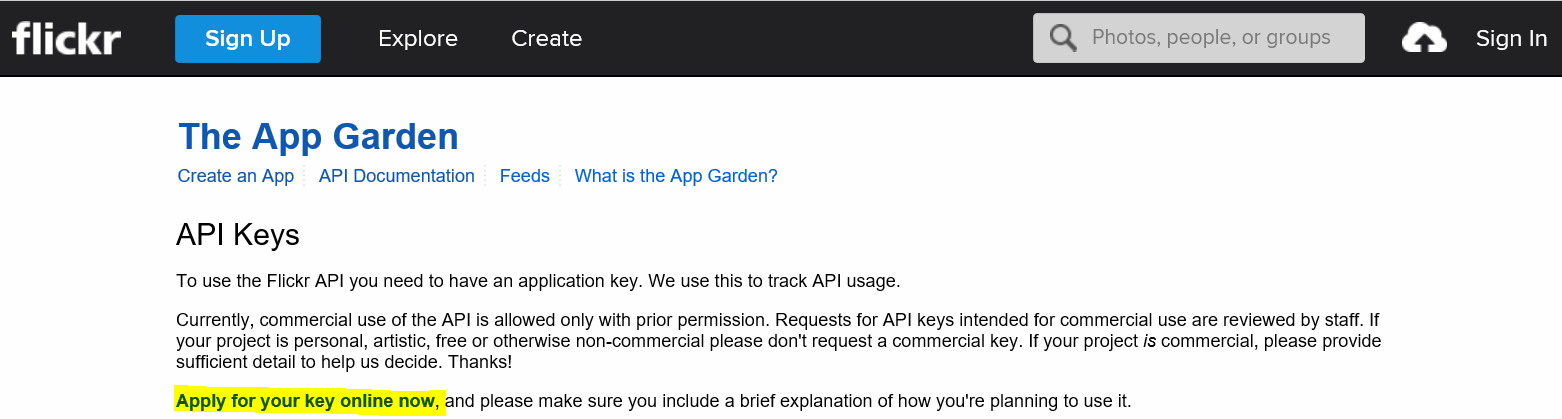
Omdat we voor deze toepassing informatie uit verschillende webservices zullen ophalen is het zinnig om de Models verder op te verdelen door middel van folders (en de laatste qualifier niet te verwijderen).

Voor deze toepassing voorzien we de Flickr, ArcGis en WorldWS folders.

## Flickr Models

Om Flickr opvragingen te kunnen uitvoeren moeten we een API- key aanvragen/bekomen:

* In [the app garden](https://www.flickr.com/services/api/misc.api_keys.html) heb je de mogelijkheid om een API key op te vragen:



* Vraag een key aan, u zal een Yahoo account nodig hebben;
* Gebruik [de photo search API explorer](https://www.flickr.com/services/api/explore/flickr.photos.search) om de API parameters in te stellen en de resulterende URL (onderaan de bladzijde) na te kijken. Genereer JSON output.
* Gebruik [json2csharp](http://json2csharp.com/) om de bekomen json tekst om te vormen tot C# klasses. U kan deze klasses in één file copiëren of voor elke klasse een aparte file maken. Ikzelf rename de klasse RootObject naar FlickrData en copieer alle tekst in 1 file
* Pas elk van de drie klasses aan zodat ze erven van ModelBaseClass (op deze manier erven ze de INotifyPropertyChanged ondersteuning)

Opmerking: code- inspiratie om de Flickr API vanuit een .NET programma te ondervragen vond ik op [Part 26: Retrieving a Photo from Flickr’s API](https://channel9.msdn.com/Series/Windows-Phone-8-Development-for-Absolute-Beginners/Part-26-Retrieving-a-Photo-from-Flickrs-API).

# Data project invulling

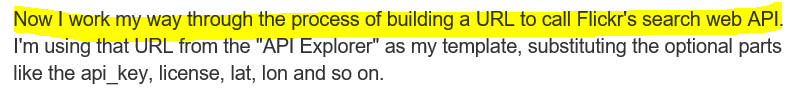
Het Data project bevat de repository klasses van de solution. Deze zorgen voor het aanmaken van Model- instanties op basis van opgehaalde gegevens (dummy data, data uit een database of van een webservice, ...). Ook hier zullen we waar mogelijk herbruikbare logica in een interface en abstracte base class definiëren.

Omdat we informatie in het JSON formaat zullen we ophalen zullen we een NuGet package installeren om de deserializatie van JSON tot .NET objecten voor ons te doen. Ikzelf verkies hiervoor het Newtonsoft package.

## FlickrRepository

Het ophalen van foto’s met behulp van de Flickr API gebeurt in de FlickrRepository. Indien we ook foto’s willen opladen of andere operaties uitvoeren zouden deze ook in deze Repository gecodeerd worden.

Indien u het artikel [Part 26: Retrieving a Photo from Flickr’s API](https://channel9.msdn.com/Series/Windows-Phone-8-Development-for-Absolute-Beginners/Part-26-Retrieving-a-Photo-from-Flickrs-API) er op naleest vindt u halverwege de tekst voorbeeld code om de API te ondervragen:



### FlickrRepository: voorbeeld implementatie

Een voorbeeld code om de FlickrRepository klasse te coderen kan zijn:



* Mijn code gebruikt bij de bepaling van de url- tekst String interpolation, een nieuwe C#6 feature die het formateren van teksten vereenvoudigt. Meer informatie hieromtrent vindt u op [Interpolated Strings (C# and VB reference)](https://msdn.microsoft.com/en-us/library/dn961160.aspx).

Bemerk dat ik nog geen BaseRepository class heb opgezet omdat de Flickr API voor mijn toepassing minder representatief is (het afhandelen van de requests gebeurt niet op dezelfde manier als in de voorbije lessen).

### FlickrRepository: testen van de implementatie

#### Photo Model update

Indien we dezelfde tekst [Part 26: Retrieving a Photo from Flickr’s API](https://channel9.msdn.com/Series/Windows-Phone-8-Development-for-Absolute-Beginners/Part-26-Retrieving-a-Photo-from-Flickrs-API) verder lezen vinden we op een bepaald punt de code om het bekomen resultaat om te vormen tot een foto- URL:

Indien u deze screenshot in de tekst opzoekt zal u merken dat het object data een Photo- instantie is.

Op basis van bovenstaande code is het duidelijk dat de berekening van de foto- URL enkel afhangt van de gegevens in een Photo- instantie. Het is dan ook zinnig om, anders dan in de code in het artikel, de klasse Photo te voorzien van de property PhotoUrl die deze berekeningen uitvoert.

* Pas de Model klasse Photo aan zodat ze een PhotoUrl property (type String) heeft die bovenstaande berekening uitvoert
* Een extension method is hier niet bruikbaar omdat we dit gegeven nodig zullen hebben om de Source property van een Image te binden. Extension methods kunnen jammer genoeg niet als bron gebruikt worden bij het opzetten van een Binding.

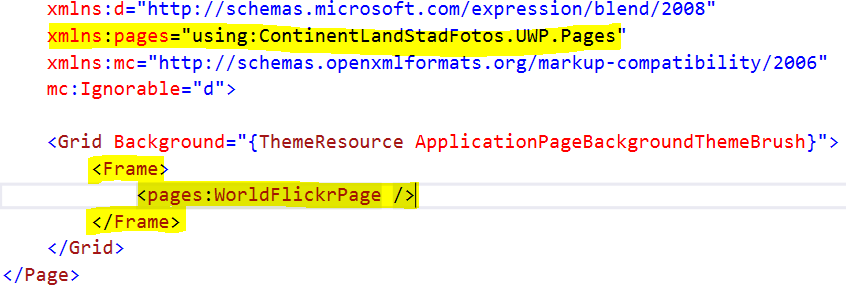
#### WorldFlickrPage, WorldFlickrViewModel en PhotoDataTemplate

Definieer in de Pages folder van het UWP project de Page WorldFlickrPage. Plaats in deze Page een GridView control. Deze zal de verschillende opgehaald foto’s als images tonen. Om dit via binding te realiseren zullen we volgende zaken nodig hebben

* Een WorldFlickrViewModel klasse met een Photos property. Deze property levert een lijst van Photo objecten aan
  + Omdat elke ViewModel klasse de INotifyPropertyChanged interface moet implementeren zullen we ook hier volgende stappen ondernemen:
    - Maak in de ViewModels folder een folder Core
    - Definieer hierin de klasse ViewModelBaseClass die de INPC interface implementeert. Verwijder uit de namespace de laatste qualifier.
    - Elke ViewModel klasse zal erven van ViewModelBaseClass, ook de klasse WorldFlickrViewModel.
  + Omdat de FlickrRepository de gegevens asynchroon aanlevert zullen we, net zoals in de vorige oefeningen eerst null teruggeven, dan de gegevens ophalen en dan via INPC de Binding verwittigen.
    - Hint: gebruik als tag de string “beautiful” indien u mooie foto’s wenst
* Een DataTemplate PhotoDataTemplate om Photo objecten te visualiseren:
  + Om dit via Binding mogelijk te maken werd de property PhotoUrl aan de Photo klasse toegevoegd
  + De DataTemplate bevat onder andere een Image control waarvan de Source property aan de PhotoUrl property wordt gebind
  + Omdat het nu toch een architectuur les betreft: gelieve de DataTemplate in een aparte resource file te definiëren.

#### Putting it all together?

* Maak een static resource in App.xaml van de viewmodel klasse met als key flickrvm
* Vergeet niet in de App.xaml een verwijzing naar de DataTemplates.xaml file op te nemen, omdat daarin de te gebruiken DataTemplate wordt beschreven
* Compileer op dit punt uw toepassing tot er geen compilatie fouten meer optreden. Op deze manier is straks bij het databinden intellisense mogelijk.
* Plaats de DataContext van WorldFlickrPage op de static resource flickrvm
* In de GridView van de WorldFlickrPage worden volgende properties ingesteld:
  + ItemsSource wordt gebind, bij het instellen van de Binding toont de intellisense u de mogelijkheden voor het Path attribuut
  + ItemTemplate wordt gebind aan de StaticResource die uw datatemplate is
* Opdat de MainPage deze WorldFlickrPage zou tonen voorzien we in MainPage.xaml de volgende code:



Wannneer we het later in deze tekst over Navigation Services zullen hebben zullen we hierop terugkeren.

Start uw toepassing en u zou foto’s moeten te zien krijgen.

Hint: indien u in uw datatemplate de image Width en Height beperkt tot bijvoorbeeld 100 (pixels) dan zal de GridView veel photo’s op één bladzijde tonen.

Opmerking: indien u de GridView tag vervangt door FlipView krijgt u een andere visualisatie van dezelfde gegevens, waarbij het niet wenselijk is om de foto’s in een klein formaat te tonen.

# World info

In het uiteindelijke programma is het de bedoeling om foto’s op basis van gegevens in de World webservice te tonen (de gegevens van de WorldWS zijn overgenomen uit de World database in een default MySQL installatie)

## World Models

Om de models voor de WorldWS te kunnen aanmaken moet u de API ondervragen. Het project WorldWS vindt u op Leho terug.

* Continenten: http://localhost:57757/api/continent
* Landen: <http://localhost:57757/api/country?continent=europe> (om de landen van europa op te vragen)
* Steden: <http://localhost:57757/api/city?countrycode=BEL> (om de steden van België op te vragen)

Op basis van de responses moet u de model klasses kunnen aanmaken. Vergeet niet:

* Uw model klasses erven van de ModelBaseClass
* Indien een value type property null kan zijn die u de nullable variant van het type te kiezen. De verkorte notatie gebruikt een ?, dus int? is gelijk aan Nullable<int>
  + Indien u geen nullable type gebruikt en het resultaat van de oproep levert geen waarde voor dit veld dan zal de deserialisatie falen.
* Maak een Continent, Country en City klasse in een World folder in het Models project.
  + Hint: ik voorzie elk van deze klasses ook van een gepaste ToString methode

## World Repositories

### ContinentRepository

Het ophalen van de informatie gebeurt op dezelfde manier als de voorbije lessen. U kan de code copiëren en u hoeft enkel volgende zaken aan te passen:

* De url van de webservice
* Het type object waarvan we een lijst opvragen moet op verschillende plaatsen worden aangepast
* De naam van de controller moet worden aangepast in de GetAsync methode

Anders dan in de vorige versies:

* De methode waarmee de gegevens worden opgehaald heet **Get**
* De Get- methode is NIET STATIC gedefinieerd

Maak deze repository klasse en test ze in WorldFlickrPage op volgende manier:

* Voorzie een property Continents op het ViewModel (op de asynchrone manier)
* Voorzie op de Page een extra rij bovenaan, plaats daarin drie comboboxen, de eerste combobox bevat de continenten en wordt gebind aan de property Continents
* Voorzie ook een SelectedContinent property op het ViewModel en deze wordt (TwoWay) gebind aan de SelectedItem property van de combobox.
* Gebruik het geselecteerde continent als extra tag bij het opvragen van foto’s. Dit betekent dat het wijzigen van het geselecteerde continent opnieuw de foto’s zal opvragen, nu met een andere tag parameter.
  + Opdat dit zou werken moet u ook de tag\_mode parameter in de flickr url opnemen

### CountryRepository

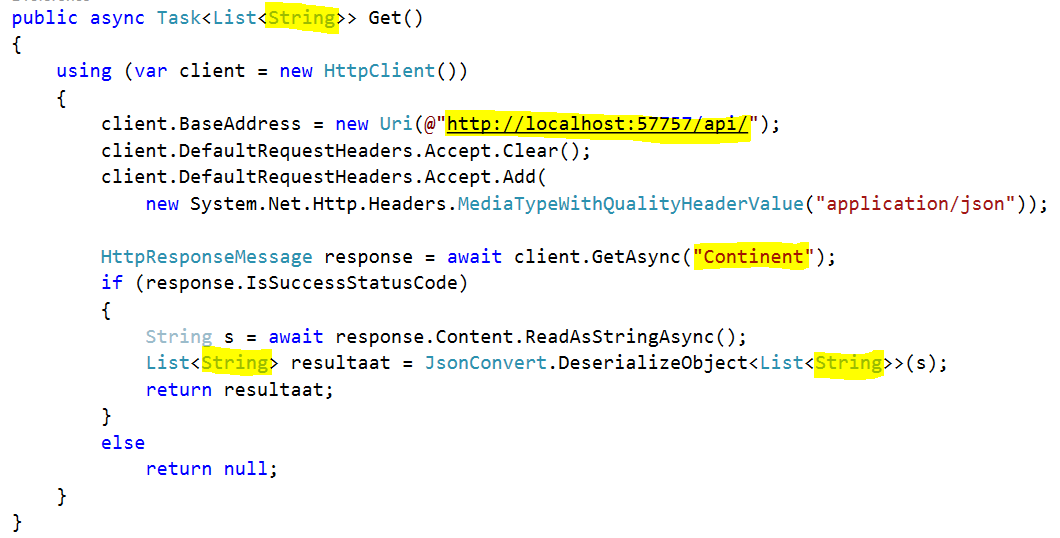
Idem, voorzie ook een combobox

### CityRepository

Idem, voorzie ook een combobox

## RepositoryBaseClass<T>

De klasses ContinentRepository, CountryRepository en CityRepository verschillen zeer weinig van elkaar. Voor de ContinentRepository hadden we onderstaande code (de klasse Continent is gewoon de String klasse):



De potentieel variabele stukken zijn in het geel aangeduid:

* Het resultaat type in de List is afhankelijk van de Repository (in dit geval String)
* De Api url kan verschillend zijn per repository
* De controller naam is verschillend per Repository (in dit geval Continent)

Om een baseclass te ontwikkelen waarvan ContinentRepository, CountryRepository, CityRepository (en eventuele toekomstige repositories) kunnen erven gaan we als volgt te werk:

* De klasse zal een String property ApiUrl bevatten die door de ervende klasses wordt ingevuld
* De klasse zal een String property Controller bevatten die door de ervende klasses wordt ingevuld
* De klasse zal generisch gedefinieerd worden: het type van de ervende klasse wordt als parameter tussen <> genoteerd.

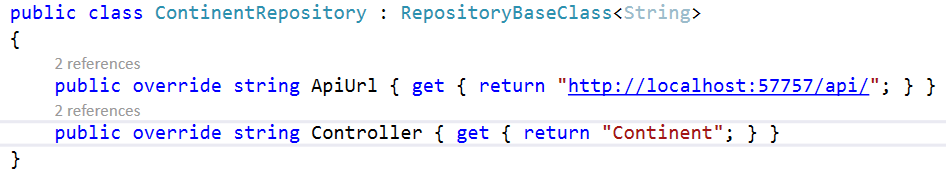
Dit levert volgende code op:



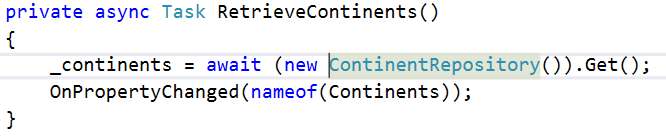
* Hopelijk valt het op dat hier dezelfde stukken in het geel werden aangeduid als in de vorige screenshot
* De properties ApiUrl en Controller werden abstract gedefinieerd. Dit verplicht een ervende klasse (indien ze niet abstract is) om deze methode van een implementatie te voorzien
* Deze properties worden in de Get methode gebruikt (zie rode boxes)
* De klasse zelf werd ook abstract gedefinieerd. Dit betekent dat er geen instanties van deze klasse gemaakt kunnen worden (men moet erven van deze klasse)

Eventueel beslist u om de BaseClass features in een Interface te definiëren en deze op te leggen aan deze BaseClass

### ContinentRepository Bis

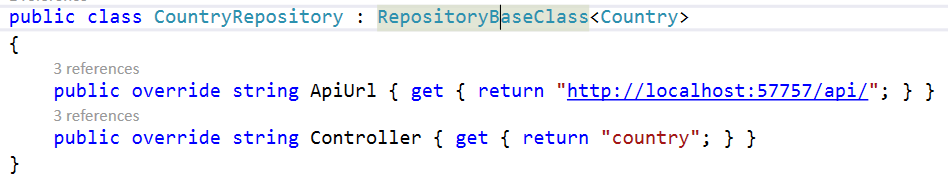


We gebruiken deze repository op net dezelfde manier als de vorige implementatie:



* Omdat Get niet static gedefinieerd werd moeten we een RepositoryObject gebruiken
* RetrieveContinents wordt gecodeerd in het ViewModel

### CountryRepository Bis



Bovenstaande CountryRepository kan via de geërfde Get methode alle landen opvragen. De REST API ondersteunt het opvragen van alle landen van een continent, om niet alle landen van de server naar de client te moeten sturen (minder netwerk verkeer, minder processing op de client die een kleinere JSON moet verwerken en zelf niet moet filteren).

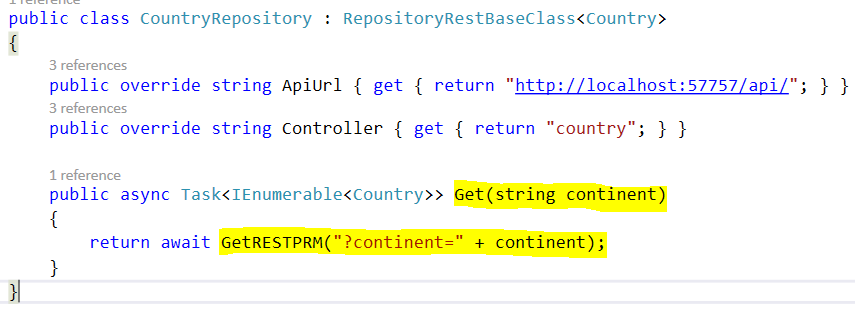
Om deze filtering mogelijk te maken zullen we de repository op volgende wijze aanpassen:

* Voorzie een Get- methode met als parameter het Continent waarop we willen filteren. Hiermee kunnen we REST API parameter bepalen (Parameters worden in een url genoteerd na het ? teken, en van elkaar gescheiden door het & teken).
* De code om de REST api te ondervragen met of zonder parameter is zeer gelijkaardig. We passen de RepositoryBaseClass aan op volgende manier:
  + Hernoem de file en de klasse naar RepositoryRESTBaseClass (omdat we REST specifieke code zullen toevoegen)
  + Voorzie de methode GetRESTPRM die gebruikt zal worden om de API te ondervragen met een parameter (deze code is bijna volledig gelijk aan de oorspronkelijke Get code)
  + Wijzig de Get code zodat deze getRESTPRM oproept met een lege string als parameter



* De methode Get zal dezelfde commando’s uitvoeren als de vorige implementatie
* Indien er een correcte Url parameter wordt meegegeven (u moet hiervoor de API documentatie raadplegen) worden gefilterde gegevens opgevraagd.

De uiteindelijke CountryRepository implementatie wordt dan:

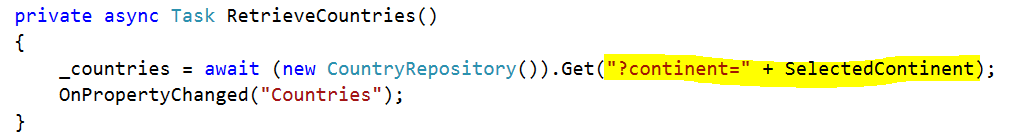


Bemerk hierbij wel dat de methode Get nog altijd kan gebruikt worden om alle landen op te vragen. In het ViewModel schrijven we volgende code om de landen van een continent op te vragen:



#### belangrijke potentiële fout

Bij het ophalen van de landen van een continent moet juiste parameter worden meegegeven. In het ViewModel kunnen we de REST API parameter reeds schrijven (zie onderstaande code waarbij de parameter in het geel werd aangeduid):



Dit is een slechte oplossing omdat op deze manier de invloed van de technische bron van de gegevens tot in het ViewModel terug te vinden is: de technische afhandeling van de data manipulaties moet zich beperken tot de repository. Indien we bijvoorbeeld niet langer een REST api als databron gebruiken maar rechtstreeks een database ondervragen is de hier in het geel aangeduide parameter duidelijk niet correct.

### CityRepository

Volledig gelijkaardig aan CountryRepository

# UWP project invulling

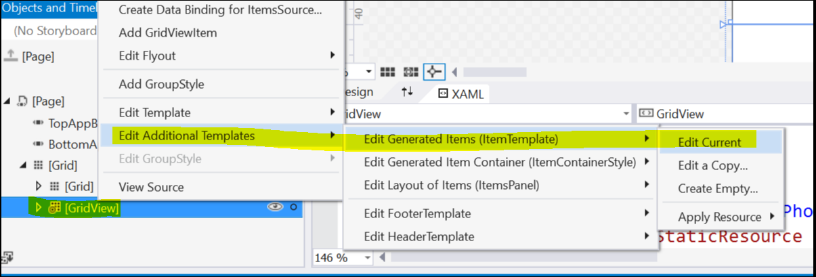
## ViewModel en View uitwerking

* U voorzag eerder reeds drie combo boxen bovenaan de Page. Deze zullen we nu koppelen aan ViewModel properties
* Vul het ViewModel aan met volgende properties:
  + Continents, Countries, Cities. Deze worden als ItemsSource van hun respectievelijke comboboxen ingesteld
  + Voorzie de properties SelectedContinent, SelectedCountry en SelectedCity die aan de SelectedItem property van de respectievelijke combobox worden gebind (TwoWay binding)
  + De setter van de Selected properties zal op basis van de geselecteerde waarde foto’s ophalen met een aangepaste tag (de tag is de naam van het geselecteerde item)
  + Indien nodig zal de setter ook nieuwe gegevens ophalen:
    - De selectie van een nieuw continent zal een nieuwe landenlijst ophalen
    - De selectie van een nieuw land zal een nieuwe stedenlijst ophalen
* Plaats de opgehaalde foto’s in een GridView en maak de foto’s behoorlijk klein zodat u er veel op 1 pagina te zien krijgt.

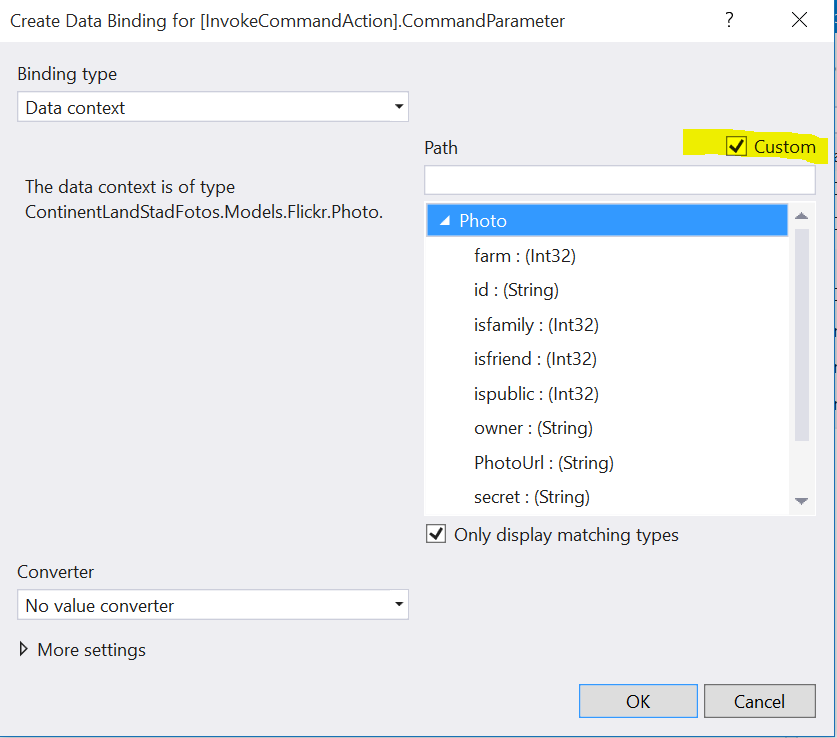
# Navigatie

## Grote foto’s tonen

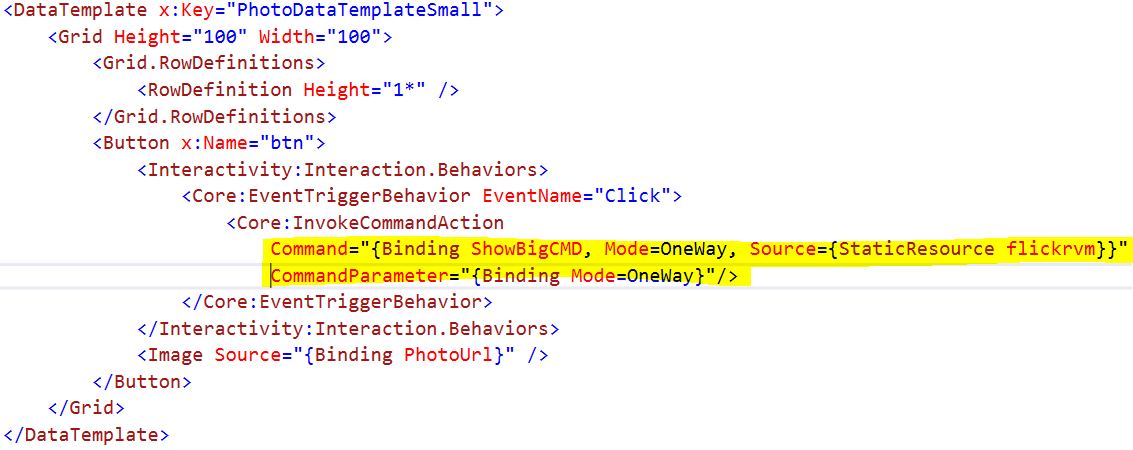
Wanneer in de GridView op een kleine foto wordt geklikt wensen we de aangeduide foto in het groot te tonen op een andere pagina.

* Voorzie op het ViewModel een RelayCommand<Object> ShowBigCMD. Deze zal de methode ShowBig(object o) oproepen.
  + Hoe u een RelayCommand definieert: zie vorige les
  + Hoe u een RelayCommand koppelt aan een event in een DataTemplate: zie vorige les (Blend) (zoek het RelayCommand in de ViewModel static resource!)
  + In dit geval wensen we een parameter aan het Commando mee te geven (de Photo die we groot wensen te tonen). Dit realiseren we in Blend op de volgende manier:
* Right click op de GridView en kies Edit Additional Templates.Edit Generated Items (ItemTemplate).Edit Current

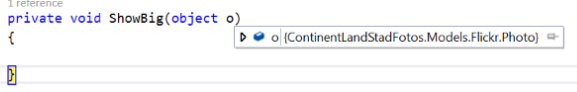
Dit laat u toe om de ItemTemplate van de GridView te editeren. Het grote voordeel is dat Blend kan afleiden (door de ItemsSource Binding) welke DataContext in de DataTemplate wordt toegepast

Het binden van de CommandParameter gebeurt nu in Blend. Blend weet dat de DataContext een Photo object is. Indien we Custom aanvinken en geen Path intikken zal de volledige datacontext als Source gebruikt worden. Dit is de bedoeling, op deze manier wordt de volledige Photo doorgegeven aan het commando.

Bovenstaande manipulaties leiden tot volgende DataTemplate:



Indien we de code uitvoeren komen we in de routine ShowBig met een Photo object.



De vraag die zich nu onmiddellijk stelt: hoe kunnen we vanuit het ViewModel van een View navigeren naar een andere View?

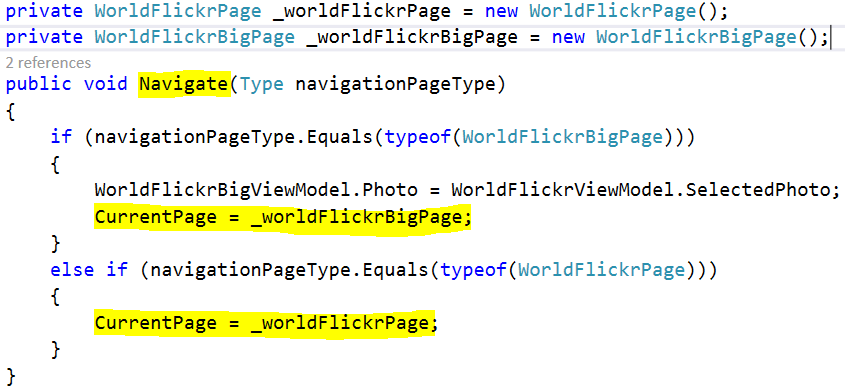
## Navigatie in MVVM: ApplicationViewModel

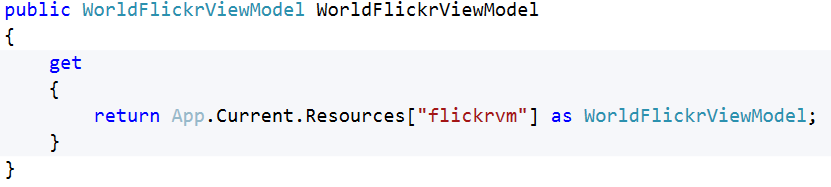
Dit is niet de enige benadering, maar ze past wel perfect in het MVVM verhaal.

### ApplicationViewModel

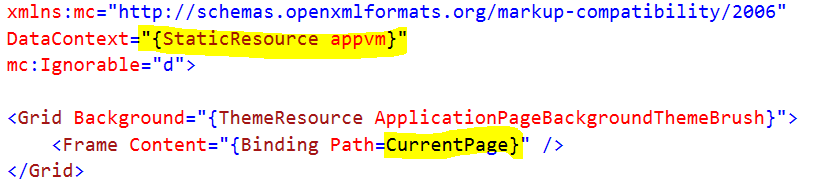
We ontwikkelen een ViewModel voor de toepassing die onder andere volgende features zal vertonen:

* CurrentPage: de momenteel getoonde Page
  + Deze zal gebind worden aan de content van het Frame op MainPage.
  + In de constructor wordt deze ingesteld op een pagina naar keuze
* public void Navigate(Type navigationPageType): deze method wijzigt de CurrentPage. De INPC interface verwittigt de Binding die op zijn beurt de nieuwe pagina toont: Navigatie!
  + Werken met het type als paramter is omslachtig maar stemt overeen met de Navigate methode op de Frame klasse.
* Voor elk Pagina type waarnaar genavigeerd kan worden houden we exact één object bij in een field. We navigeren naar deze pagina’s op de volgende wijze:



* De CurrentPage property heeft als return type Page (waar alle pages van erven) en INPC wordt getriggerd bij een wijziging van de property waarde
* WorldFlickrViewModel en WorldFlickrBigViewModel zijn properties die verwijzen naar de in App.xaml gedefinieerde static resources van het overeenkomstige type. Het zijn dezelfde objecten die ook als DataContext van hun respectievelijke Pages worden ingesteld!
* 

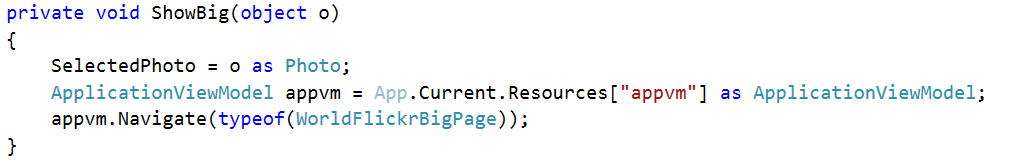
### MainPage.xaml aanpassingen



* appvm is een StaticResource (gedefinieerd in App.xaml) voor het ApplicationViewModel
* door de content van het Frame te binden aan CurrentPage wordt deze getoond.

### PageViewModel aanpassingen

Indien een ViewModel moet navigeren zal het uit te voeren commando het application ViewModel uit de resources ophalen en de Navigate instructie uitvoeren:



# Services

## ViewModelLocator

In de huidige benadering zal elke pagina zijn ViewModel bij naam ophalen uit de App.xaml resources. Dit heeft als nadeel dat we om een ander ViewModel (test? Design? Produktie?) te gebruiken onze code moeten aanpassen: ofwel een ander type met dezelfde naam in App.xaml, ofwel een andere naam gebruiken. We kunnen dit probleem omzeilen door de introductie van de klasse ViewModelLocator. Deze klasse heeft volgende eigenschappen:

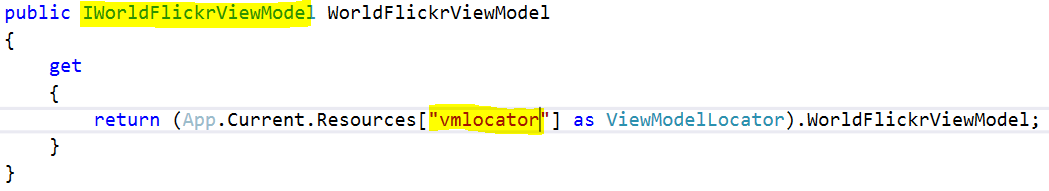
* Voor elke Page voorzien we een ViewModel property. Deze mag uiteraard niet gebonden zijn aan een specifieke ViewModelKlasse, anders is deze property terug niet uitwisselbaar
  + Maak van de huidige ViewModel klasse van een Page een interface. De gemakkelijk Visual Studio manier om dit te bekomen is ‘Edit.Refactor.Extract interface’ wanneer de cursor zich in de ViewModel file bevindt. Maak voor elk van de ViewModels een interface met dezelfde naam als de klasse, voorafgegaan door I (hoofdletter i);
  + Het resultaat type van de property is de interface
  + Bij het opvragen van deze property waarde geven we altijd hetzelfde object terug.

Maak deze klasse in het UWP project in de folder Services (nog aan te maken)

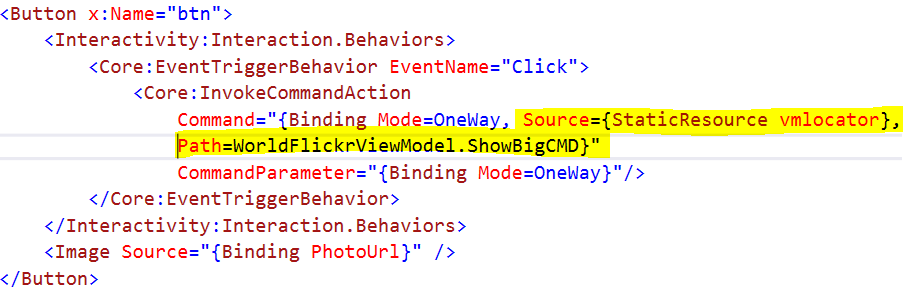
* Voorzie in App.xaml een staticresource van dit type met als key vmlocator
* Verwijder de andere viewmodel static resources.
* We zullen de datacontext van elke pagina instellen vertrekkend vanuit vmlocator:



* In het ApplicationViewModel zullen we de ViewModel properties aanpassen zodat ze hun informatie niet langer uit de resources halen, maar opvragen aan de ViewModelLocator. Bemerk dat het resultaat type nu ook een Interface moet zijn (omdat de ViewModelLocator als resultaat type een interface heeft):

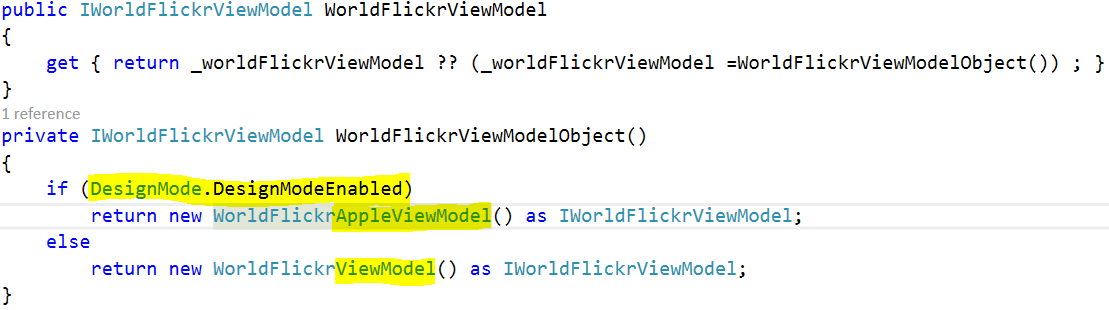


* Indien in een DataTemplate het ViewModel StaticResource wordt gebruikt (wellicht voor de CommandBinding) zullen we ook hier (net zoals in de Page DataContext) gebruik maken van vmlocator:

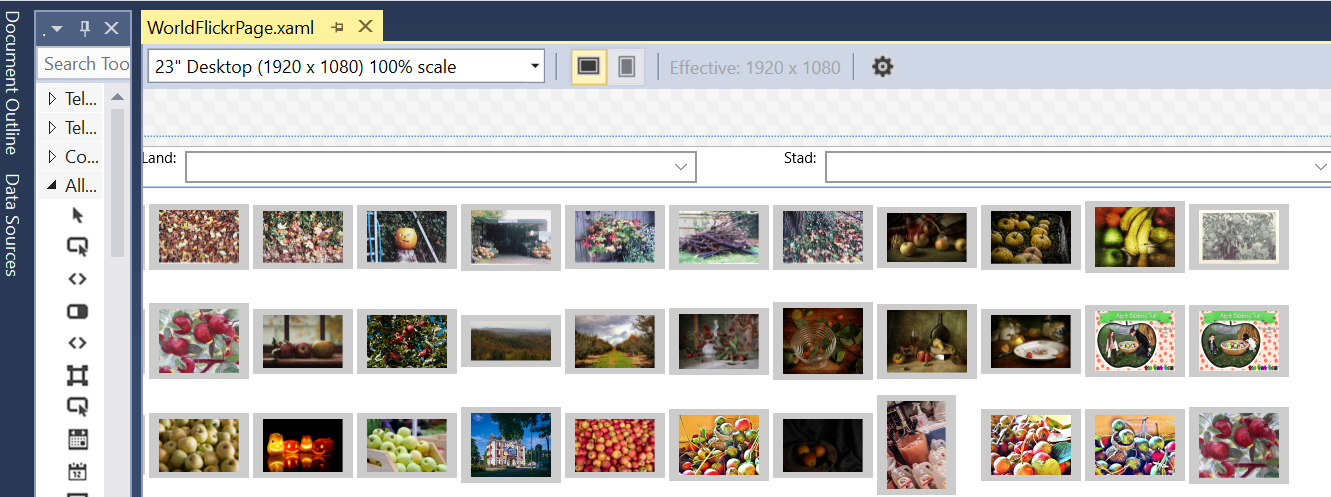


Het voordeel van deze benadering is dat de ViewModelLocator klasse, afhankelijk van voorwaardes, een ander ViewModel voor een bepaalde pagina kan teruggeven:

* Een admin gebruiker krijgt een ViewModel te zien met alle mogelijkheden, een gewone user met minder mogelijkheden (bijvoorbeeld de beschikbaarheid van commando’s)
* In test kan een ander ViewModel gebruikt worden dan in productie
* Bij het uitvoeren van UnitTest kunnen andere ViewModels nodig zijn



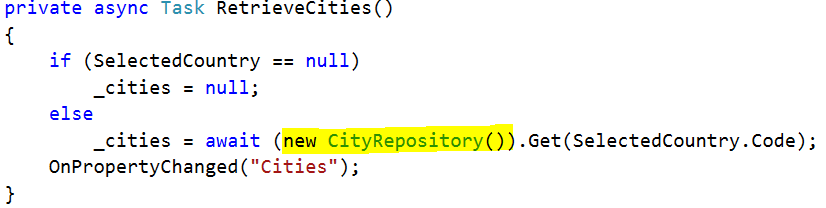
Met bovenstaande code worden at design time vooral appels getoond omdat apples de gebruikte tag is in het AppleViewModel:



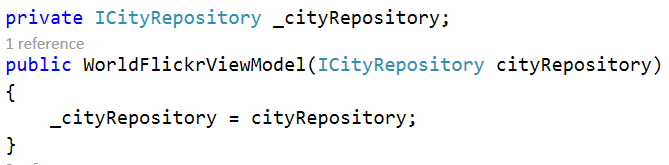
Tussen (): het DesignViewModel werd zeer snel gecodeerd door middel van overerving.

## Dependency injection (DI)

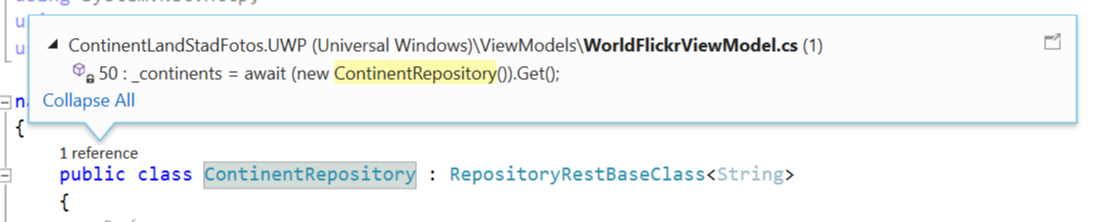
Voor een Wiki uitleg: [Dependency injection](https://en.wikipedia.org/wiki/Dependency_injection)

In de viewmodels klasse is er veelal een hardgecodeerde afhankelijk (dependency) met de verschillende repository klasses. Zo is in de code aan de rechterkant de gebruikte CityRepository hard coded. Indien we een andere repository klasse wensen te gebruiken (door de ganse toepassing) moeten we dit doorheen de ganse toepassing aanpassen (op elke plaats waar deze repository wordt gebruikt moeten we een keuze implementeren. Indien de keuze mogelijkheden wijzigen moet dit overal aangepast worden).

We kiezen voor een andere aanpak:

* We maken een interface die alle features van deze repository klasse definieert (gebruik terug de ‘extract interface’ edit.refactor feature terwijl de cursor zich in de klasse CityRepository bevindt
  + Indien uw repository klasse erft van een base class dan moet u ook voor de base class een interface maken en moet de repository interface de baseclass interface erven
* Definieer een field in de klasse van dit interface type
* Wijzig de constructor van de ViewModel klasse zodat deze een parameter van het interface type ontvangt. Ken deze parameter waarde toe aan het zonet gedefinieerde field
* Gebruik nu in de ganse ViewModel klasse dit field in plaats van de new geconstrueerde objecten.

Herhaal dit voor elk van de repository klasses in elk van de ViewModels. Indien u wenst na te kijken waar deze klasses overal gebruikt worden gebruikt u de CodeLens feature:



Met de nu aangepaste wijzigingen is het niet meer mogelijk om een ViewModel resource van deze klasses te maken omdat er geen parameterloze constructor meer is. Het laatste op te lossen probleem is dus het centraal aanmaken van objecten van deze klasses.

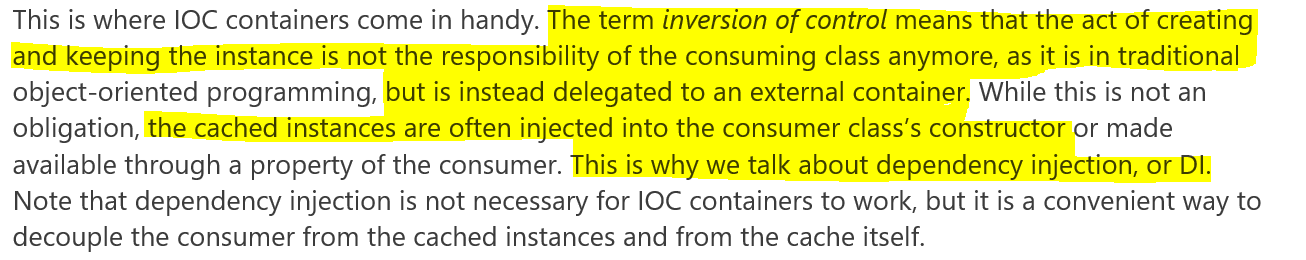
## Inversion Of Control (IoC)

Voor een Wiki uitleg: [Inversion Of Control](https://en.wikipedia.org/wiki/Inversion_of_control)

Geinterpreteerd en vertaald uit bovenstaande tekst: Inversion of Control (IoC) is een design waarbij custom code de ‘flow of control’ ontvangt van een algemene, herbruikbare bibliotheek.

De interpretatie van ‘flow of control’ kan wellicht aanleiding geven tot urenlange discussies zonder definitief uitsluitsel.

Uit [MVVM - IOC Containers and MVVM](https://msdn.microsoft.com/en-us/magazine/jj991965.aspx) (een tekst van de ontwikkelaar van MVVM Light) licht ik onderstaande tekst:



Een IoC container levert volgende diensten/voordelen:

* We zullen op één centrale plaats registereren welke klasses worden gebruikt om welke interfaces te implementeren
  + Gelijkaardig aan het voorbeeld in de ViewModelLocator klasse: at design time of at runtime, voor een admin user of een gewone user, voor een unittest run: telkens kunnen verschillende klasses gebruikt worden om éénzelfde interface (gebruikt in de verschillende klasses) te realiseren
* De IoC containter laat toe om telkens hetzelfde object, aangemaakt voor een interface te gebruiken. Op die manier moeten we object niet meer als static resource definiëren
* De IoC container laat toe om automatisch dependency injection te bekomen: indien een constructor nood heeft aan een parameter van een type (interface) dat reeds in de IoC container gekend is, dan zal dit object automatisch gebruikt worden.

Wow, kunnen we wat code zien?



SimpleIoc is een klasse gedefinieerd in het MVVM light package. De eerste registratie registreert welke klasse gebruikt zal worden voor welke interface. Er wordt ook onmiddellijk een object aangemaakt.

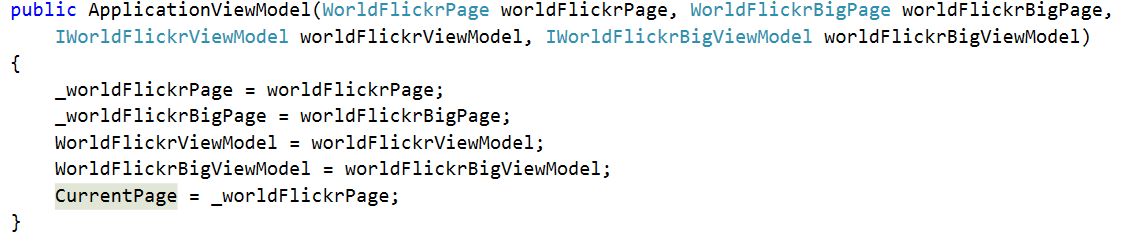
De tweede geel aangeduide lijn registreert een klasse zonder interface, en ook hier wordt onmiddellijk een object van deze klasse gemaakt.

De derde oranje lijn registreert een klasse zonder onmiddellijk een instantie te maken. Dit kan soms nodig zijn indien onderlinge afhankelijkheden nog niet afgewerkt zijn. In bovenstaand geval hebben de Pages en het ApplicationViewModel nood aan een IocContainer object, dat uiteraard nog niet gemaakt is, gezien de static constructor nog uitgevoerd wordt.

Opmerking: een static constructor wordt één keer uitgevoerd, bij eerste referentie naar gelijk welk element van de klasse.

De ViewModelLocator property toont aan hoe nu objecten van deze geregistreerde klasses kunnen opgevraagd worden. Door ze als property op deze klasse te definiëren kunnen ze onderdeel uitmaken van een binding. Ik definieerde ze static om een snelle toegang via C# mogelijk te maken.



In het ApplicationViewModel (en de andere klasses) moet de afhankelijk van geregistreerde klasses vervangen worden door Dependency Injection: via de constructor van die klasses worden de nodige objecten meegegeven en bewaard in een field of property:

## Uiteindelijke opkuis

Verwijder alle static resources op een static resource voor de klasse IoCContainer na.

Hint: gebruik een find instructie om alle voorkomens van het static resource dat u zal verwijderen terug te vinden en wijzig de code zodat er gewerkt kan worden zonder dit static resource vooraleer u het verwijdert!

# Conclusie

Een volledige uitwerking van al het voorgaande is:

* Veel werk (tijdrovend)
  + Het is veel minder werk wanneer we onmiddellijk alles goed uitwerken
  + In voorgaande tekst werd een ontwikkel traject gevolgd (zullen we in de oefeningen niet doen)
  + De interfaces zijn niet nodig in eenvoudige toepassingen: er is maar één implementatie van ViewModels en Repositories
* Dependency Injection vereist een IoC container implementatie (omwille van de parameters in de constructors)
  + Indien een IoC container gebruikt wordt moet dit vanaf de start goed geïmplementeerd worden.
* De koppeling van een DataTemplate commando aan een specifiek static resource element is een grote beperking: het maakt de DataTemplate onbruikbaar op een Page met een ander ViewModel
  + In theorie zou het wellicht mogelijk zijn een CommandLocator op te zetten ...
* Het ApplicationViewModel biedt nog veel extra mogelijkheden: we kunnen de navigatie historiek bijhouden indien dit zinnig zou zijn.
* Dependency Injection en bijhorende IoC container zijn moeilijk te debuggen ☹: de exceptions bevatten zeer weinig informatie

# ArcGis webservice (appendix)

Wanneer we met de flickr api foto’s voor locaties opvragen kunnen we meer accurate gegevens bekomen door de geolocatie van de op te zoeken plaats mee te geven. Een gratis webservice die op basis van een stadsnaam deze informatie teruggeeft kan u vinden op [ArcGis Rest API](https://developers.arcgis.com/rest/geocode/api-reference/geocoding-find.htm)