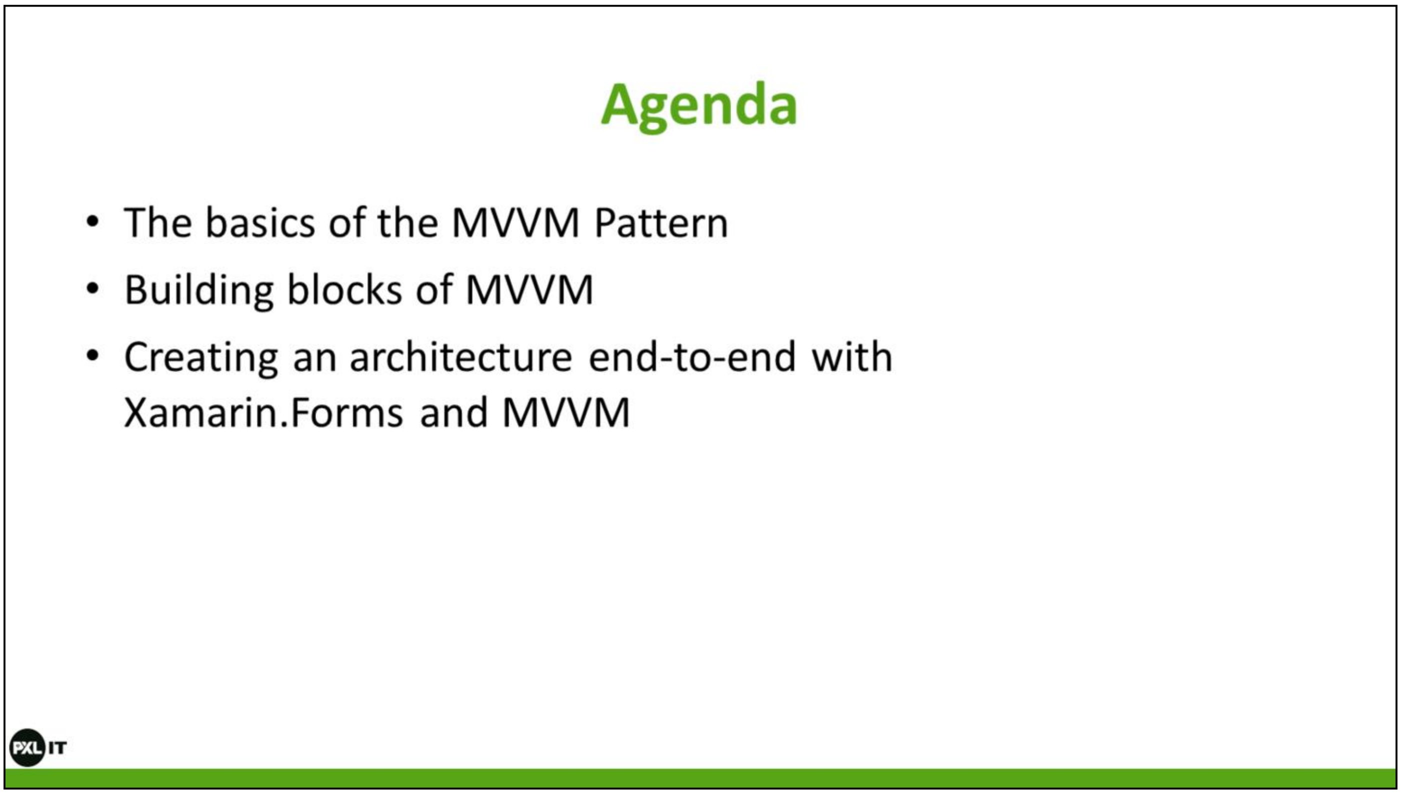
PXL IT

Xamarin.Forms and MVVM

DE HOGESCHOOL
MET HET NETWERK

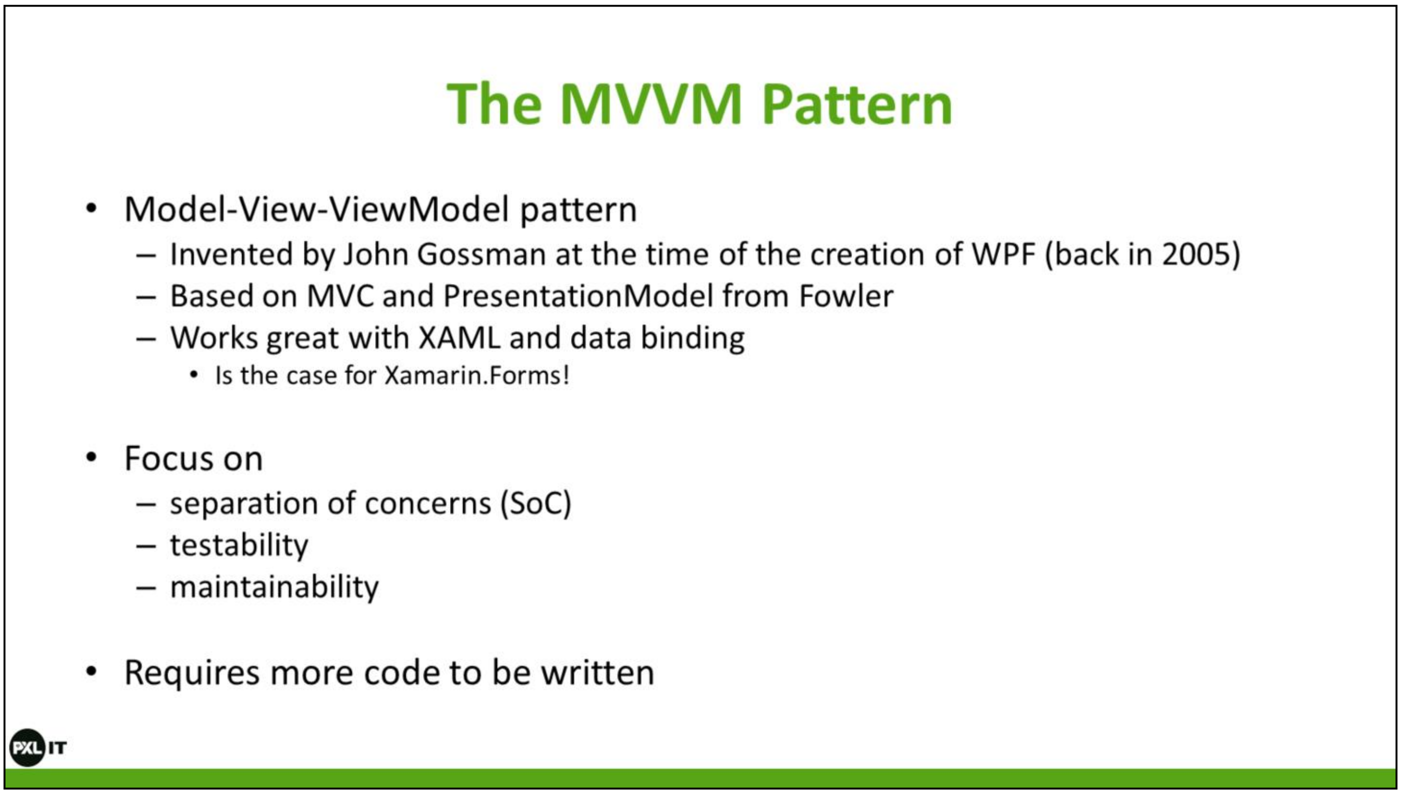
Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook



Agenda

- The basics of the MVVM Pattern
- Building blocks of MVVM
- Creating an architecture end-to-end with Xamarin.Forms and MVVM

THE MVVM PATTERN



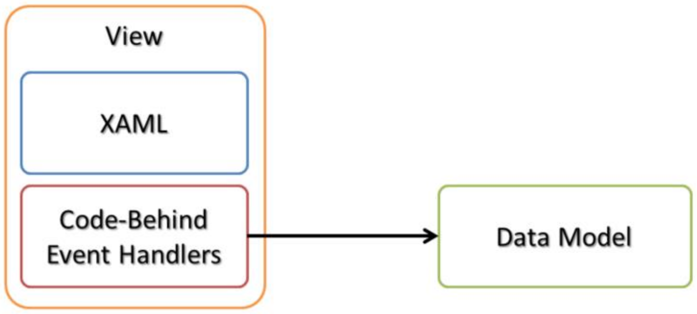
The MVVM Pattern

- Model-View-ViewModel pattern
  - Invented by John Gossman at the time of the creation of WPF (back in 2005)
  - Based on MVC and PresentationModel from Fowler
  - Works great with XAML and data binding
    - Is the case for Xamarin.Forms!
- Focus on
  - separation of concerns (SoC)
  - testability
  - maintainability
- Requires more code to be written

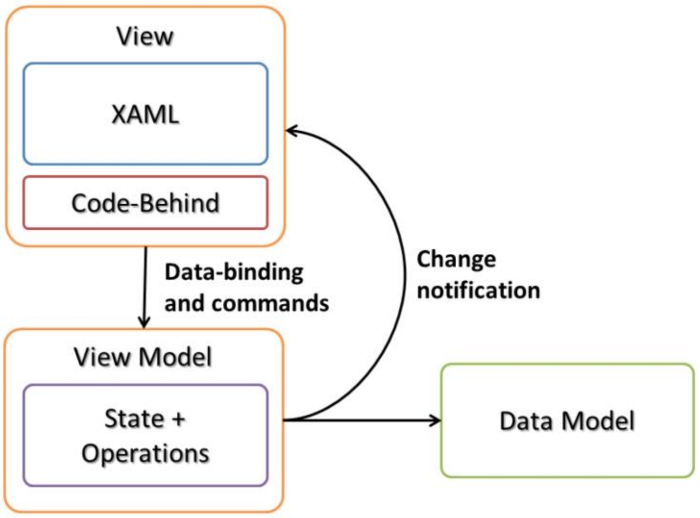
The Model-View-ViewModel (MVVM) architectural pattern was invented with XAML in mind. The pattern enforces a separation of the XAML user interface (the View) from the underlying data (the Model) through a class that serves as an intermediary between the View and the Model (the ViewModel). The View and the ViewModel are often connected through data bindings defined in the XAML file. The BindingContext for the View is usually an instance of the ViewModel.

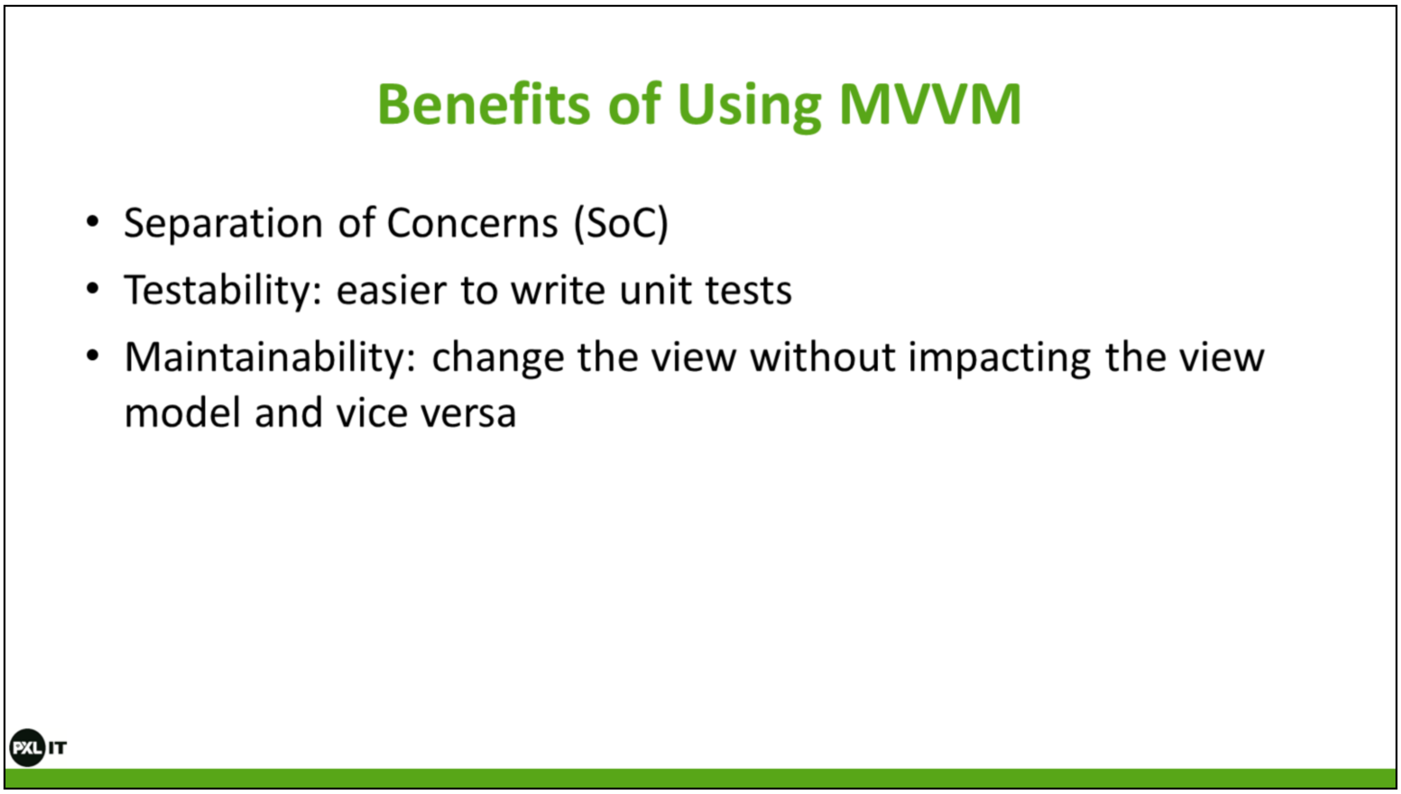
What we all did when we were young…

Write code in code-behind… Lots of it.



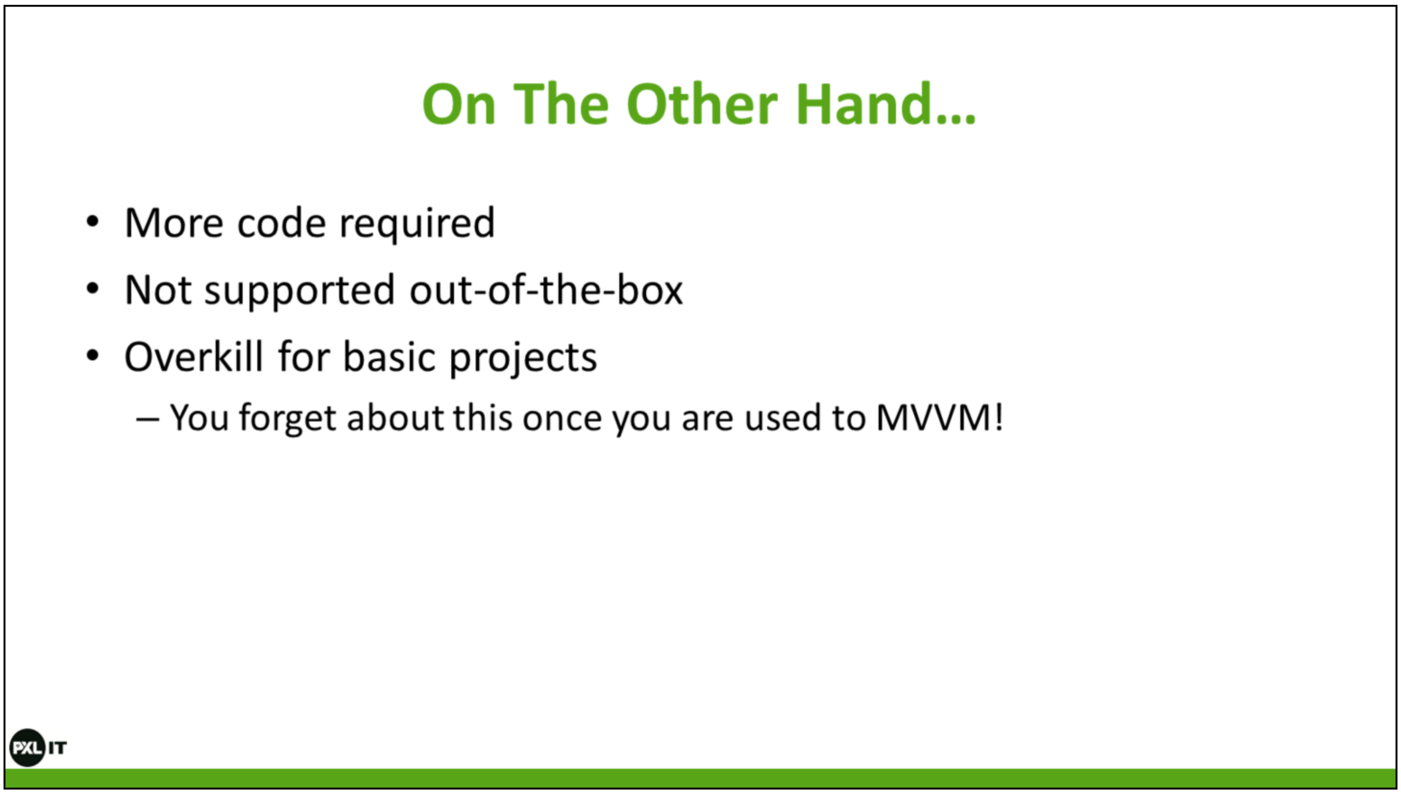
Writing testable code however, is becoming the norm. Finally. Courtesy of MVVM.





Benefits of Using MVVM

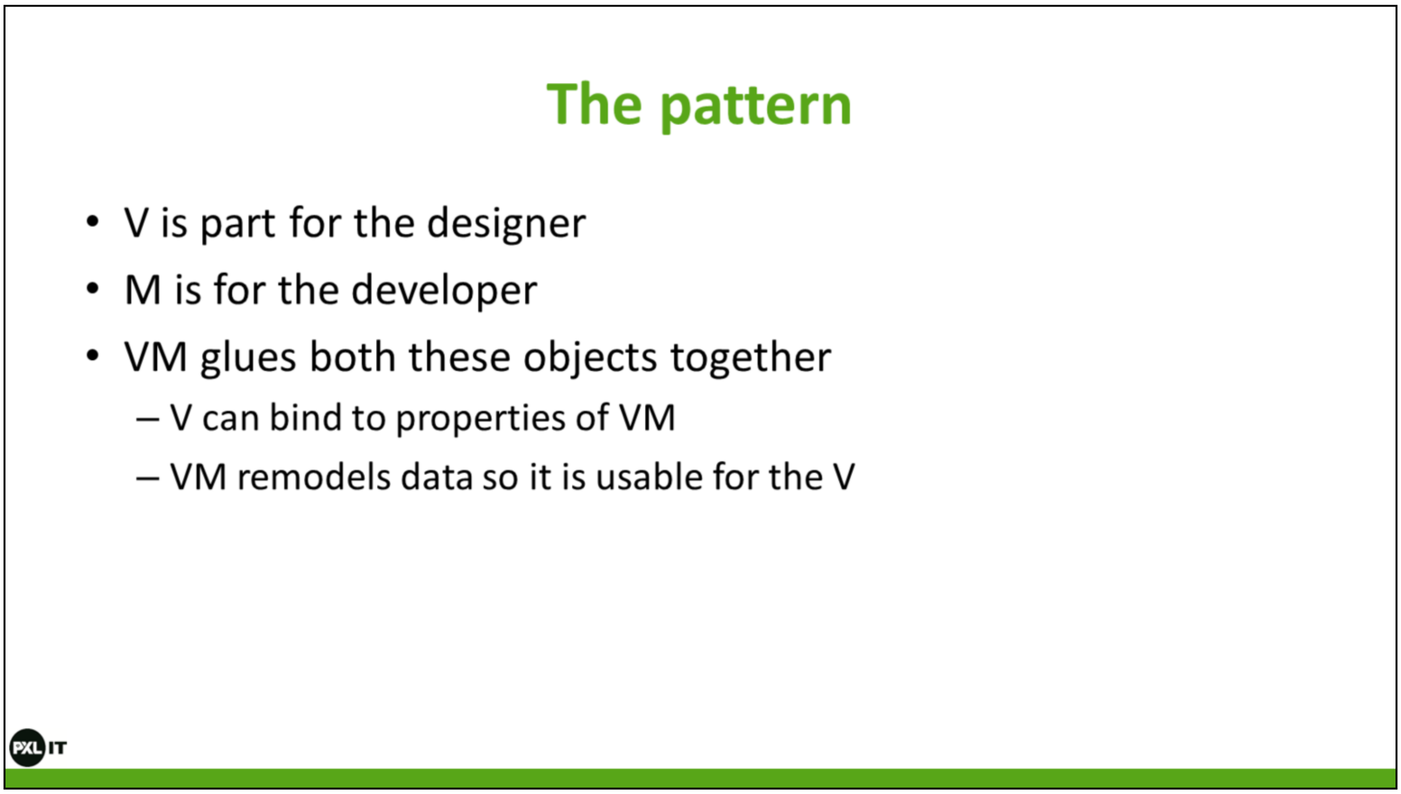
- Separation of Concerns (SoC)
- Testability: easier to write unit tests
- Maintainability: change the view without impacting the view model and vice versa



On The Other Hand…

- More code required
- Not supported out-of-the-box
- Overkill for basic projects
  - You forget about this once you are used to MVVM!

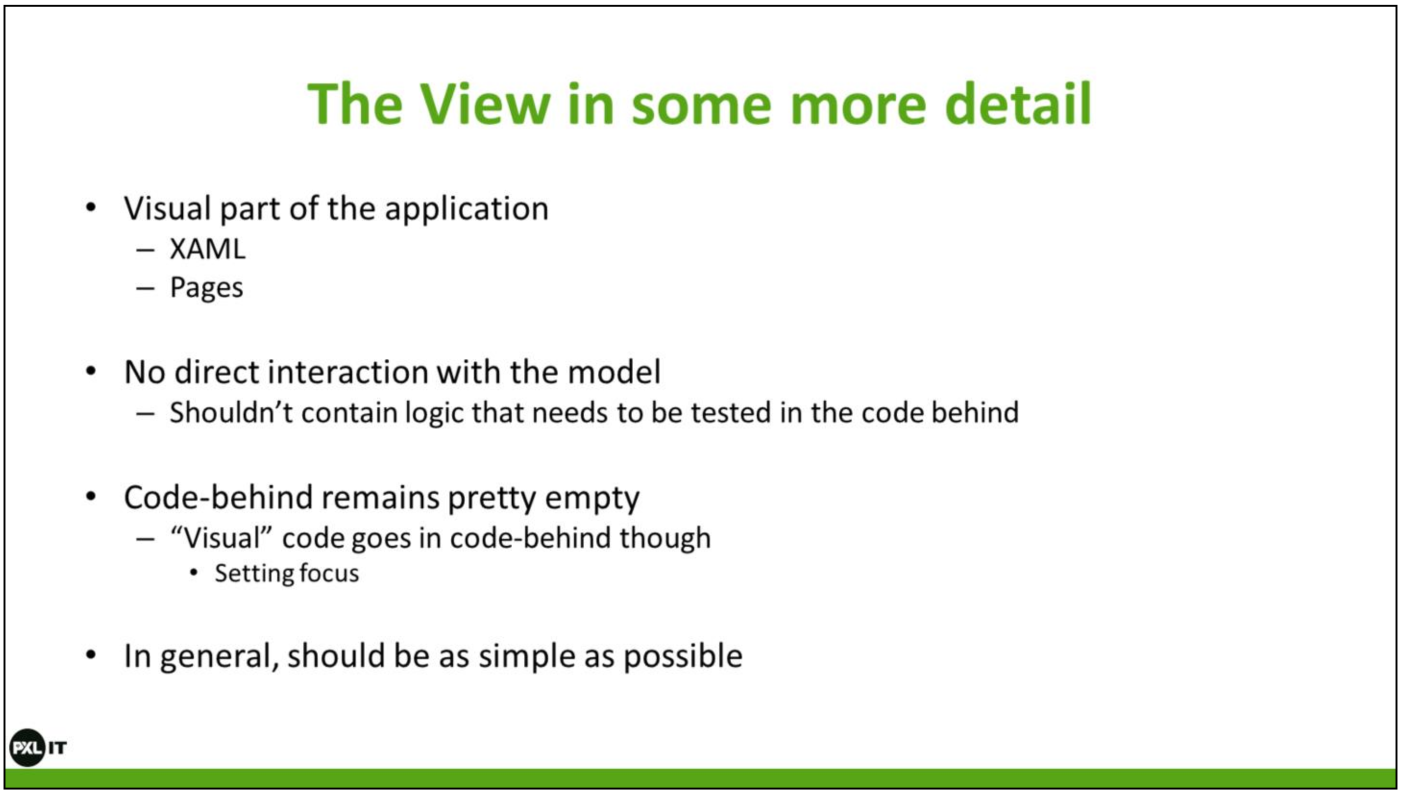
BUILDING BLOCKS OF MVVM



The pattern

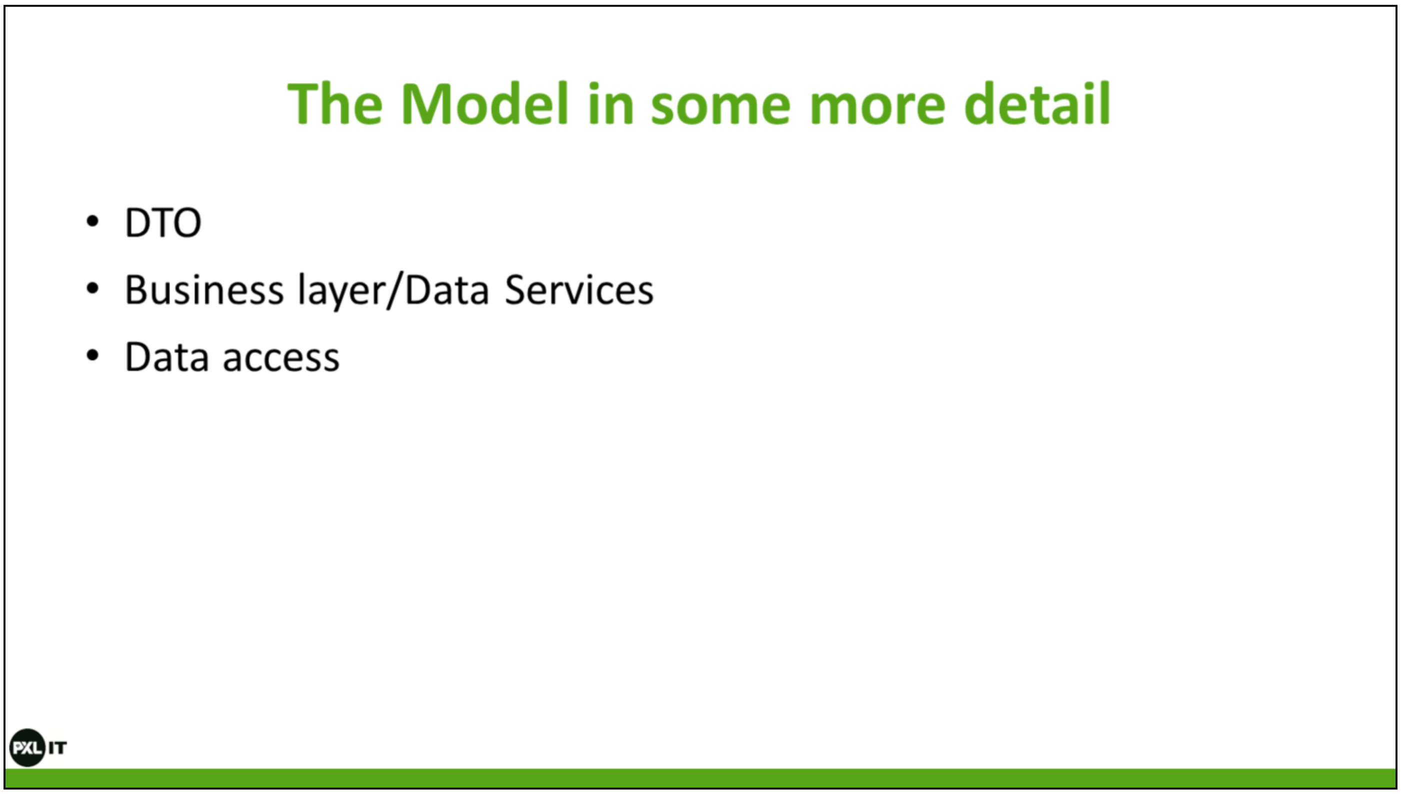
- V is part for the designer
- M is for the developer
- VM glues both these objects together
  - V can bind to properties of VM
  - VM remodels data so it is usable for the V

PXL IT



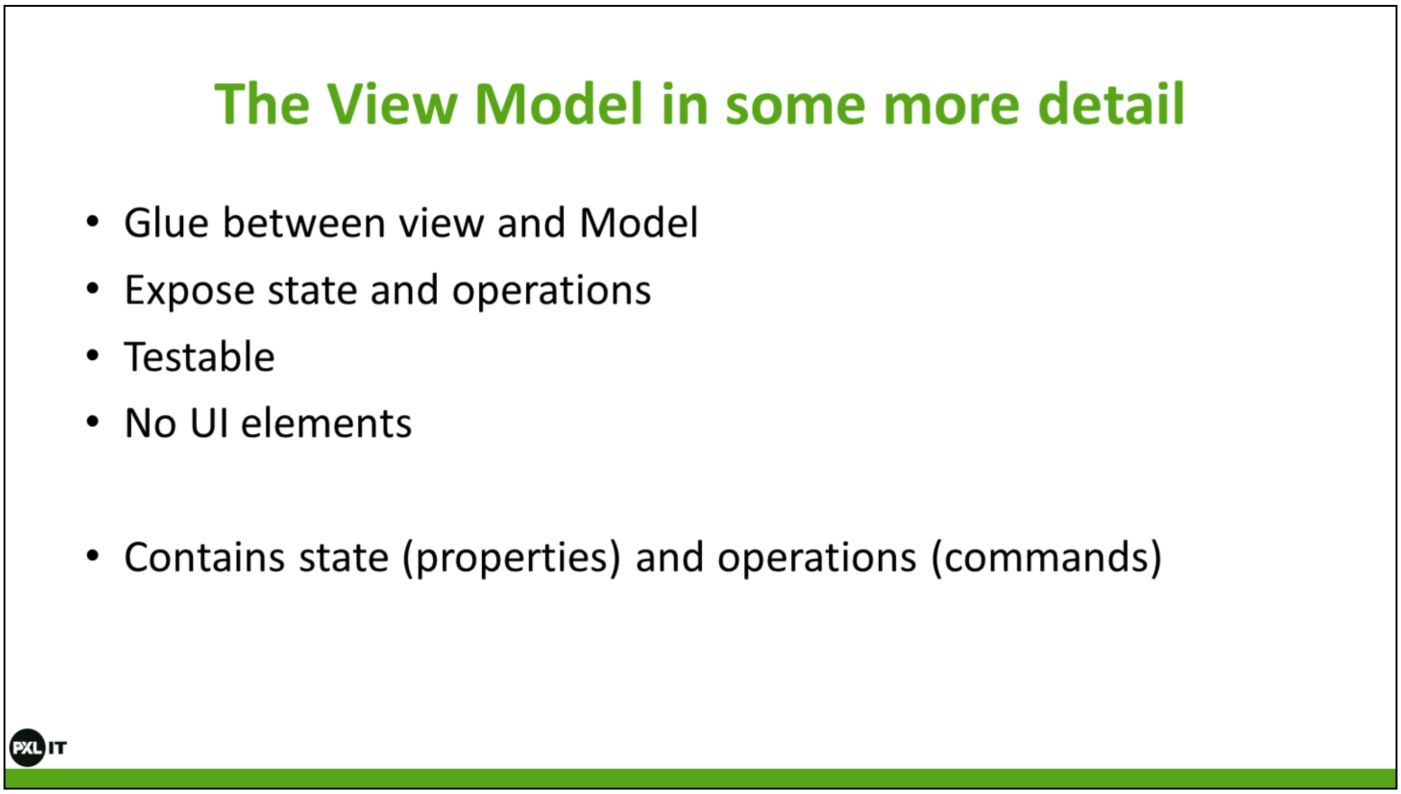
The View in some more detail

- Visual part of the application
  - XAML
  - Pages
- No direct interaction with the model
  - Shouldn't contain logic that needs to be tested in the code behind
- Code-behind remains pretty empty
  - "Visual" code goes in code-behind though
    - Setting focus
- In general, should be as simple as possible



The Model in some more detail

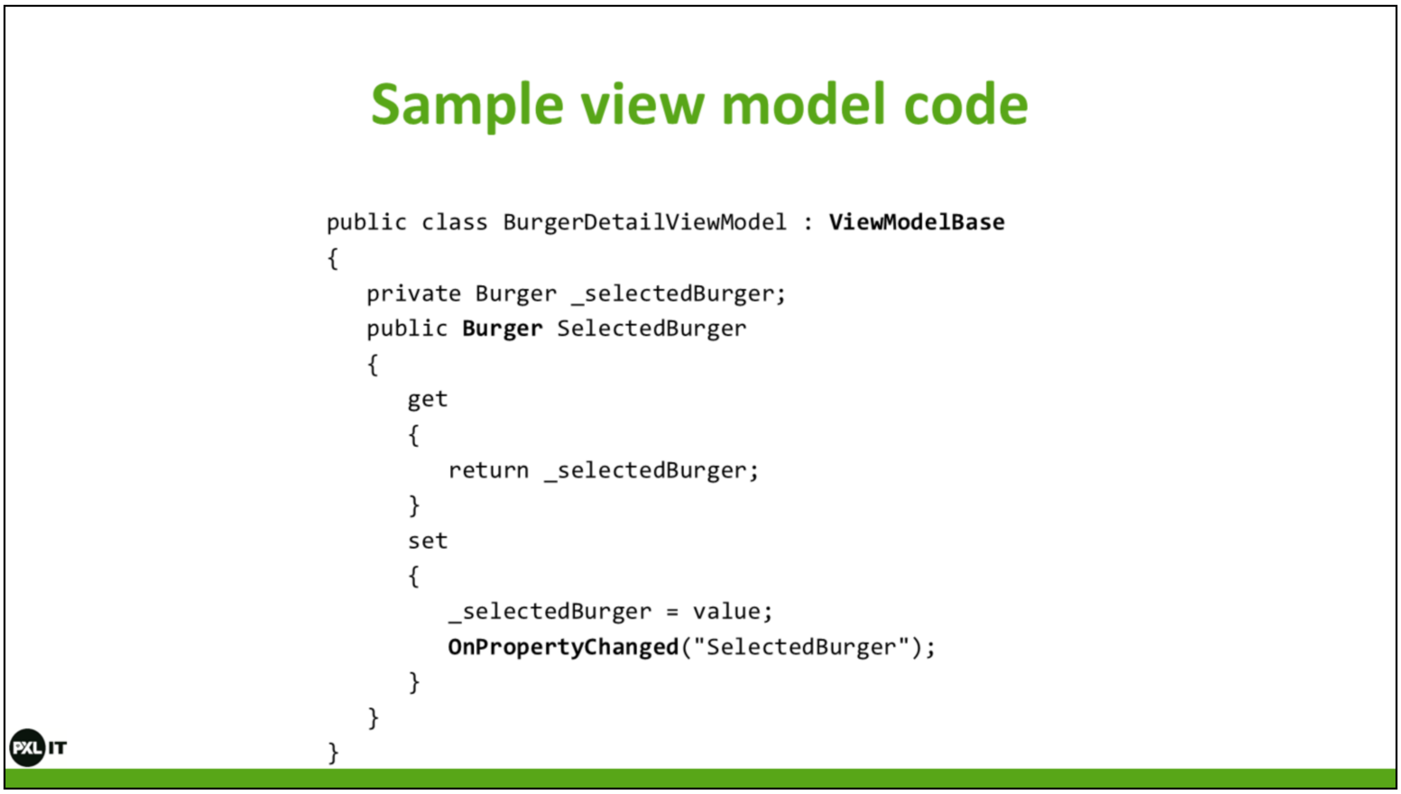
- DTO
- Business layer/Data Services
- Data access



The View Model in some more detail

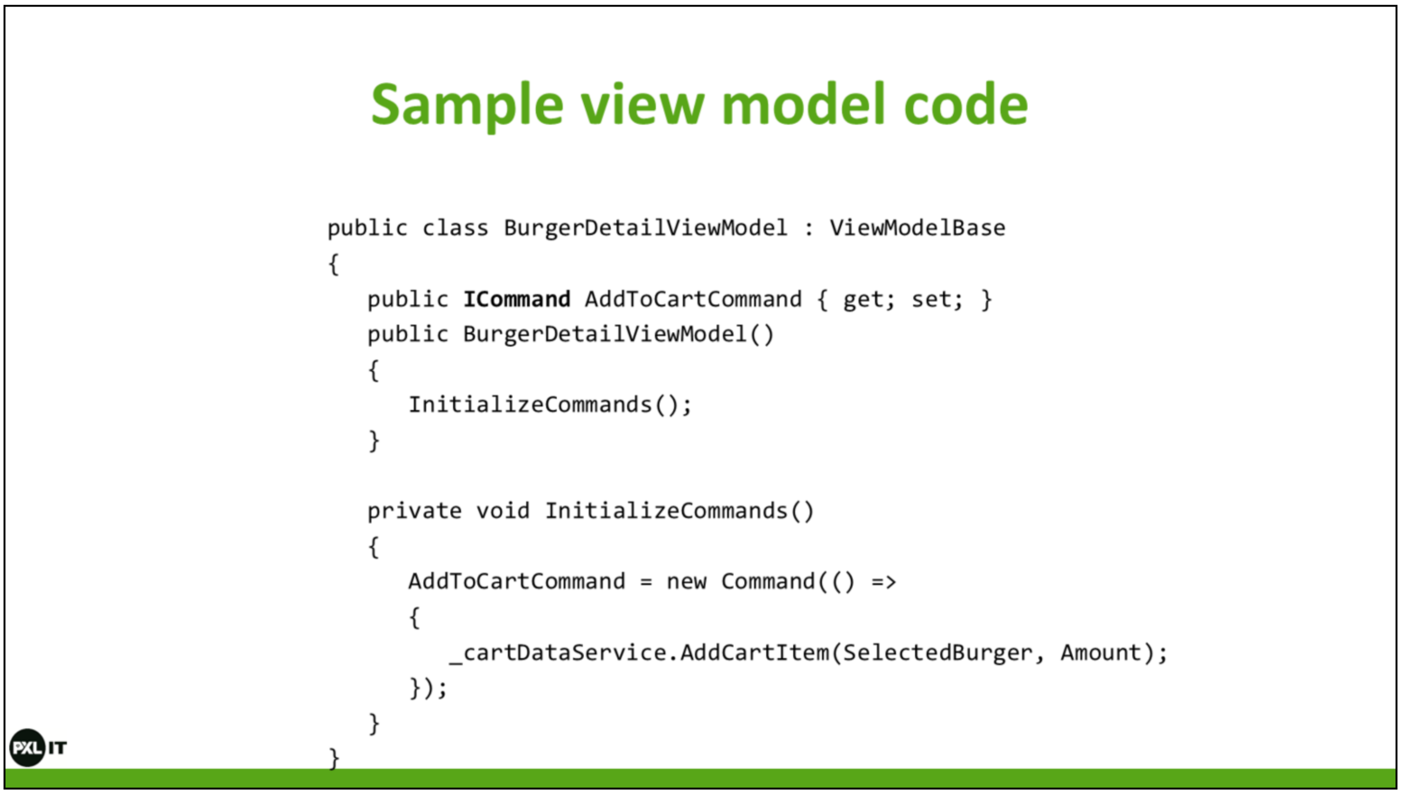
- Glue between view and Model
- Expose state and operations
- Testable
- No UI elements

- Contains state (properties) and operations (commands)



Sample view model code

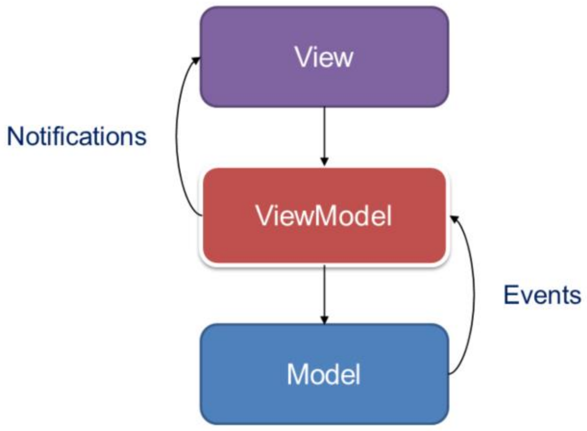
```
public class BurgerDetailViewModel : ViewModelBase
{
   private Burger _selectedBurger;
   public Burger SelectedBurger
   {
      get
      {
         return _selectedBurger;
      }
      set
      {
         _selectedBurger = value;
         OnPropertyChanged("SelectedBurger");
      }
   }
}
```



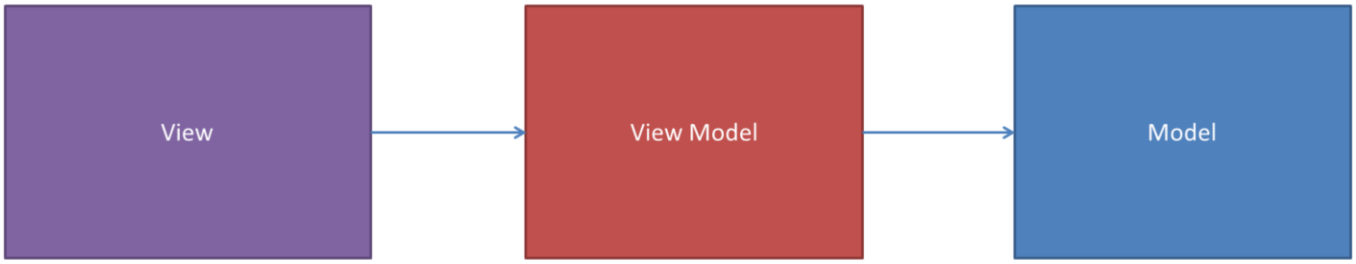
Sample view model code

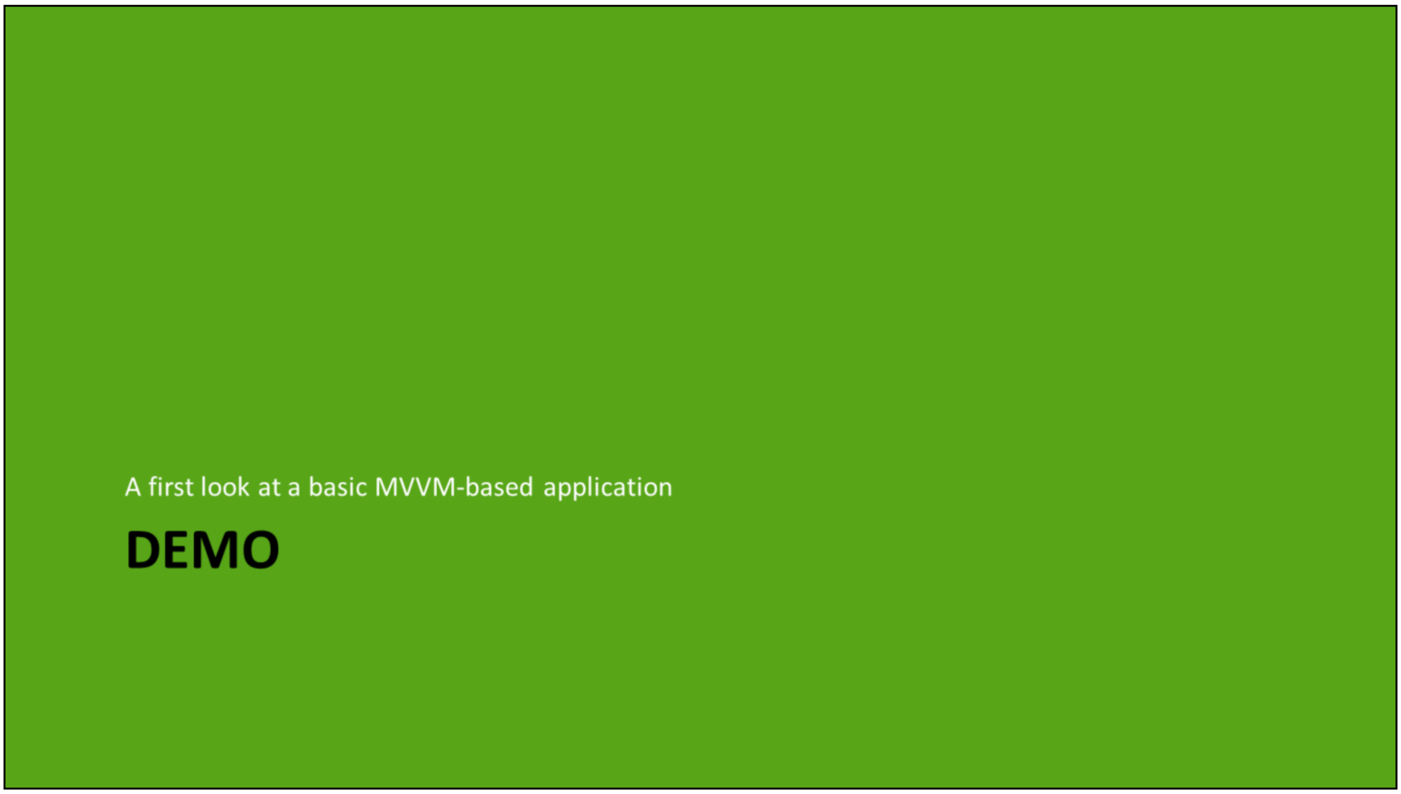
```
public class BurgerDetailViewModel : ViewModelBase
{
   public ICommand AddToCartCommand { get; set; }
   public BurgerDetailViewModel()
   {
      InitializeCommands();
   }

   private void InitializeCommands()
   {
      AddToCartCommand = new Command(() =>
      {
         _cartDataService.AddCartItem(SelectedBurger, Amount);
      });
   }
}
```

The View Model in some more detail



Who knows who in MVVM





A first look at a basic MVVM-based application

DEMO

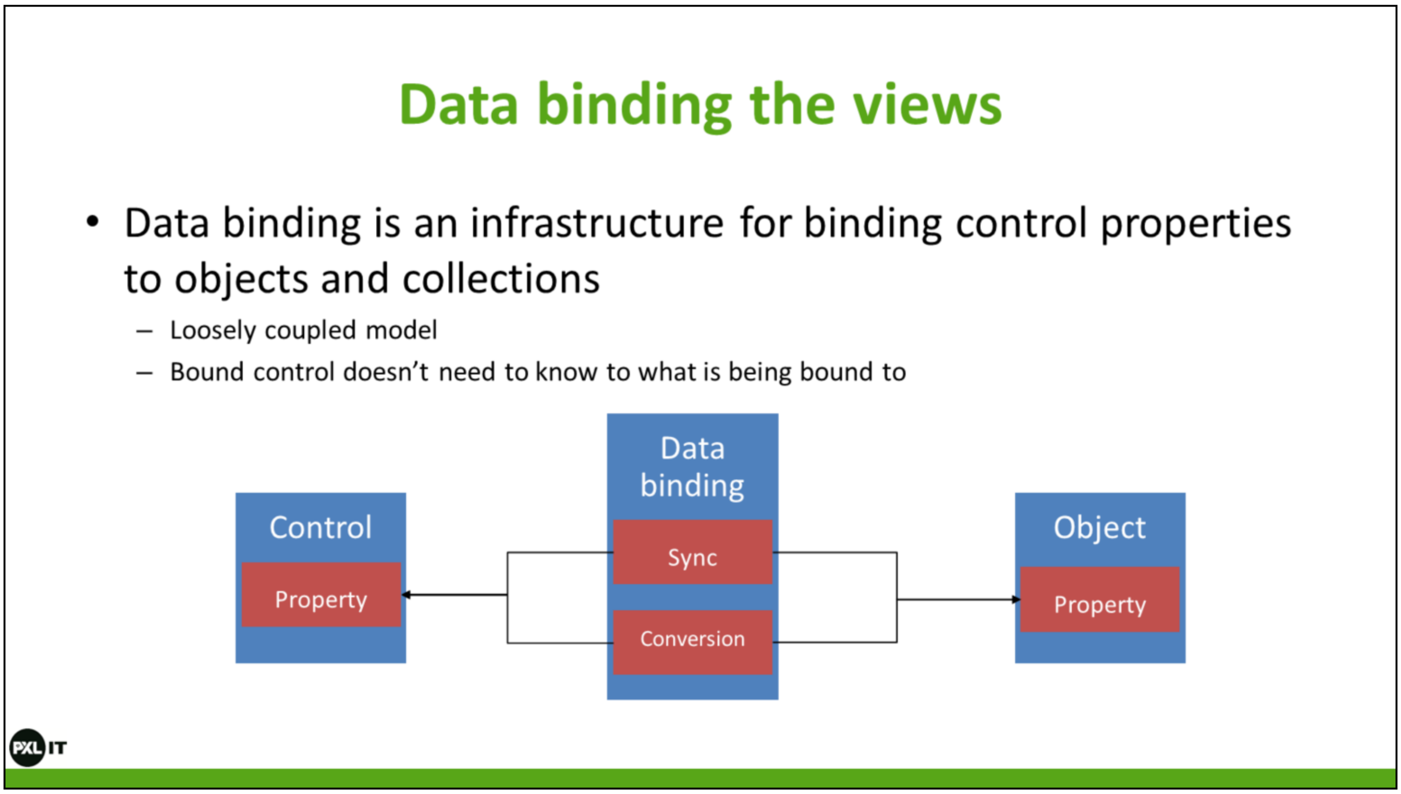
Demo 1

Show the running app here

Focus on the MVVM building blocks

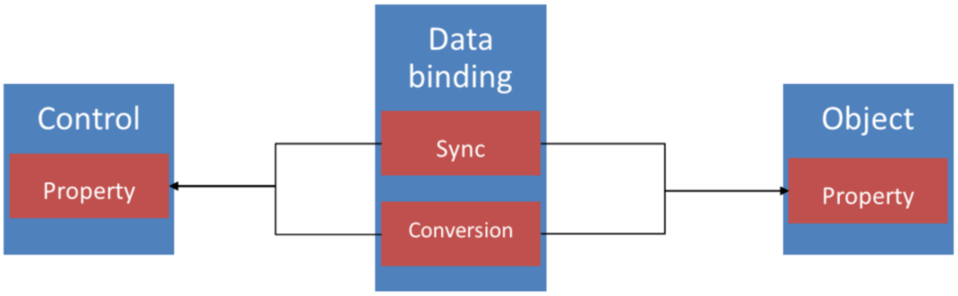
Show the view models containing state and operations

Show that they are standalone with only links to models (currently via the data services)



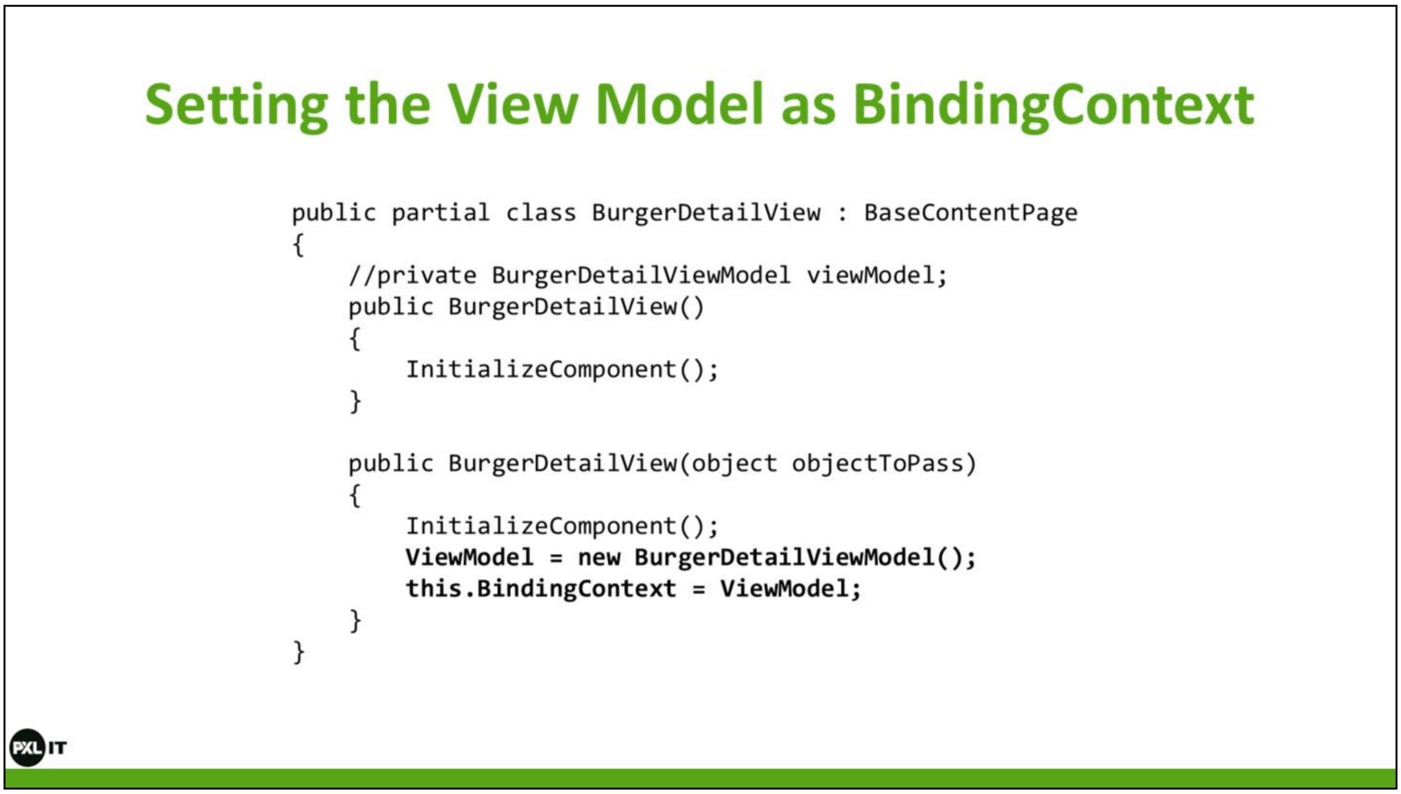
Data binding the views

- Data binding is an infrastructure for binding control properties to objects and collections
  - Loosely coupled model
  - Bound control doesn't need to know to what is being bound to



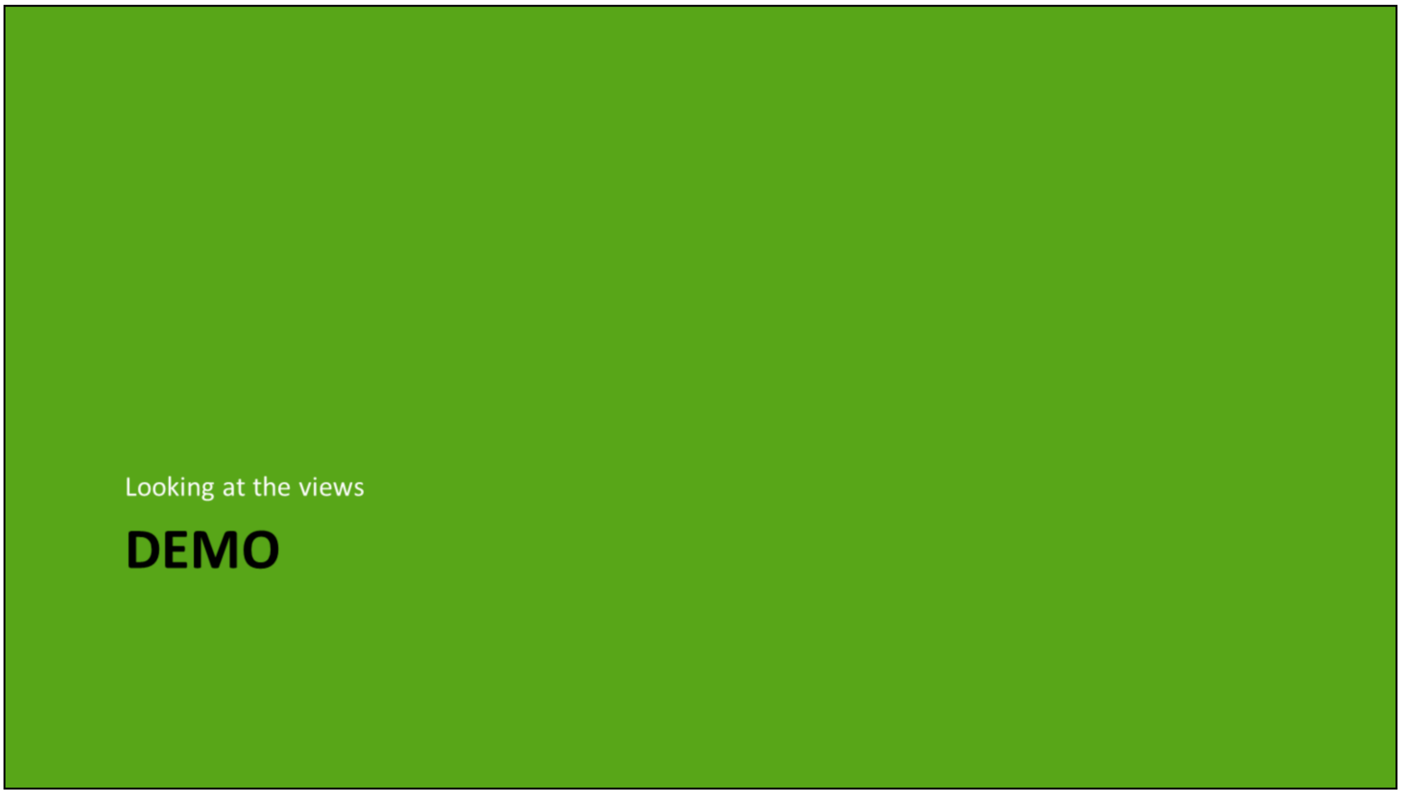
Binding the view to properties on the View Model

```
<Label Text="{Binding SelectedBurger.Name}" />
<Label Text="{Binding SelectedBurger.ShortDescription}" />
<Label Text="{Binding SelectedBurger.Price}" />
```



Setting the View Model as BindingContext

```
public partial class BurgerDetailView : BaseContentPage
{
    //private BurgerDetailViewModel viewModel;
    public BurgerDetailView()
    {
        InitializeComponent();
    }

    public BurgerDetailView(object objectToPass)
    {
        InitializeComponent();
        ViewModel = new BurgerDetailViewModel();
        this.BindingContext = ViewModel;
    }
}
```

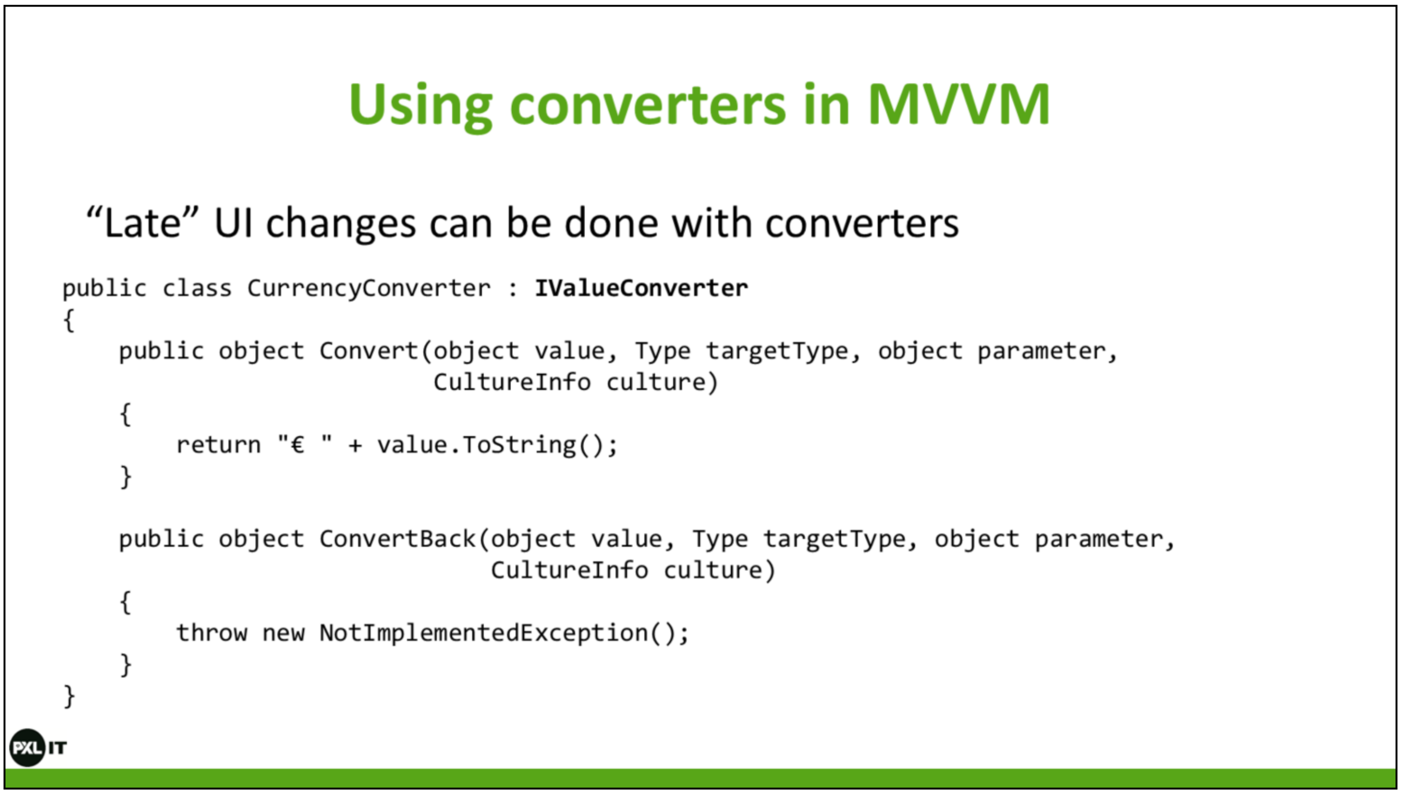


Looking at the views

DEMO

Show here how the view models are linked to the view models

In the demo, the view models are retrieved using IOC, this is added later. For now, just explain that this can be done in a one-on-one way.

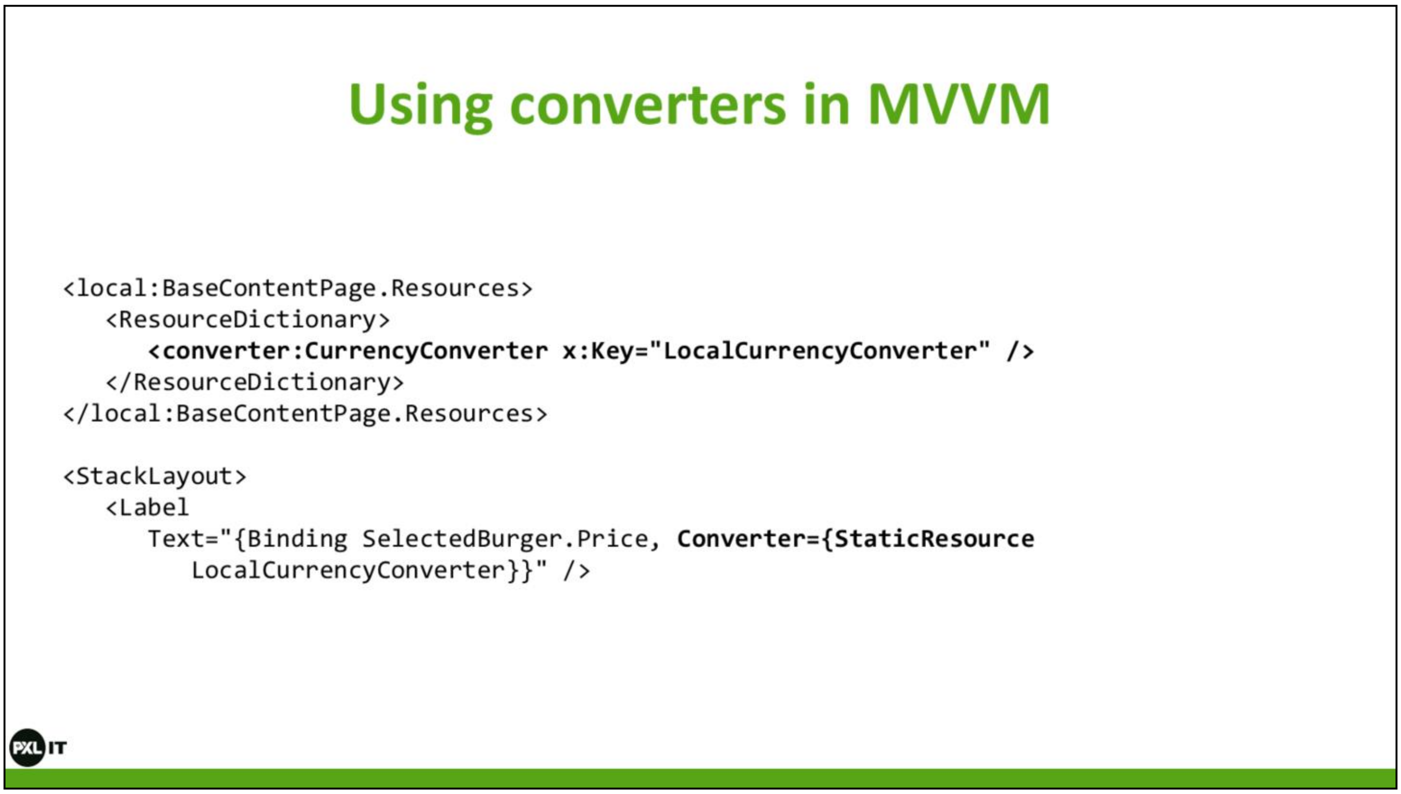


Using converters in MVVM

"Late" UI changes can be done with converters

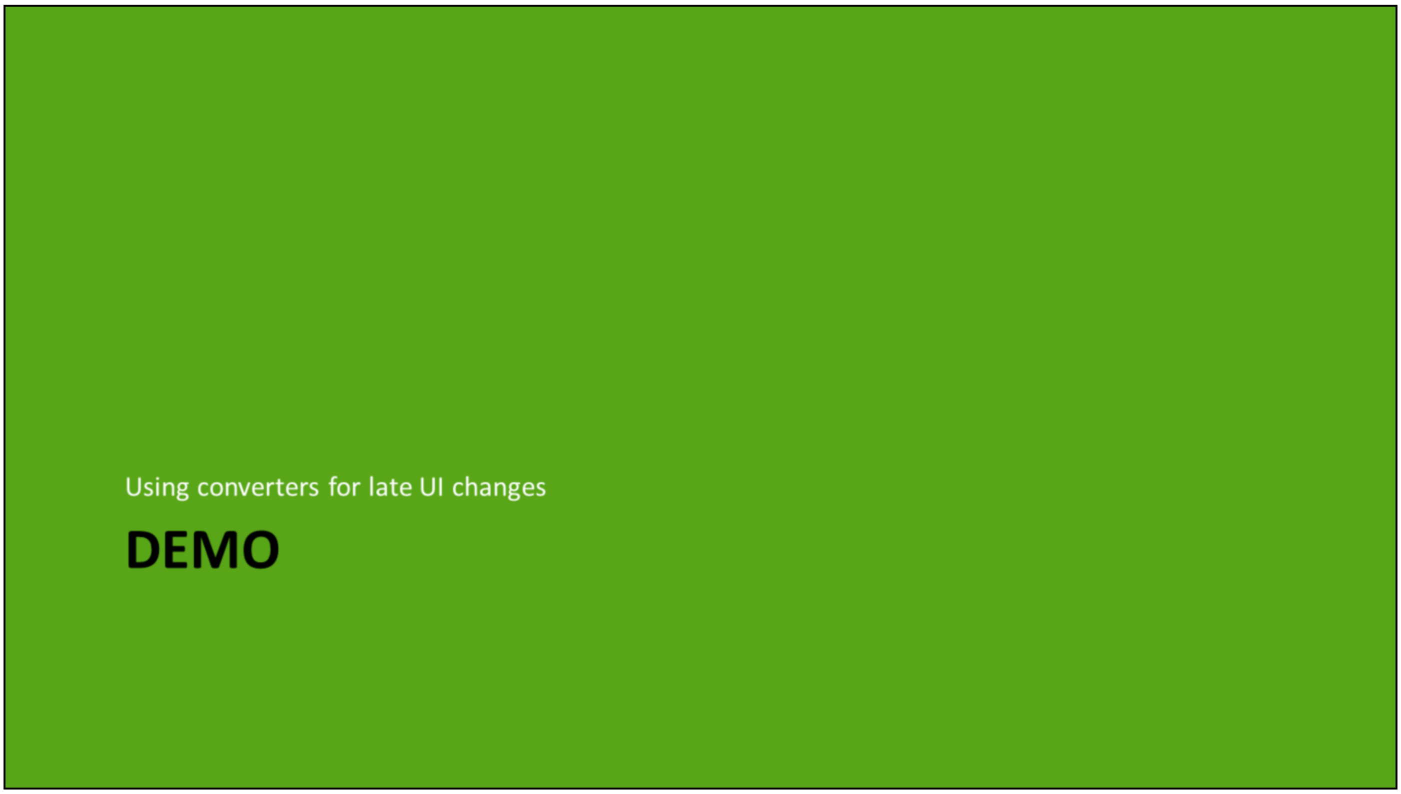
```
public class CurrencyConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
                          CultureInfo culture)
    {
        return "€ " + value.ToString();
    }

    public object ConvertBack(object value, Type targetType, object parameter,
                              CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Explain that converters are often used for late UI updates in MVVM. This is a practice which is by some considered bad practice but is often done



Using converters in MVVM

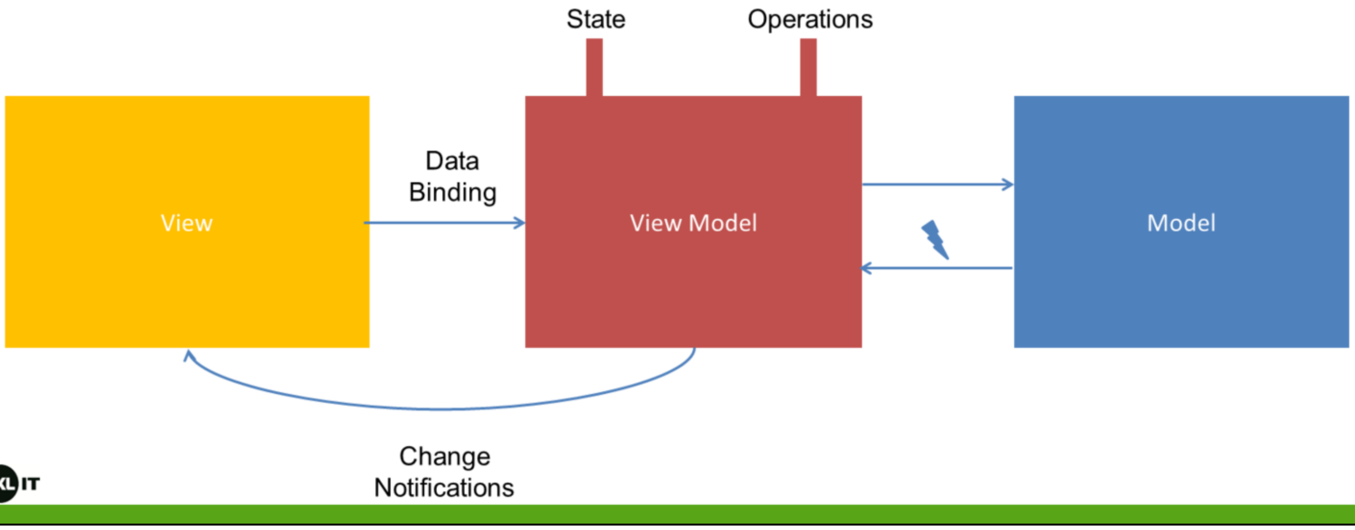
```
<local:BaseContentPage.Resources>
   <ResourceDictionary>
      <converter:CurrencyConverter x:Key="LocalCurrencyConverter" />
   </ResourceDictionary>
</local:BaseContentPage.Resources>

<StackLayout>
   <Label
      Text="{Binding SelectedBurger.Price, Converter={StaticResource
         LocalCurrencyConverter}}" />
```

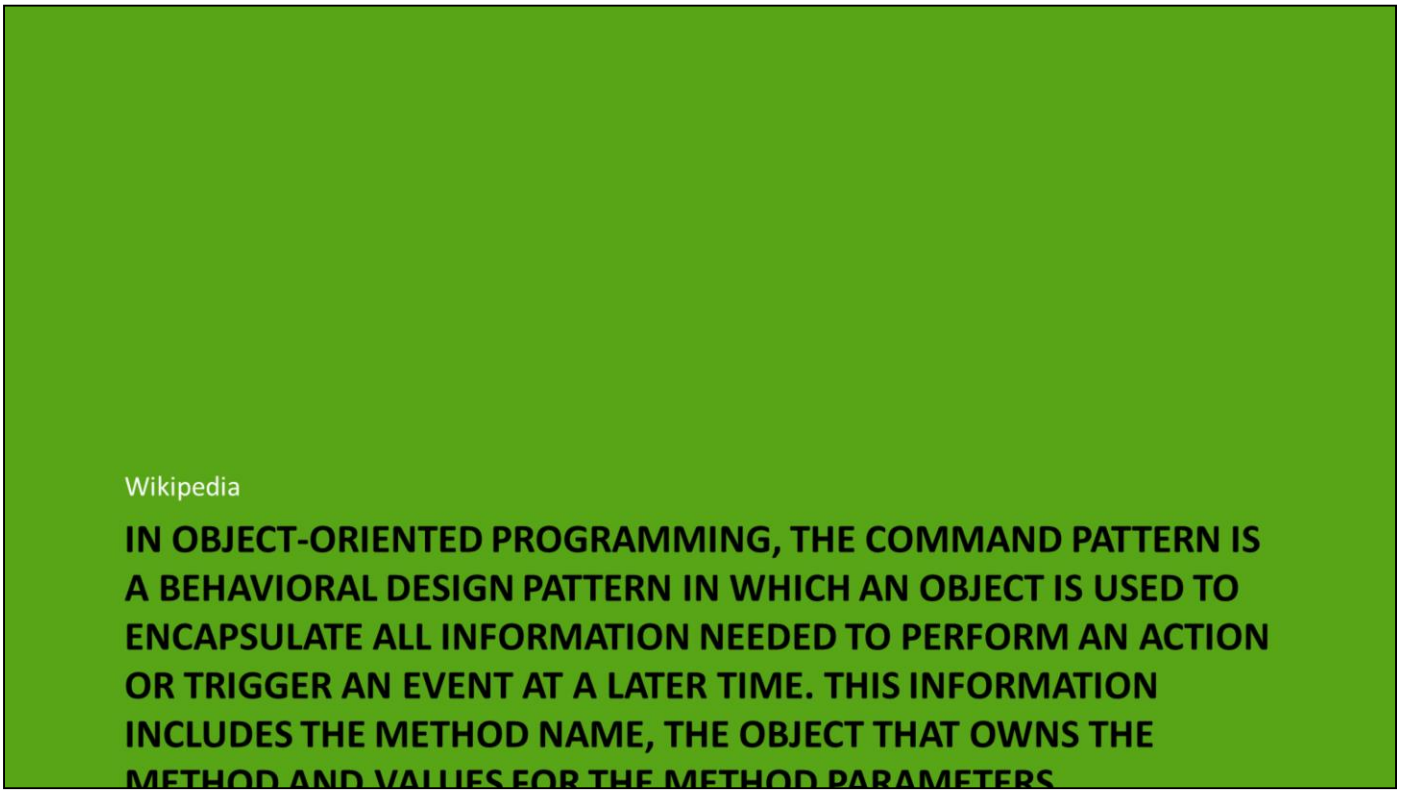

Using converters for late UI changes

DEMO

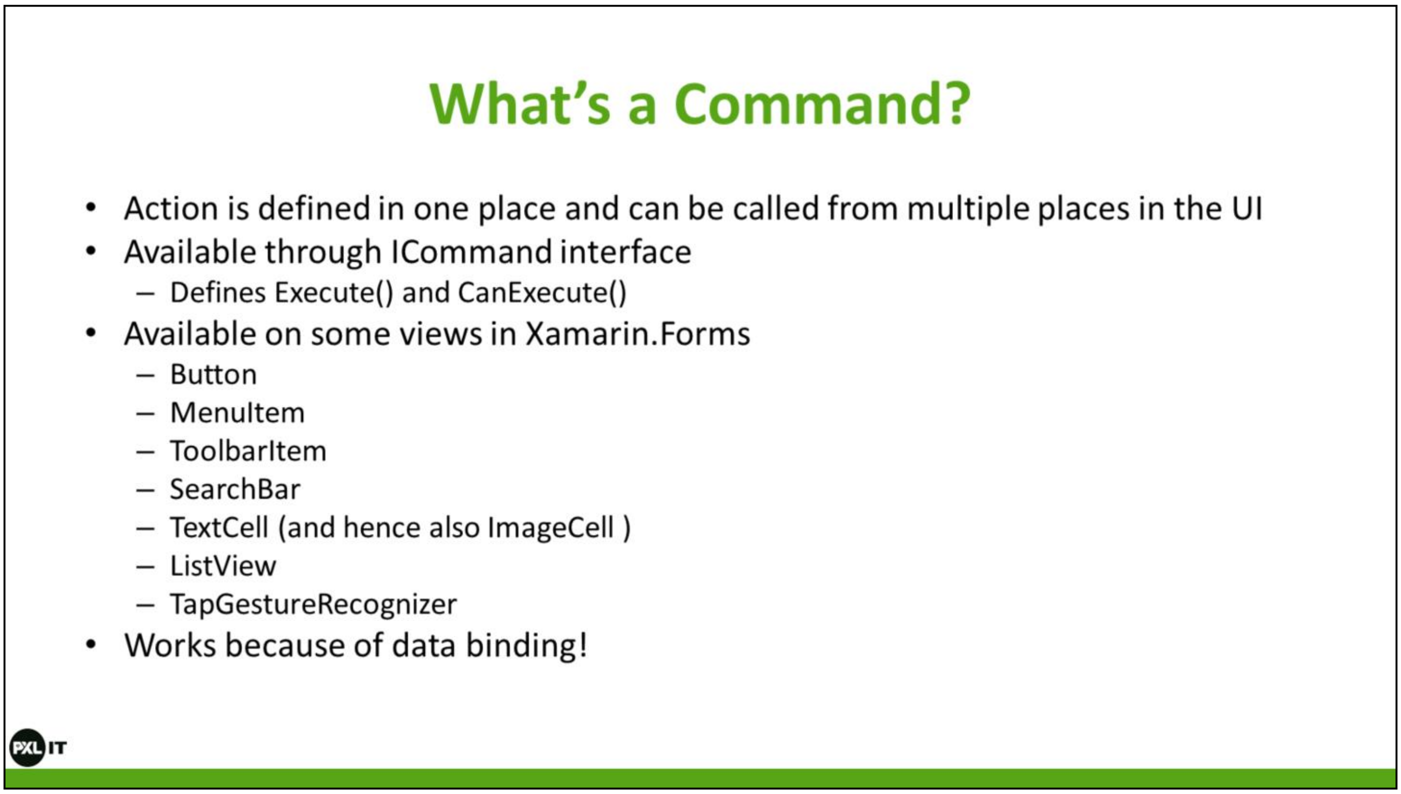
We’re still missing something

How can we interact from the UI with code that lives in the VM?




Wikipedia

IN OBJECT-ORIENTED PROGRAMMING, THE COMMAND PATTERN IS A BEHAVIORAL DESIGN PATTERN IN WHICH AN OBJECT IS USED TO ENCAPSULATE ALL INFORMATION NEEDED TO PERFORM AN ACTION OR TRIGGER AN EVENT AT A LATER TIME. THIS INFORMATION INCLUDES THE METHOD NAME, THE OBJECT THAT OWNS THE METHOD AND VALUES FOR THE METHOD PARAMETERS



What's a Command?

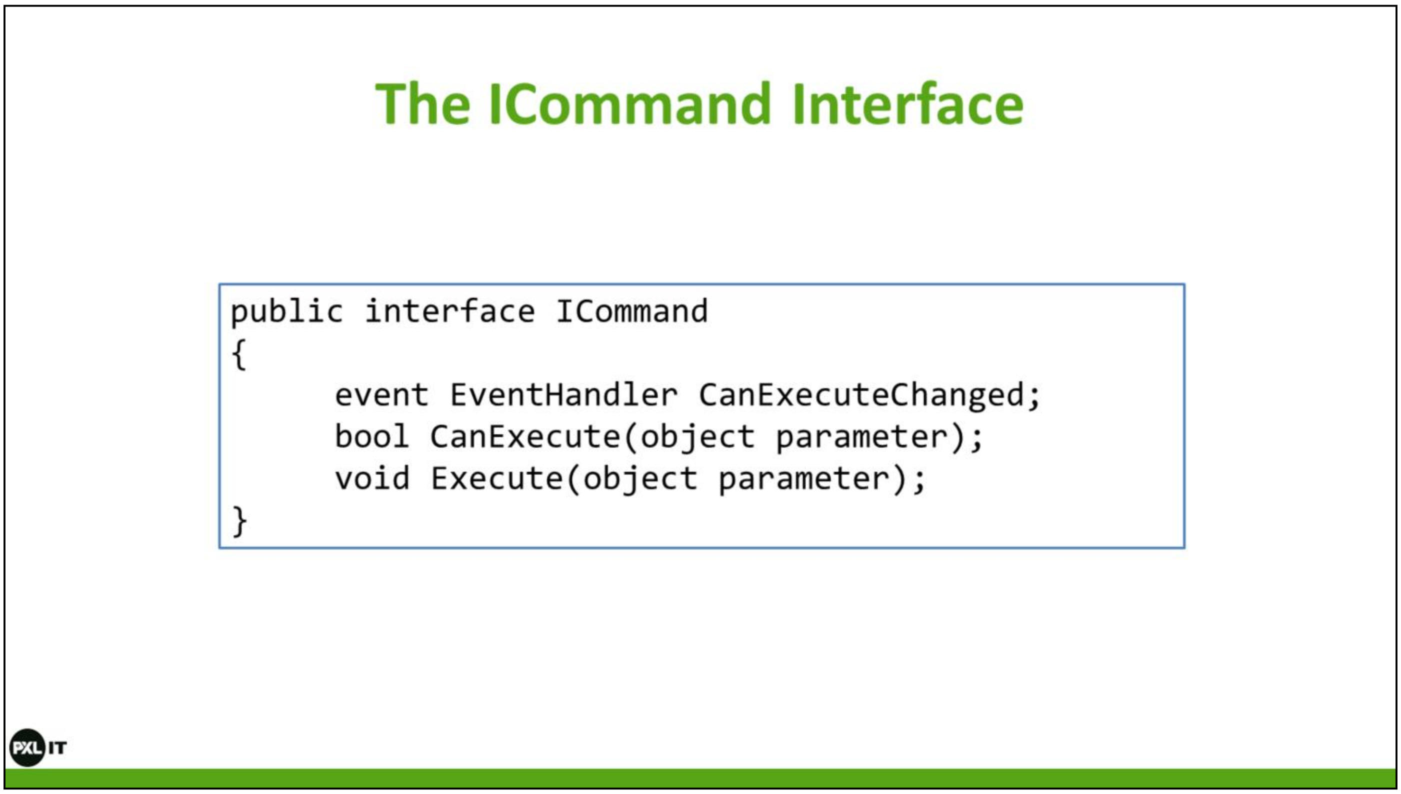
- Action is defined in one place and can be called from multiple places in the UI
- Available through ICommand interface
  - Defines Execute() and CanExecute()
- Available on some views in Xamarin.Forms
  - Button
  - MenuItem
  - ToolbarItem
  - SearchBar
  - TextCell (and hence also ImageCell)
  - ListView
  - TapGestureRecognizer
- Works because of data binding!

In many cases, the MVVM pattern is restricted to the manipulation of data items: User-interface objects in the View parallel data objects in the ViewModel.

Sometimes, however, the View needs to contain buttons that trigger various actions in the ViewModel. But the ViewModel must not contain Clicked handlers for the buttons because that would tie the ViewModel to a particular user-interface paradigm.

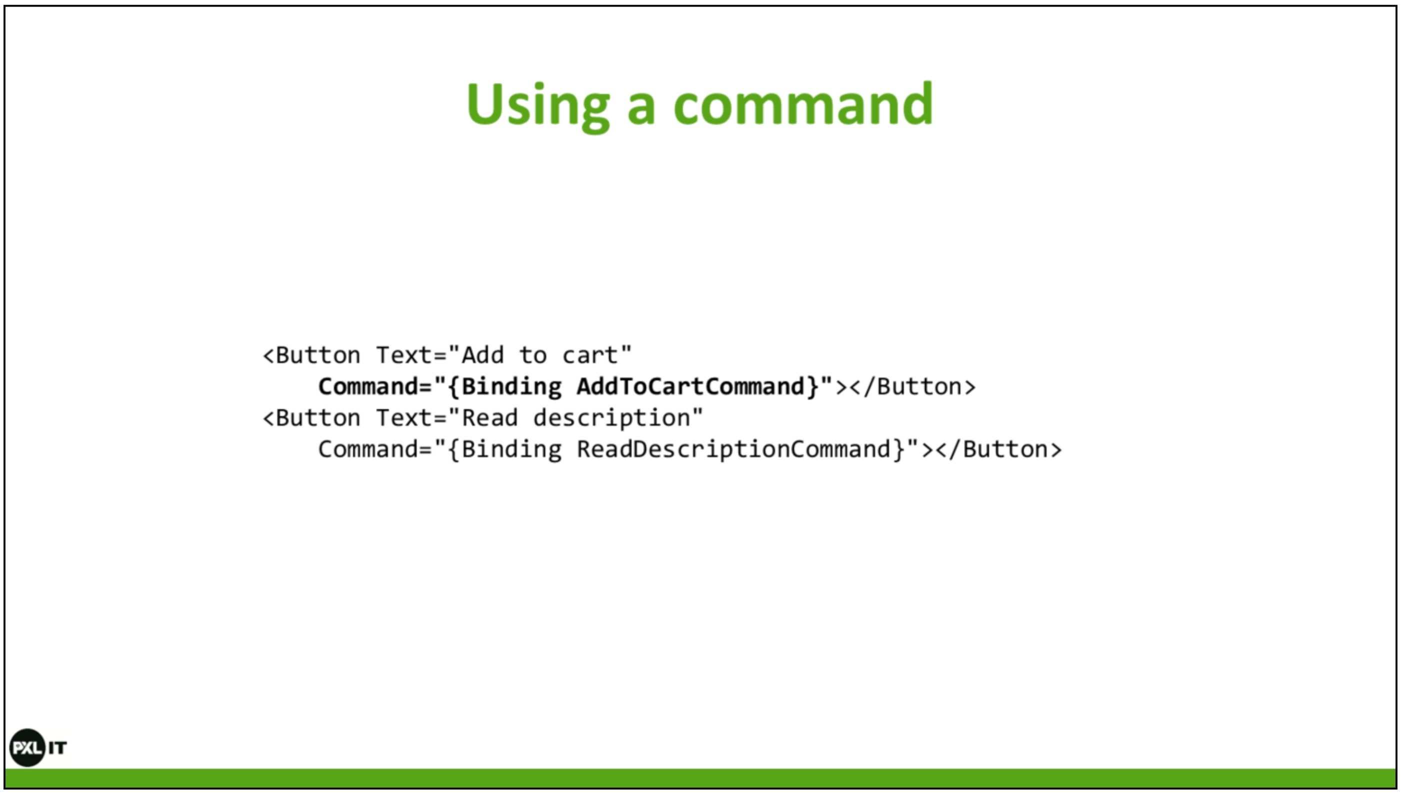
To allow ViewModels to be more independent of particular user interface objects but still allow methods to be called within the ViewModel, a *command* interface was developed. This command interface is supported by the following elements in Xamarin.Forms:

- Button
- MenuItem
- ToolbarItem
- SearchBar
- TextCell (and hence also ImageCell)
- ListView
- TapGestureRecognizer



The ICommand Interface

```
public interface ICommand
{
     event EventHandler CanExecuteChanged;
     bool CanExecute(object parameter);
     void Execute(object parameter);
}
```



Using a command

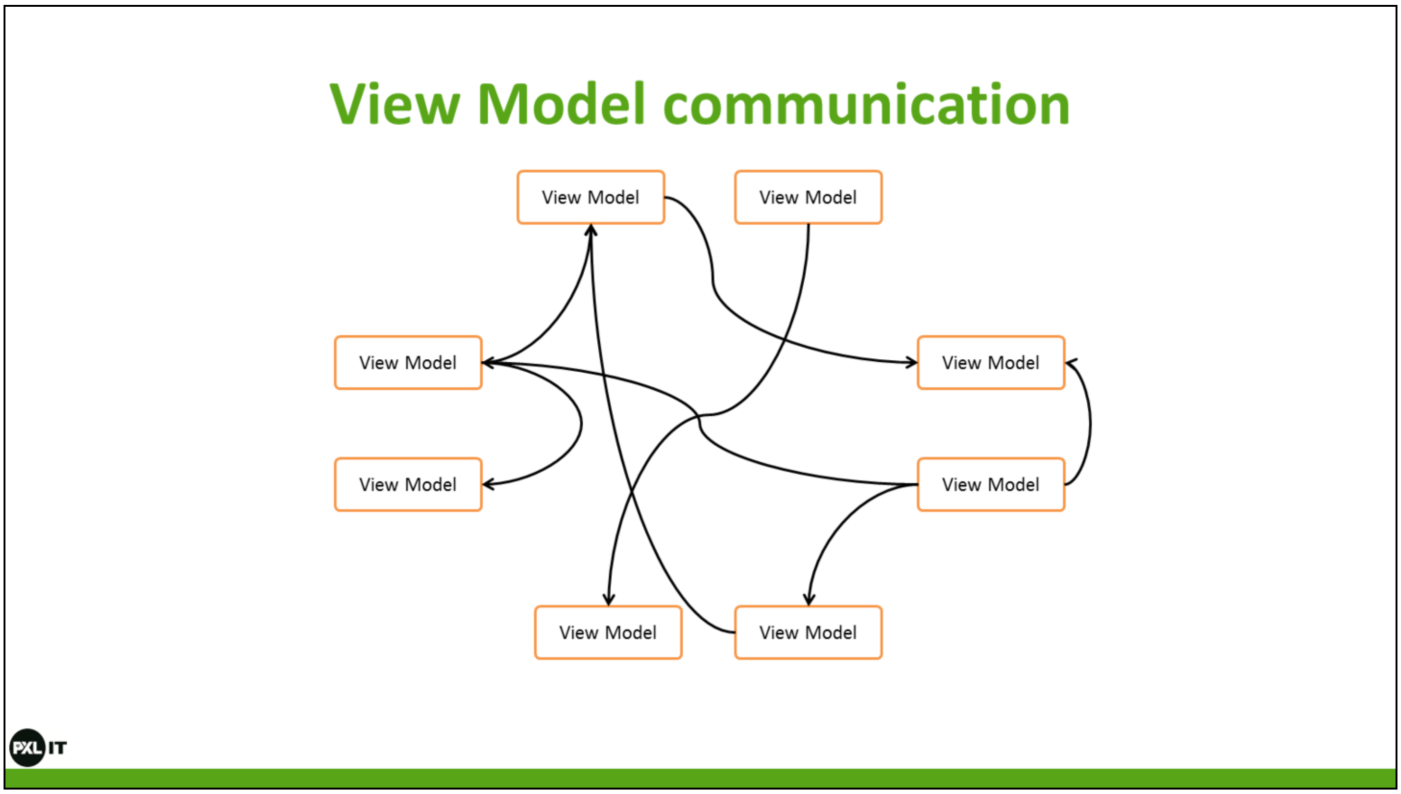
```
<Button Text="Add to cart"
    Command="{Binding AddToCartCommand}"></Button>
<Button Text="Read description"
    Command="{Binding ReadDescriptionCommand}"></Button>
```

Sadly, the don’t work on every control… Behaviors to the rescue!

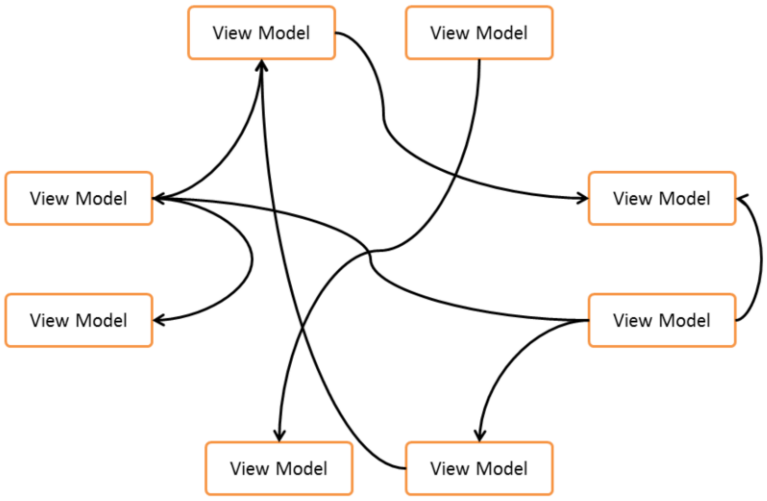
```
<ContentPage
    xmlns:local="clr-namespace:JoesBurgerStore;assembly=JoesBurgerStore">
...
<ListView x:Name="listView" IsGroupingEnabled="true" ItemsSource="{Binding
BurgerGroups}">
   <ListView.Behaviors>
      <utility:EventToCommandBehavior
         EventName="ItemSelected"
         Command="{Binding BurgerSelectedCommand}"
         Converter="{StaticResource localSelectedItemEventArgsToSelectedItemConverter}"
      />
   </ListView.Behaviors>
...
</ListView>
```

Looking at commands

DEMO

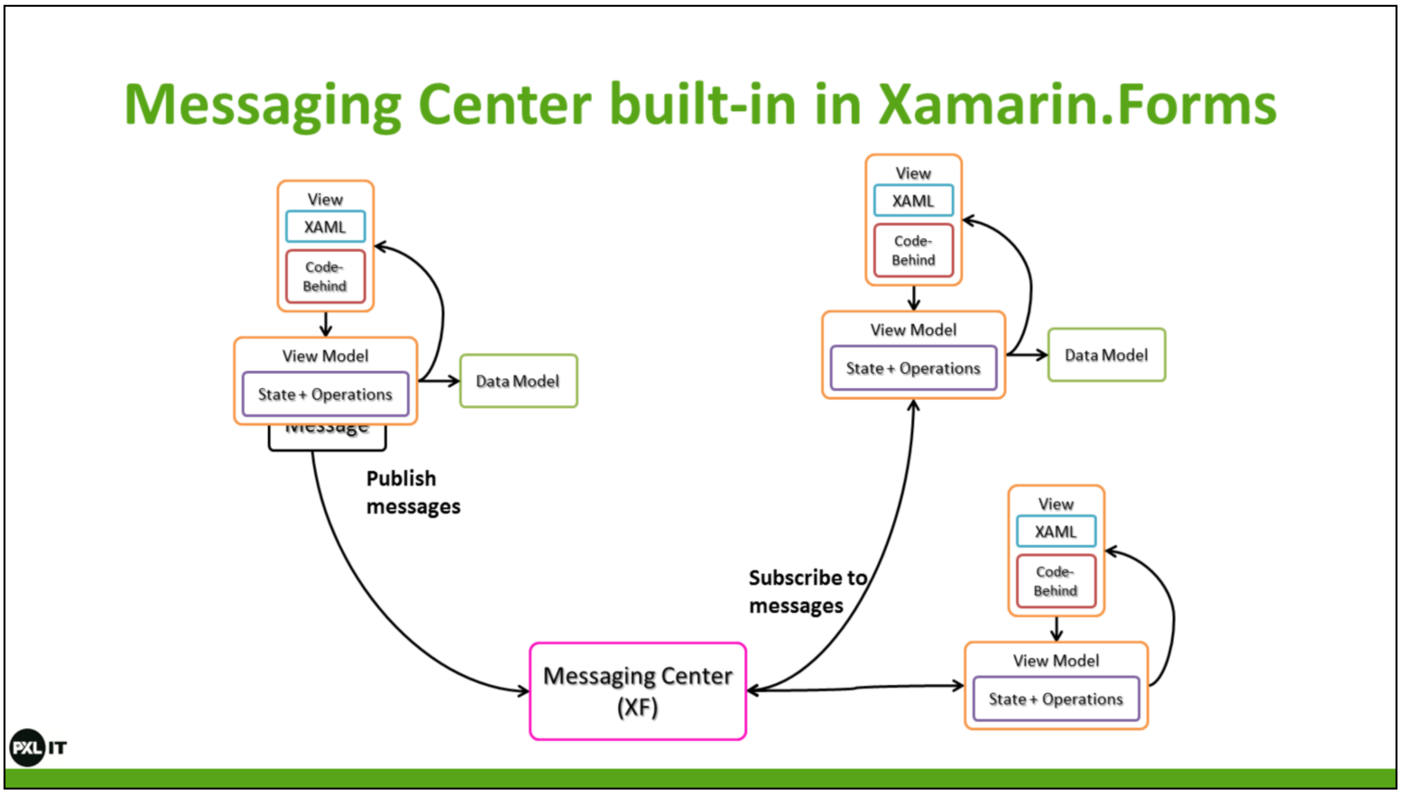


View Model communication

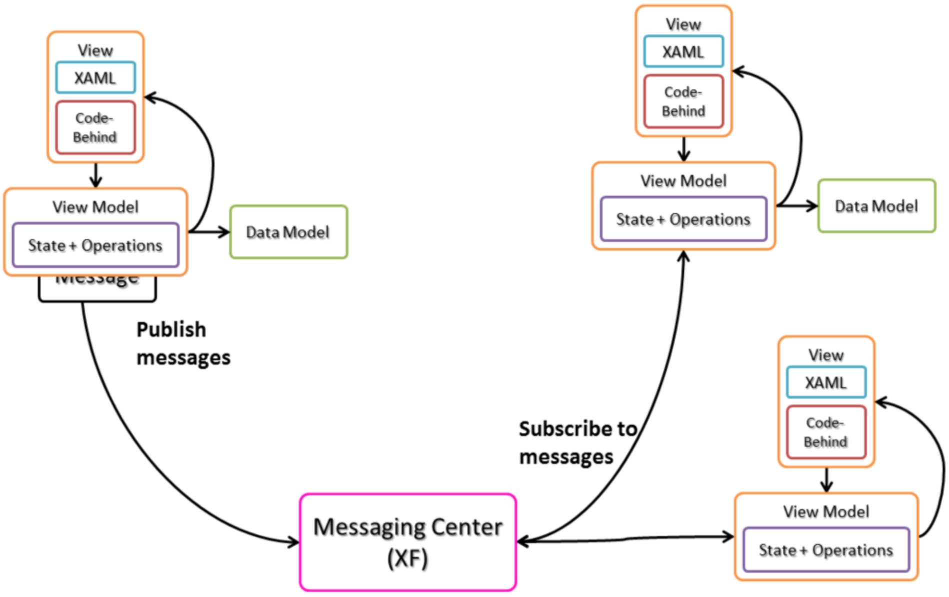


Explain that you don’t want to create hard references between the different view models

This is hard to test afterwards since it’s difficult to mock all the VM references.

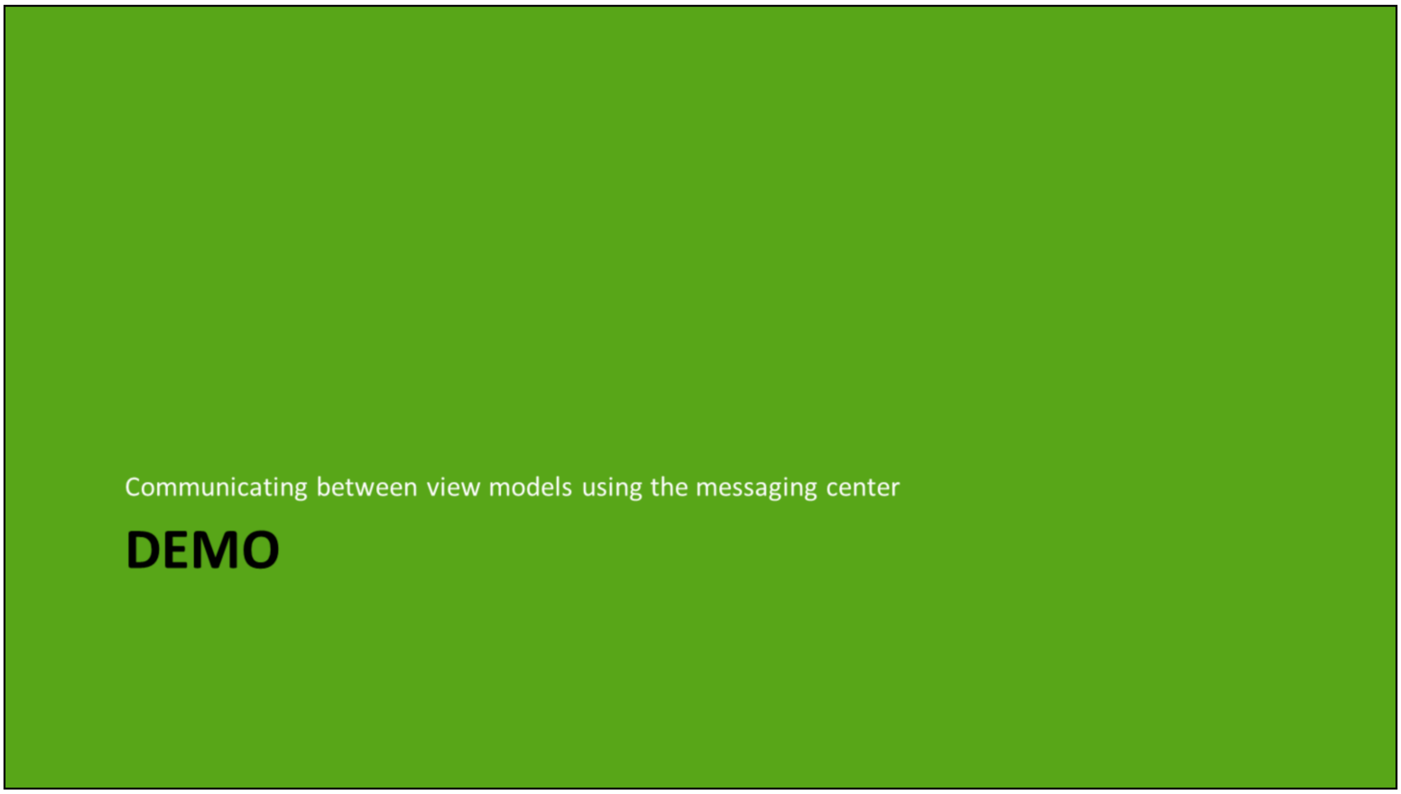


Messaging Center built-in in Xamarin.Forms



Solution: messaging using a mediator/messenger.

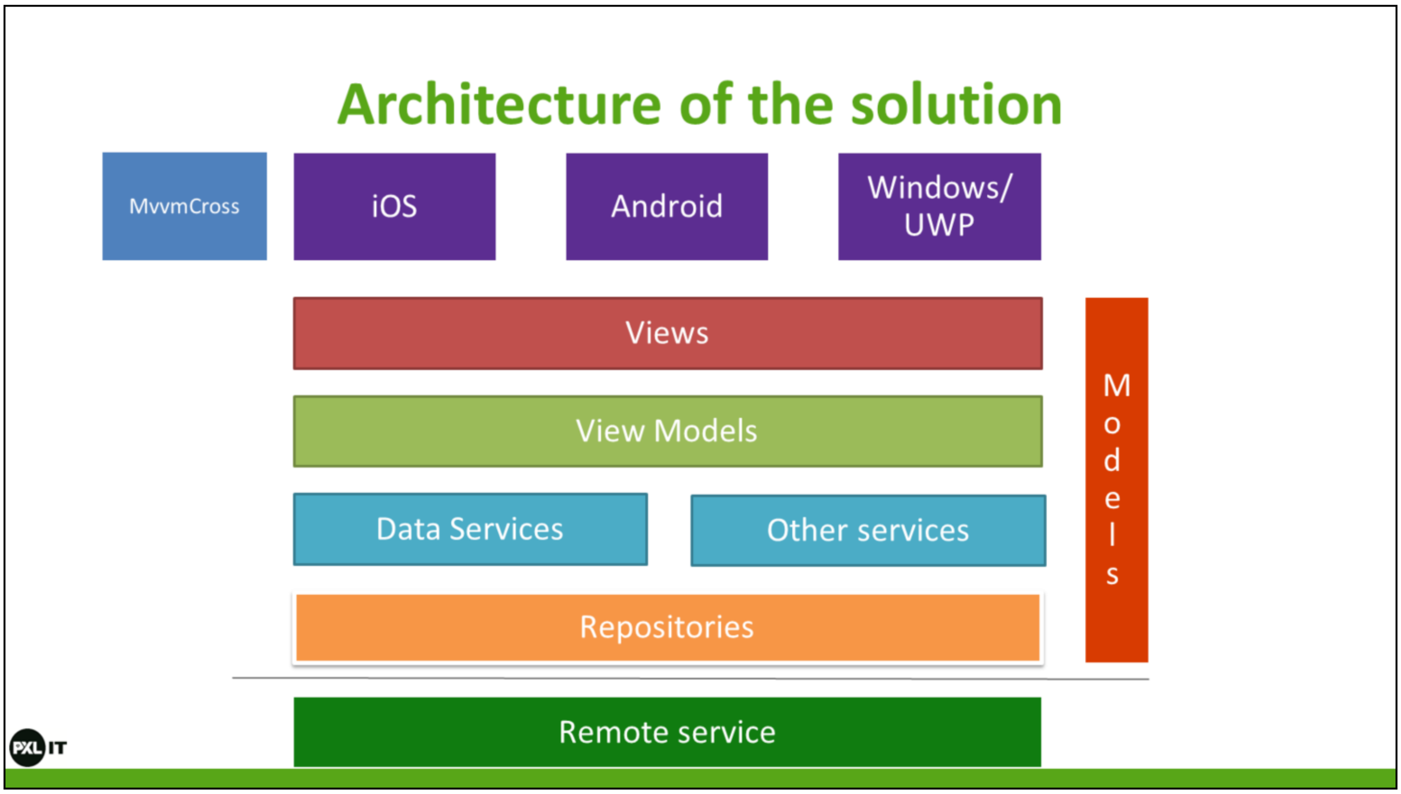
Xamarin Forms has a built-in mediator called the Messaging Center



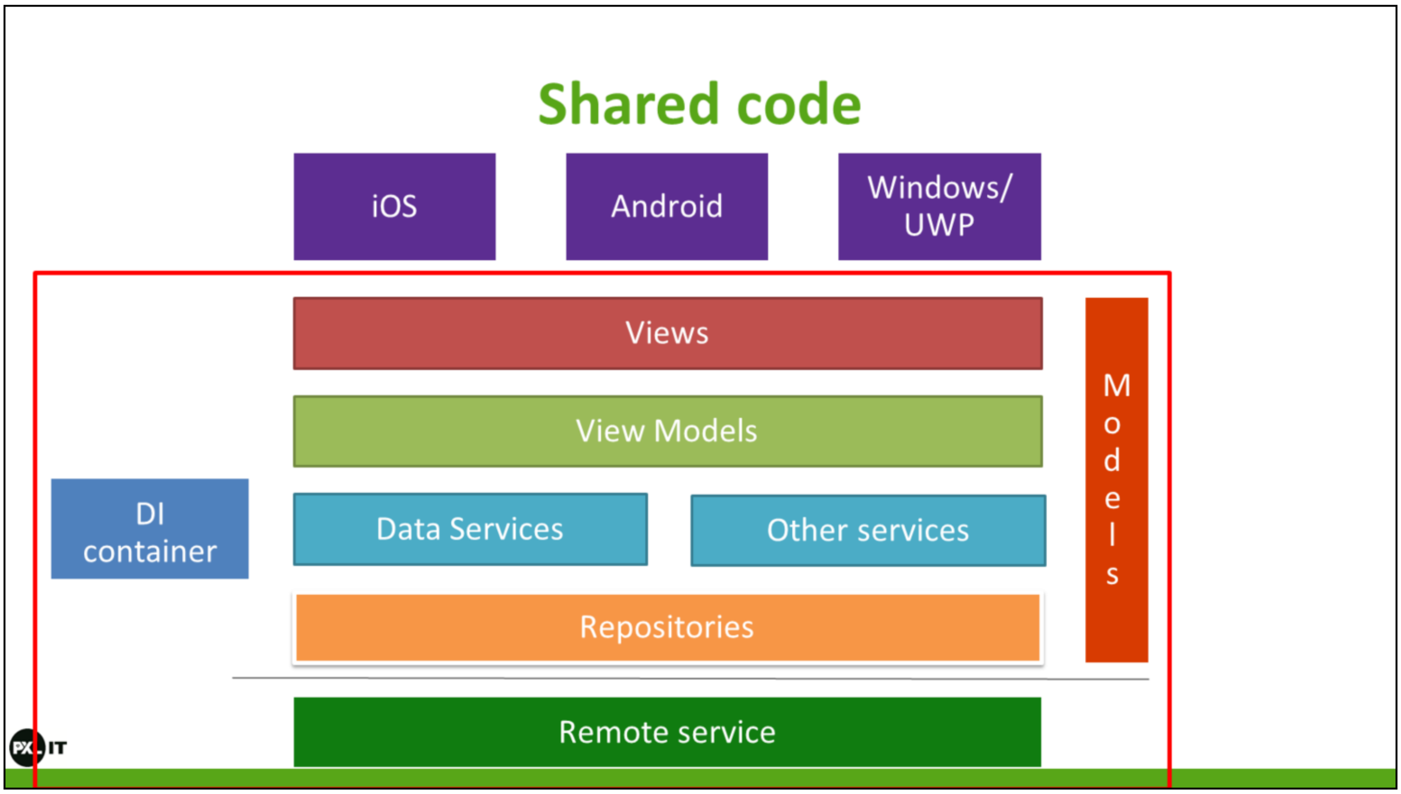
Communicating between view models using the messaging center

DEMO

EXPLORING A FULL MVVM-BASED XAMARIN.FORMS ARCHITECTURE

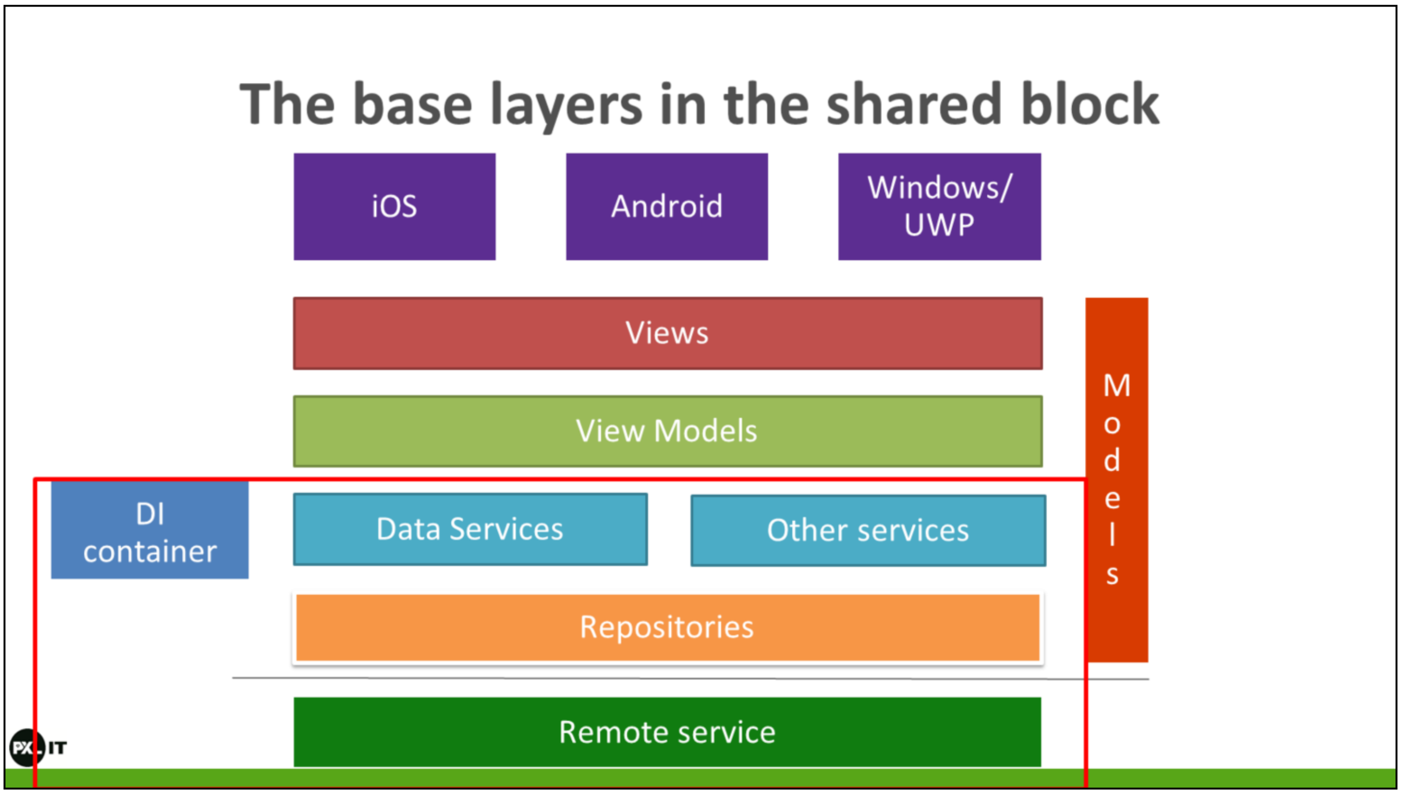


Architecture of the solution



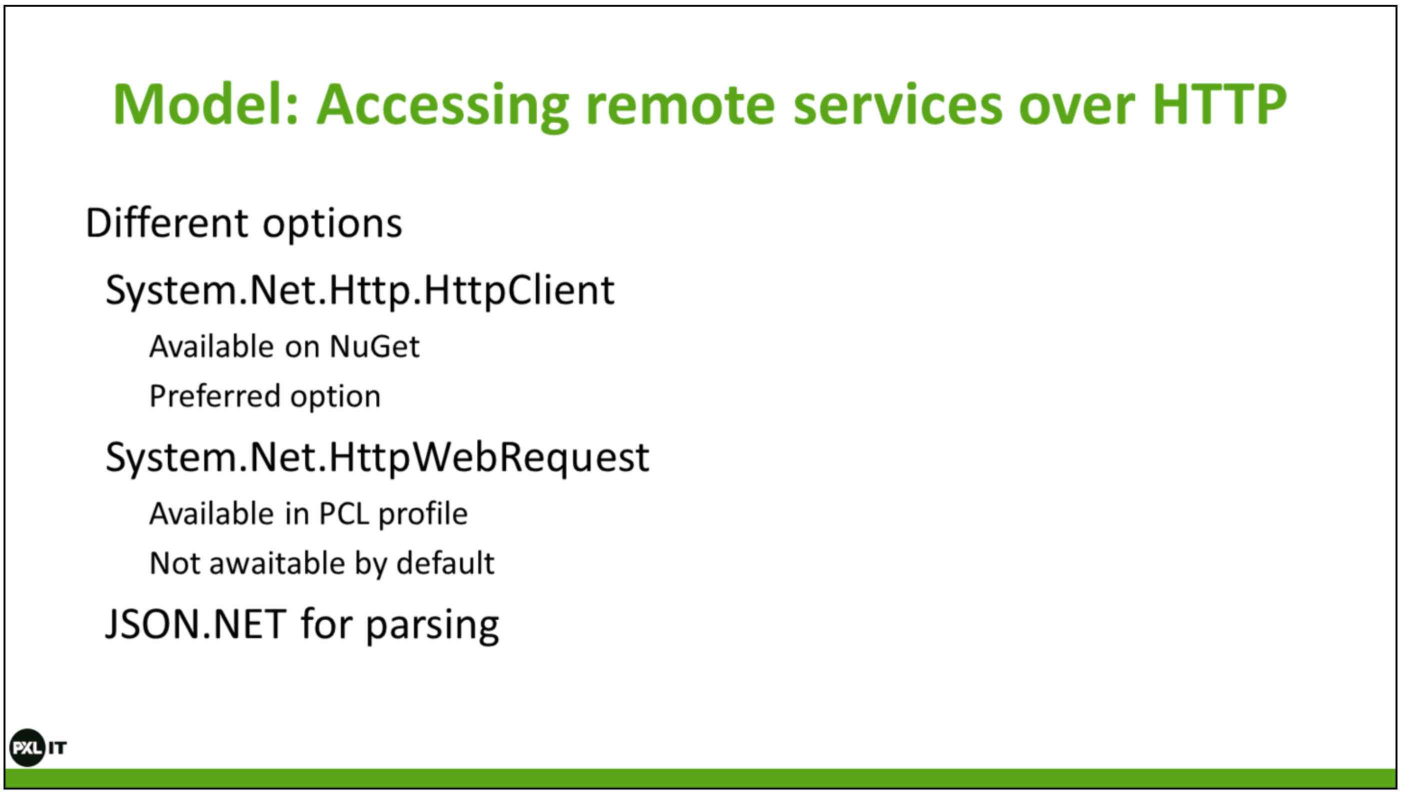
Shared code

Explain that with Xamarin Forms, all this can be shared.



The base layers in the shared block

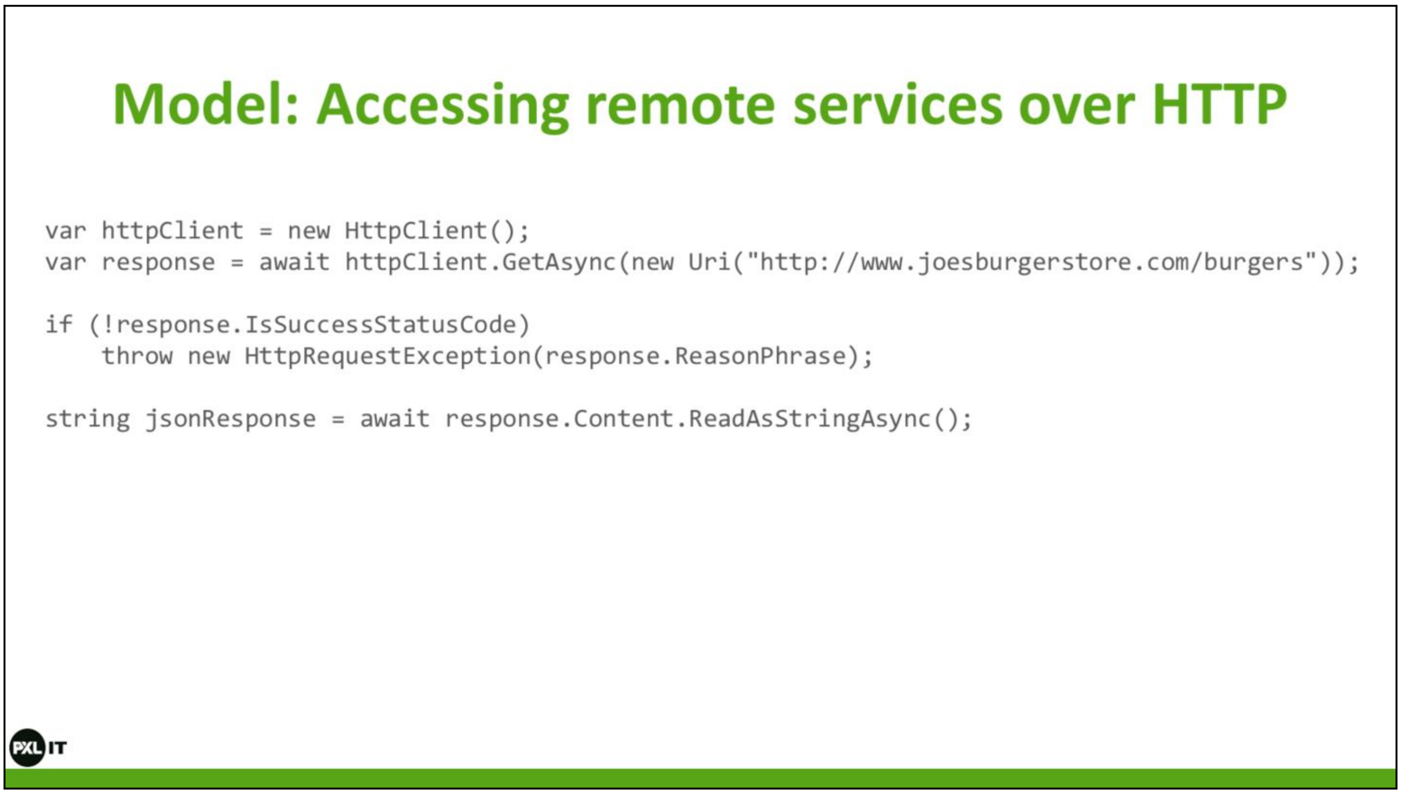
Next, we'll look at the exploration with the PCL exploration (JoesBurgerStore)



Model: Accessing remote services over HTTP

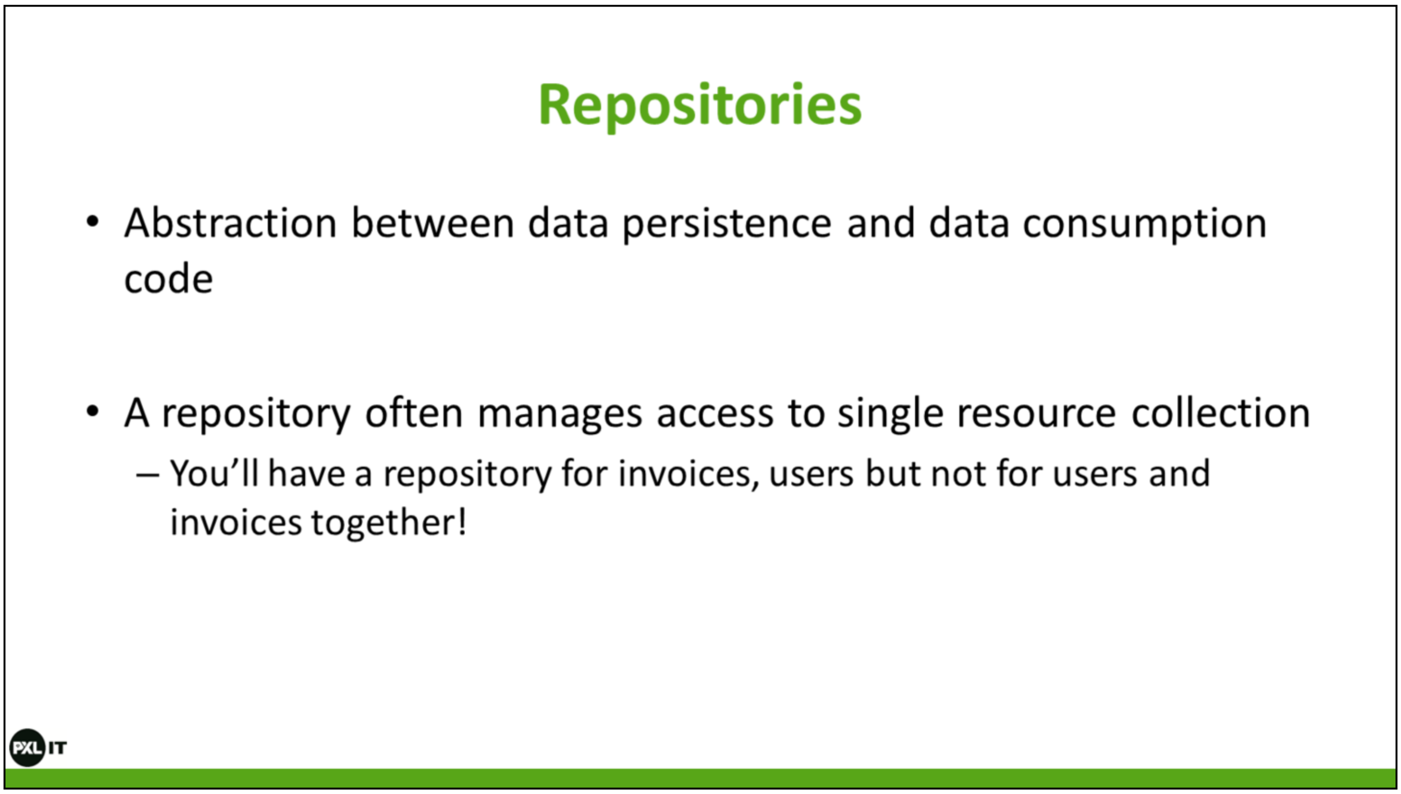
Different options

- System.Net.Http.HttpClient
  - Available on NuGet
  - Preferred option
- System.Net.HttpWebRequest
  - Available in PCL profile
  - Not awaitable by default
- JSON.NET for parsing



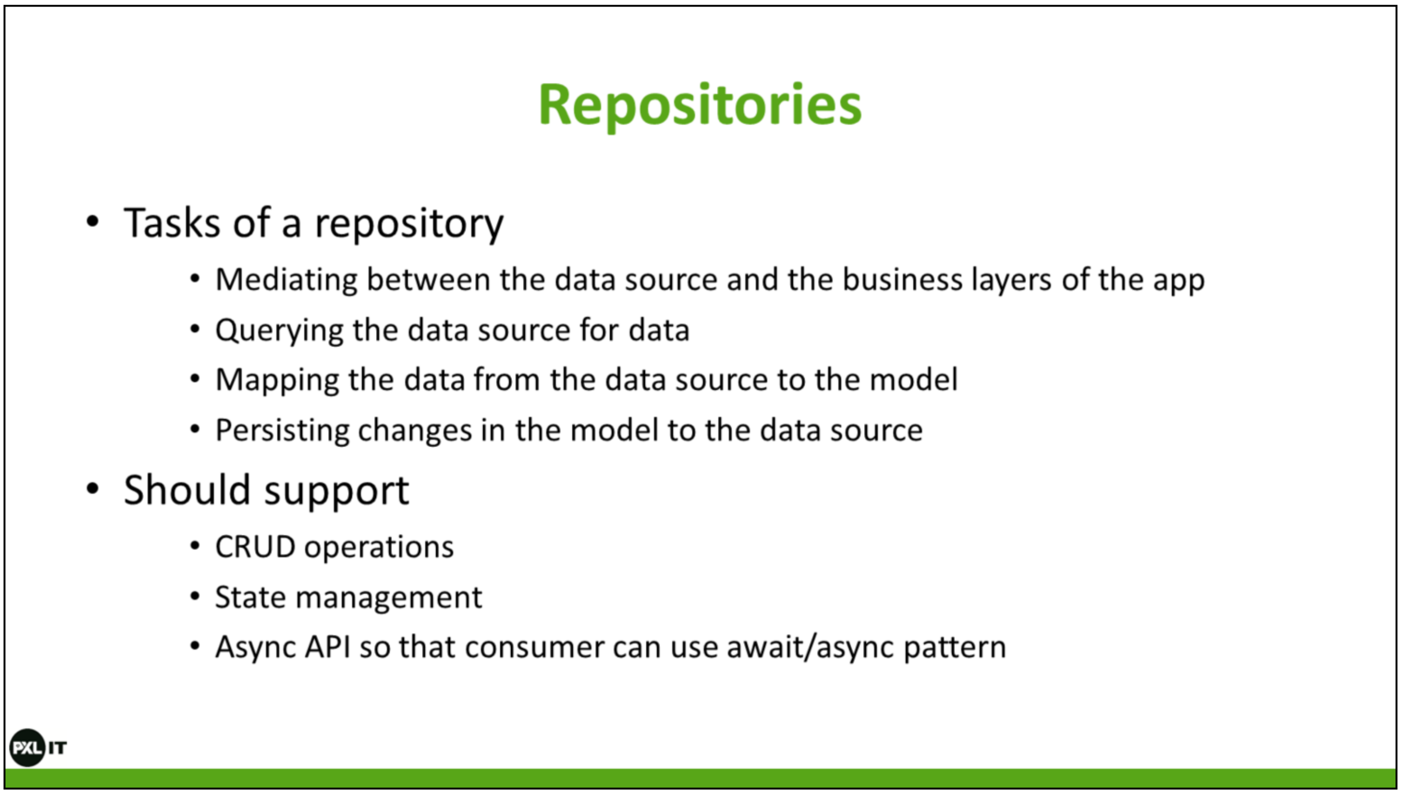
Model: Accessing remote services over HTTP

```
var httpClient = new HttpClient();
var response = await httpClient.GetAsync(new Uri("http://www.joesburgerstore.com/burgers"));

if (!response.IsSuccessStatusCode)
    throw new HttpRequestException(response.ReasonPhrase);

string jsonResponse = await response.Content.ReadAsStringAsync();
```



Repositories

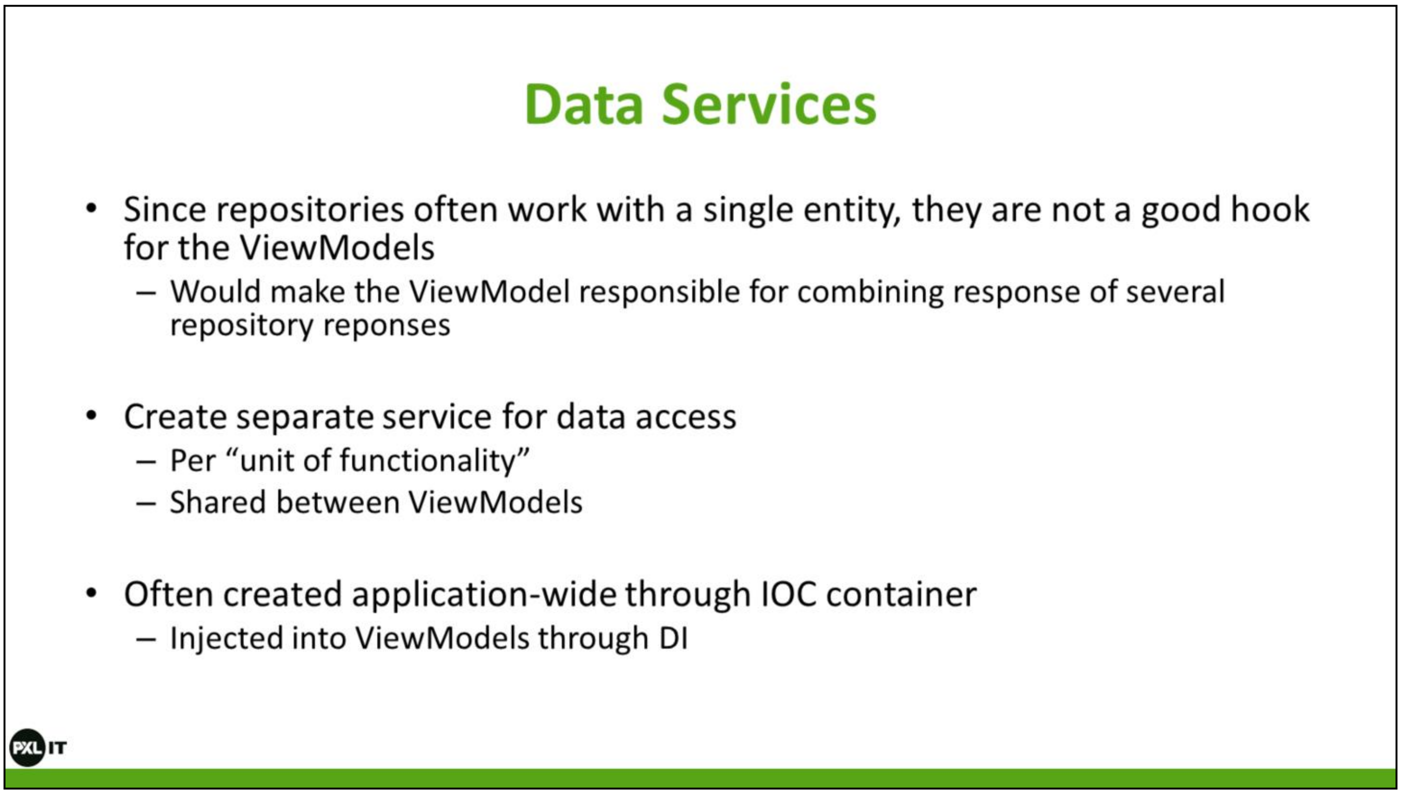
- Abstraction between data persistence and data consumption code
- A repository often manages access to single resource collection
  - You'll have a repository for invoices, users but not for users and invoices together!



Repositories

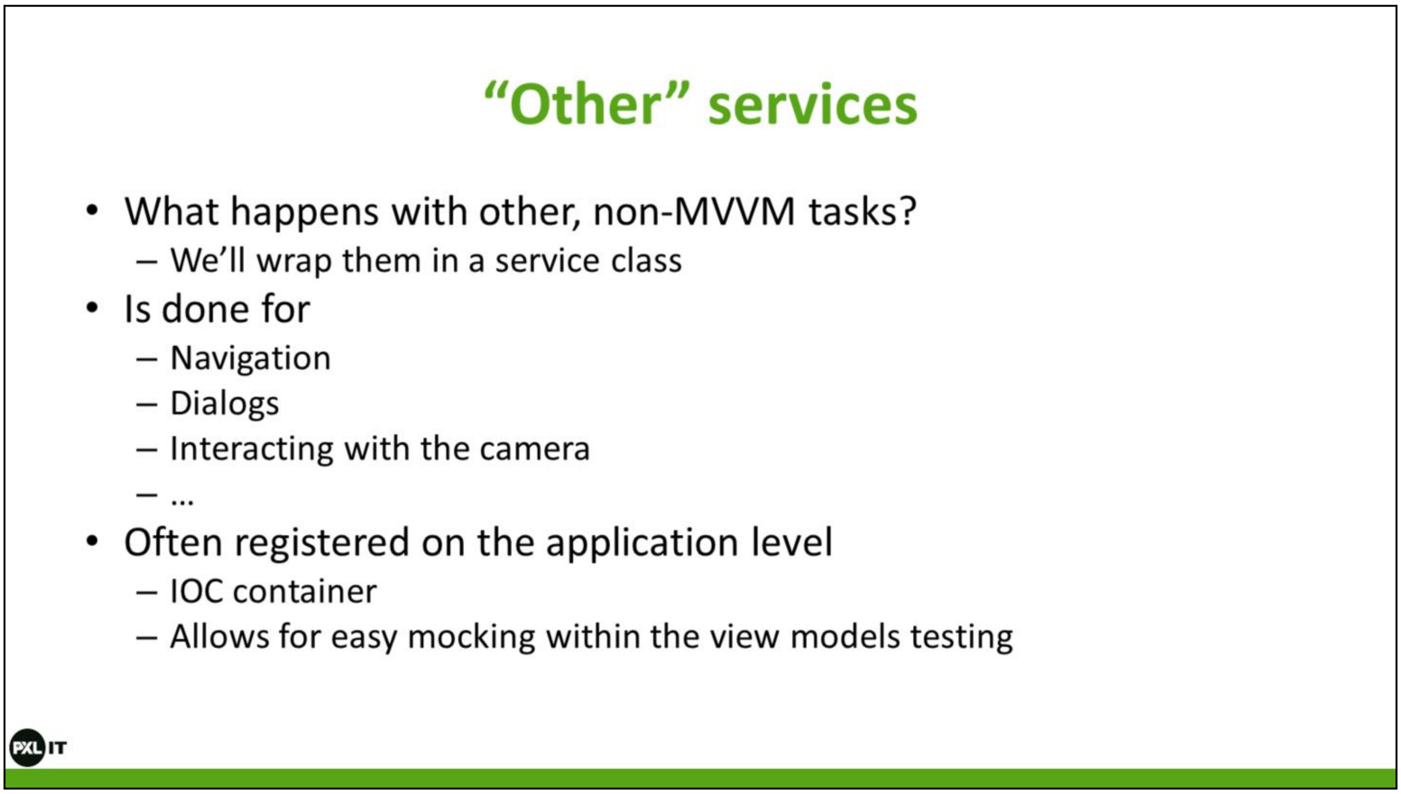
- Tasks of a repository
  - Mediating between the data source and the business layers of the app
  - Querying the data source for data
  - Mapping the data from the data source to the model
  - Persisting changes in the model to the data source
- Should support
  - CRUD operations
  - State management
  - Async API so that consumer can use await/async pattern

PXL IT



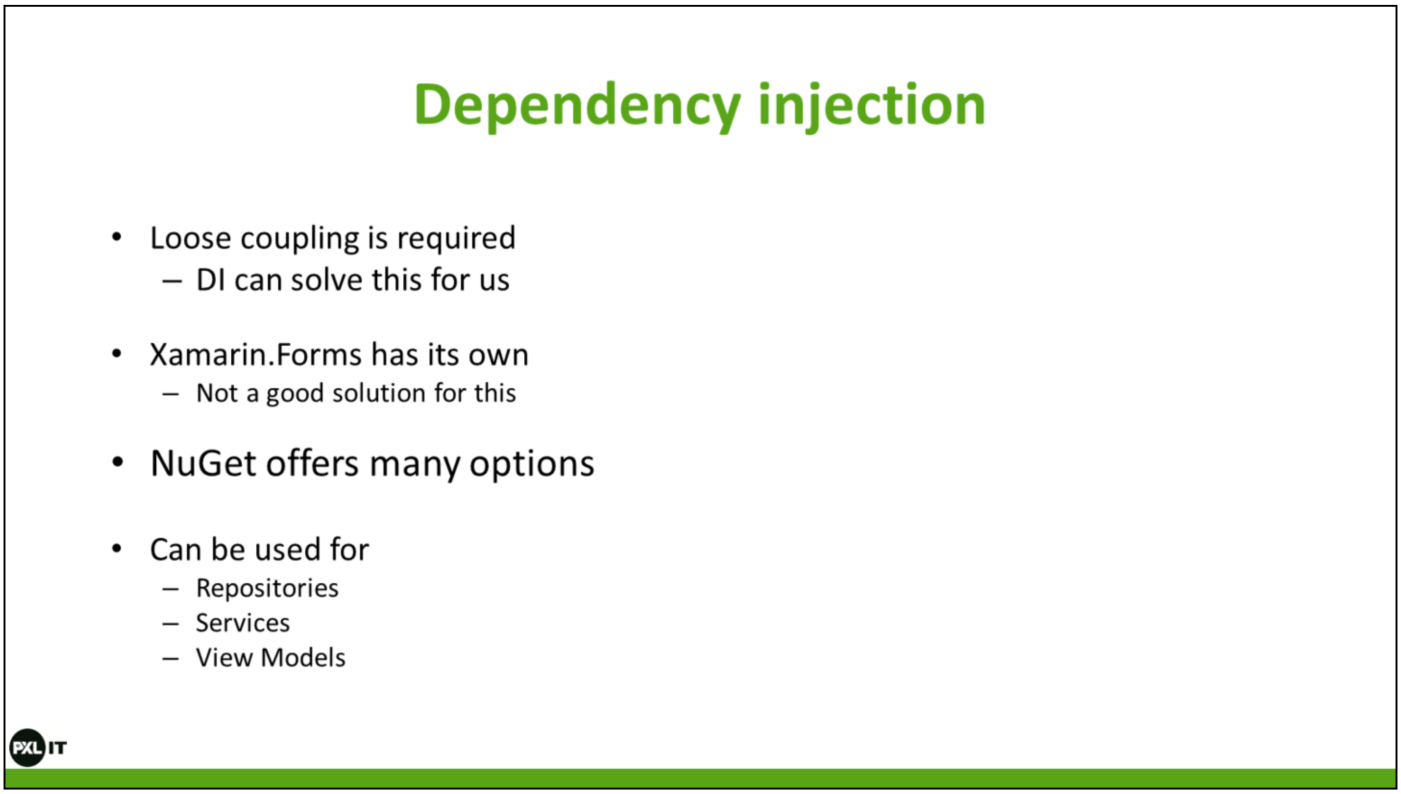
Data Services

- Since repositories often work with a single entity, they are not a good hook for the ViewModels
  - Would make the ViewModel responsible for combining response of several repository reponses
- Create separate service for data access
  - Per “unit of functionality”
  - Shared between ViewModels
- Often created application-wide through IOC container
  - Injected into ViewModels through DI



"Other" services

- What happens with other, non-MVVM tasks?
  - We'll wrap them in a service class
- Is done for
  - Navigation
  - Dialogs
  - Interacting with the camera
  - …
- Often registered on the application level
  - IOC container
  - Allows for easy mocking within the view models testing



Dependency injection

- Loose coupling is required
  - DI can solve this for us
- Xamarin.Forms has its own
  - Not a good solution for this
- NuGet offers many options
- Can be used for
  - Repositories
  - Services
  - View Models

Looking at the shared code

DEMO