Data Binding and MVVM

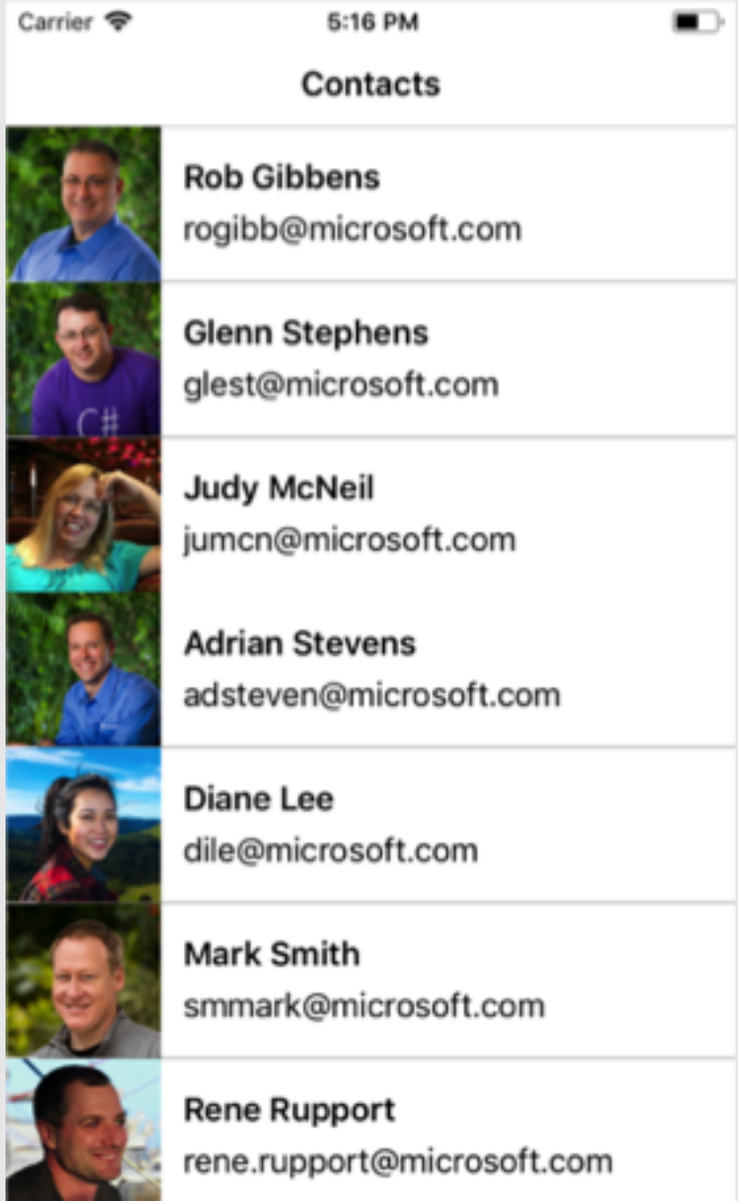
Apps are driven by data

Most applications display and manipulate data in some form

- internally generated
- read from an external source

Classes created to represent data are often referred to as <u>Models</u>

- can also refer to "entity" objects



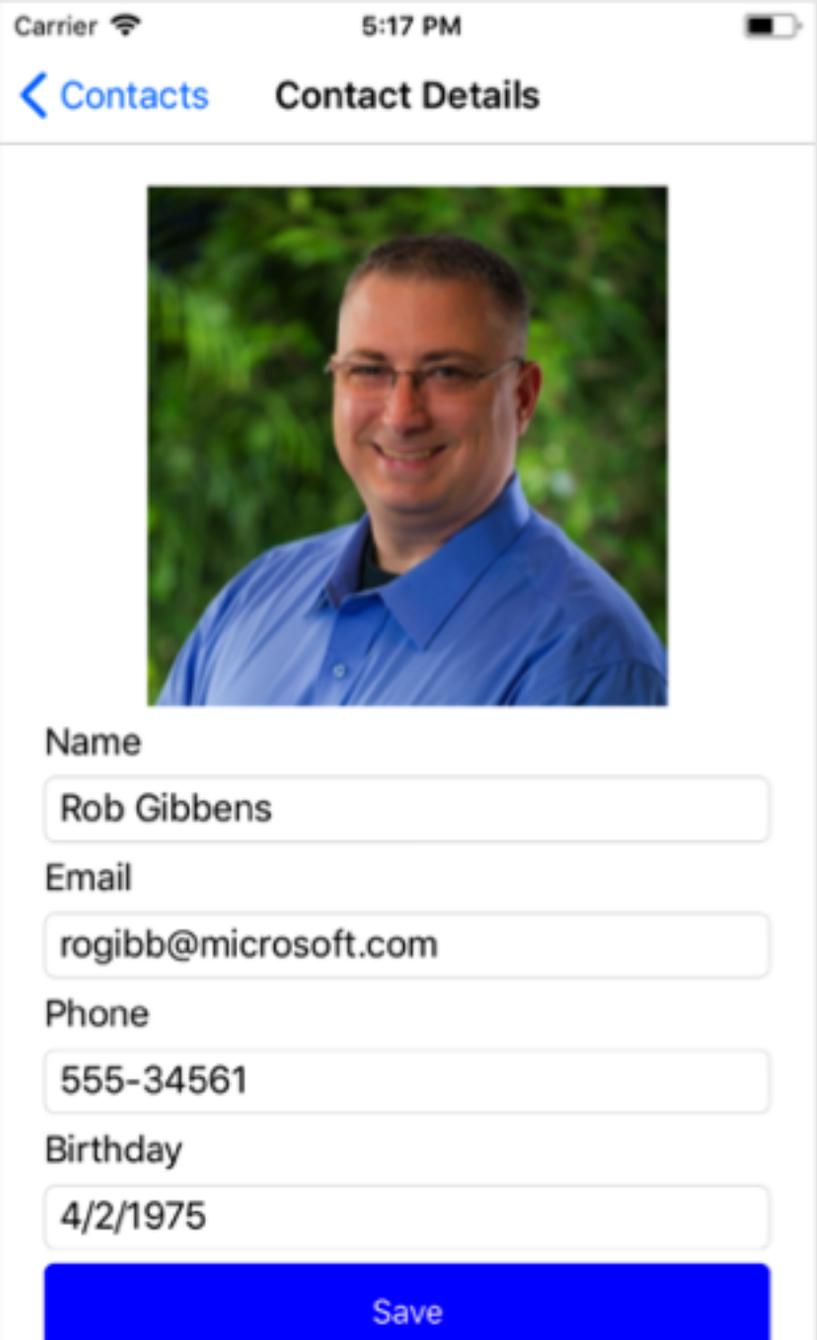
Data > Views

We use code to display internal data in our pages

```
headshot.Source = ...;
nameEntry.Text = person.Name;
emailEntry.Text = person.Email;
birthday.Date = person.Dob;
...
```

... and events to provide interactivity / behavior

```
nameEntry.TextChanged += (sender, e) =>
        person.Name = nameEntry.Text;
emailEntry.TextChanged += (sender, e) =>
        person.Email = emailEntry.Text;
```



Data > Views in code

This approach works, and for small-ish applications is perfectly adequate but it has disadvantages as the application grows in complexity

Updates to data are not centralized

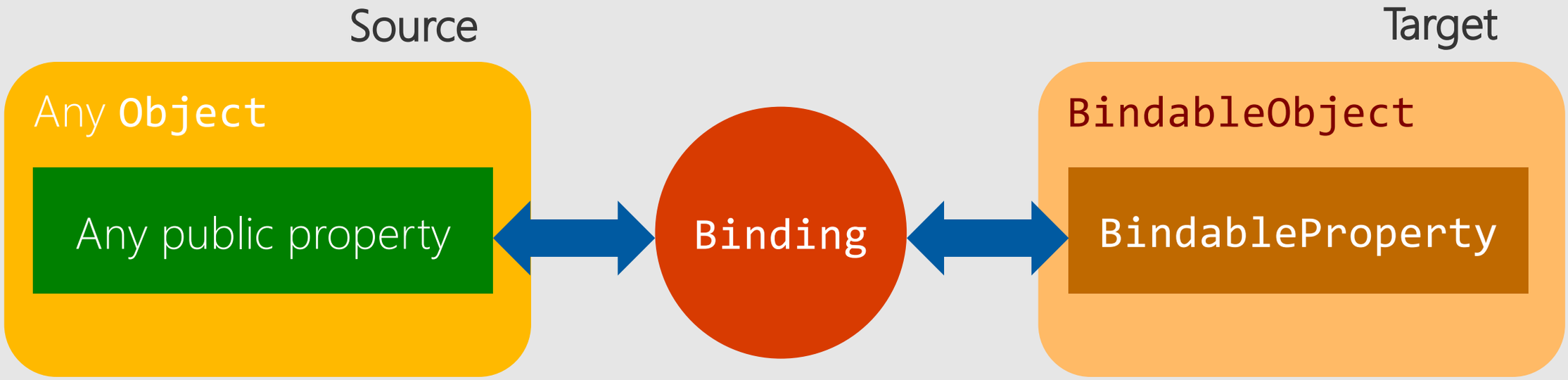
Relationships in data or UI behavior is harder to manage

Hard to unit test

UI is tightly coupled to the code behind logic, changes ripple through code

Introducing: Data Binding

Data Binding involves creating a loose **relationship** between a **source property** and a **target property** so that the source and target are **unaware of each other**



Binding acts as an *intermediary* – moving the data between the source and target

Creating Bindings in Xamarin.Forms

Bindings require **three pieces of information**

Source

Can be any accessible object

Path

Public property defined on the source object

Target

Must be a `BindableProperty`

Creating bindings [Source]

```
Person person = new Person() { Name = "Rob Gibbens", ... };

Entry nameEntry = new Entry();

Binding nameBinding = new Binding();
nameBinding.Source = person;

...
```

1

Binding identifies the source of the binding data – this is where the data comes from, in this case it's a single person defined in our application

Creating bindings [Path]

```
Person person = new Person() { Name = "Rob Gibbens", ... };

Entry nameEntry = new Entry();

Binding nameBinding = new Binding();
nameBinding.Source = person;
2 nameBinding.Path = "Name";

...
```

Binding identifies the *property path* which identifies a property on the source to get the data from, in this case we want to get the value from the `Person.Name` property

Creating bindings [Target]

```
Person person = new Person() { Name = "Rob Gibbens", ... };

Entry nameEntry = new Entry();

Binding nameBinding = new Binding();
nameBinding.Source = person;
nameBinding.Path = "Name";

3 nameEntry.SetBinding(Entry.TextProperty, nameBinding);
```

Binding is associated to the target property using the **`BindableObject.SetBinding`** method

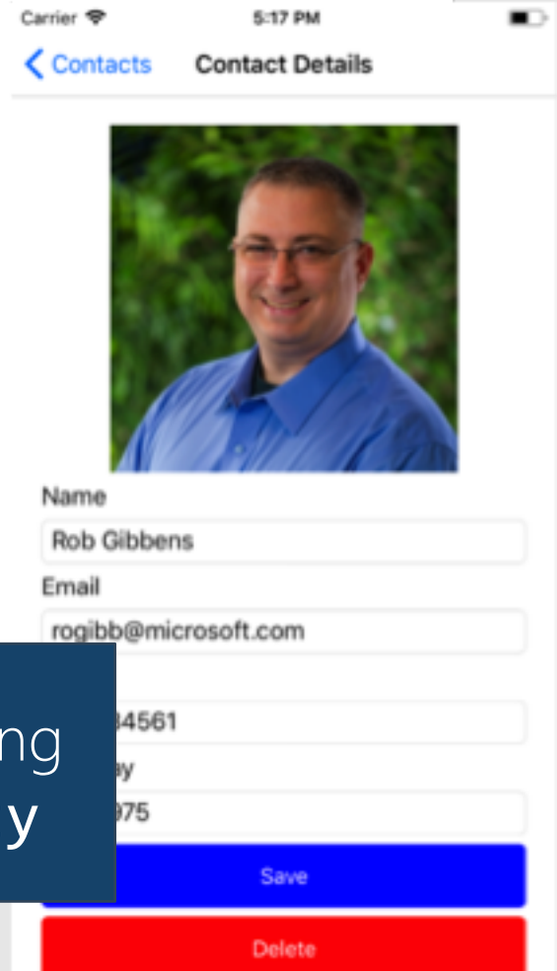
Creating bindings [Target]

```
Person person = new Person() { Name = "Rob Gibbens", ... };

Entry nameEntry = new Entry();

Binding nameBinding = new Binding();
nameBinding.Source = person;
nameBinding.Path = "Name";

3 nameEntry.SetBinding(Entry.TextProperty, nameBinding);
```



This is passed the specific target property the binding will work with – this must be a **`BindableProperty`**

Creating bindings [Target]

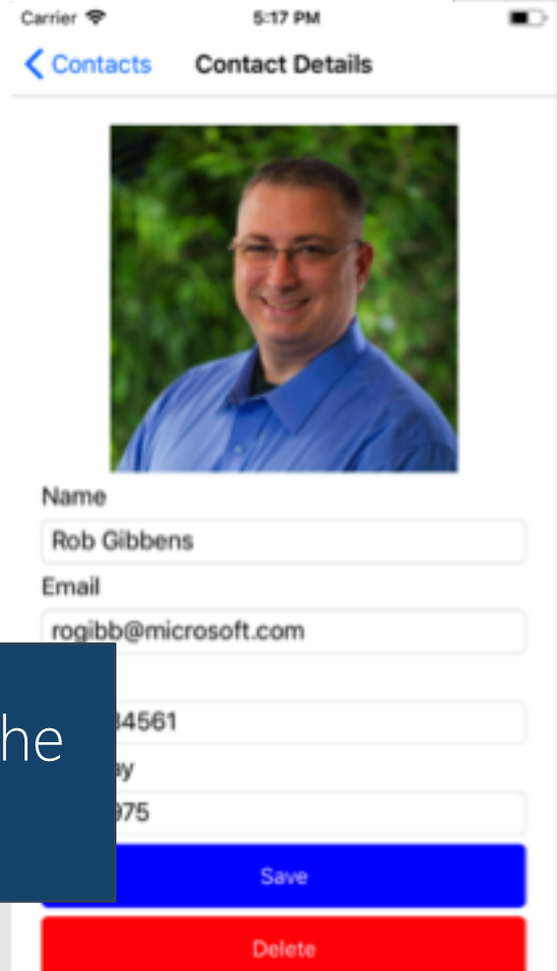
```
Person person = new Person() { Name = "Rob Gibbens", ... };

Entry nameEntry = new Entry();

Binding nameBinding = new Binding();
nameBinding.Source = person;
nameBinding.Path = "Name";

nameEntry.SetBinding(Entry.TextProperty, nameBinding);
```

3



… and the binding which identifies the source and the property on the source to apply

Demonstration

Simple data binding

Creating bindings [XAML]

Create bindings in XAML with **{Binding}** markup extension

Must set the **Source** and **Path**

```
<Entry Text="{Binding Source=????, Path=Name}" />
```

Assigned to **Target** property

Creating bindings [XAML]

Create bindings in XAML with **{Binding}** markup extension

Shortcut to setting the **Path** property

```
<Entry Text="{Binding Name, Source=????}" />
```

Data binding source

Pages often display properties from a small number of data objects

Can set the binding source on each binding separately, or use the **`BindingContext`** as the *default* binding source

```
public class Person
{
   public string Name { get; set; }
   public string Email { get; set; }
   public string Phone { get; set; }
}
```

Multiple Bindings

`BindingContext` supplies the source for any binding associated with a view when the **`Binding.Source`** property is not set

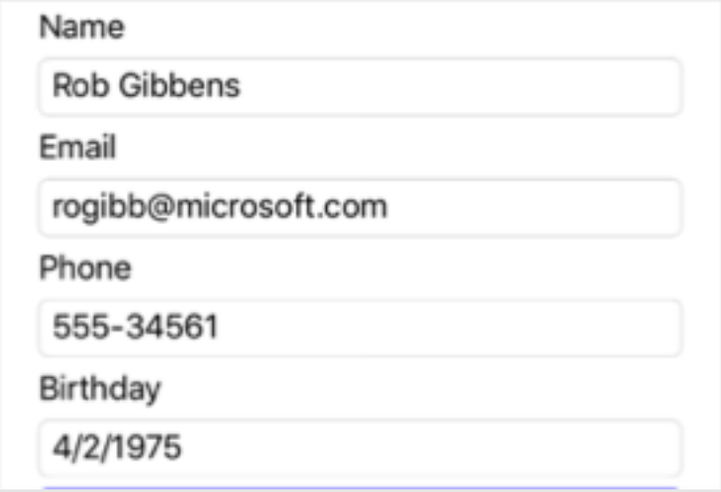
```
Person person = ...;
...
nameEntry.BindingContext = person;
nameEntry.SetBinding<Person>(Entry.TextProperty,
                             data => data.Name, BindingMode.TwoWay);
```

Useful to use a generic form of **`SetBinding`** to create bindings with typed properties when establishing bindings in code, notice we are *not* setting a source property on the binding – instead, it will use **`BindingContext`**

BindingContext inheritance

`BindingContext` is automatically *inherited* from parent to child – can set it once on the root view and it will be used for all children

```
public partial class PersonDetailsPage : ContentPage
{
    public PersonDetailsPage (Person person)
    {
      BindingContext = person;
      InitializeComponent ();
    }
}
```



BindingContext inheritance

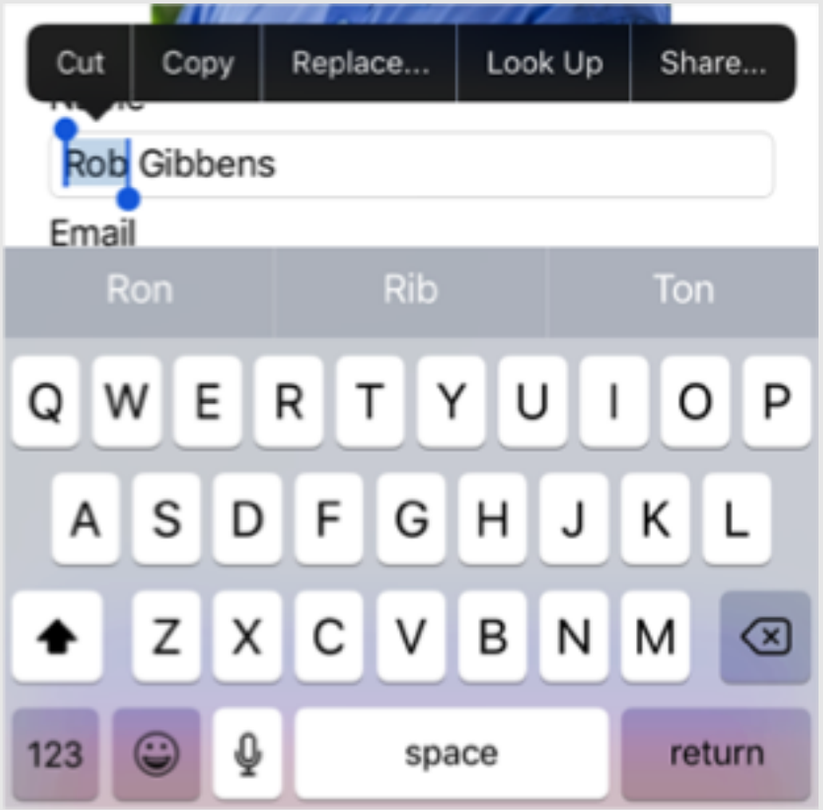
`BindingContext` is automatically *inherited* from parent to child – can set it once on the root view and it will be used for all children

```
BindingContext = person;
```

```
<StackLayout Padding="20" Spacing="20">
   <Entry Text="{Binding Name}" />
   <Entry Text="{Binding Email}" />
   <Label Text="{Binding Gender}" />
   <Label Text="{Binding Dob}" />
</StackLayout>
```

By setting the binding context to the **`Person`**, no explicit source is necessary in XAML

Creating two-way bindings



Often want data transfer to be bi-directional

- source > target (always happens)
- <u>target > source</u> (optional)

```
nameEntry.TextChanged += (sender, e)
      => person.Name = nameEntry.Text;
```

Binding Mode

Binding **Mode** controls the direction of the data transfer, can set to "TwoWay" to enable bi-directional bindings

Source Property must have **public setter**

```
name.SetBinding(Entry.TextProperty,
  new Binding("Name") {
      Mode = BindingMode.TwoWay
});
```

```
<Entry
  Text="{Binding Name, Mode=TwoWay}" />
```

Manually controlled through the **`Binding.Mode`** property

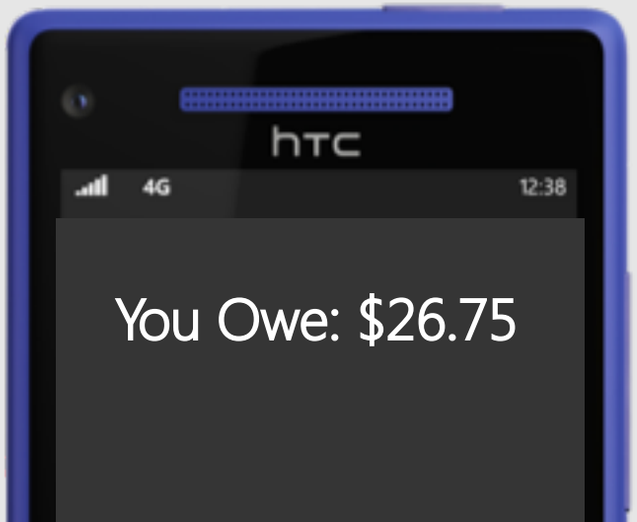
Simple Textual Conversions

Binding can do simple, text formatting when going from Source > Target

```
public double BillAmount { get; set; }
```

```
<Label Text="{Binding BillAmount,
                      StringFormat='You Owe: {0:C}'}"/>
```

Binding calls a **`String.Format`** passing the specified format string and the source value before assigning it to the target

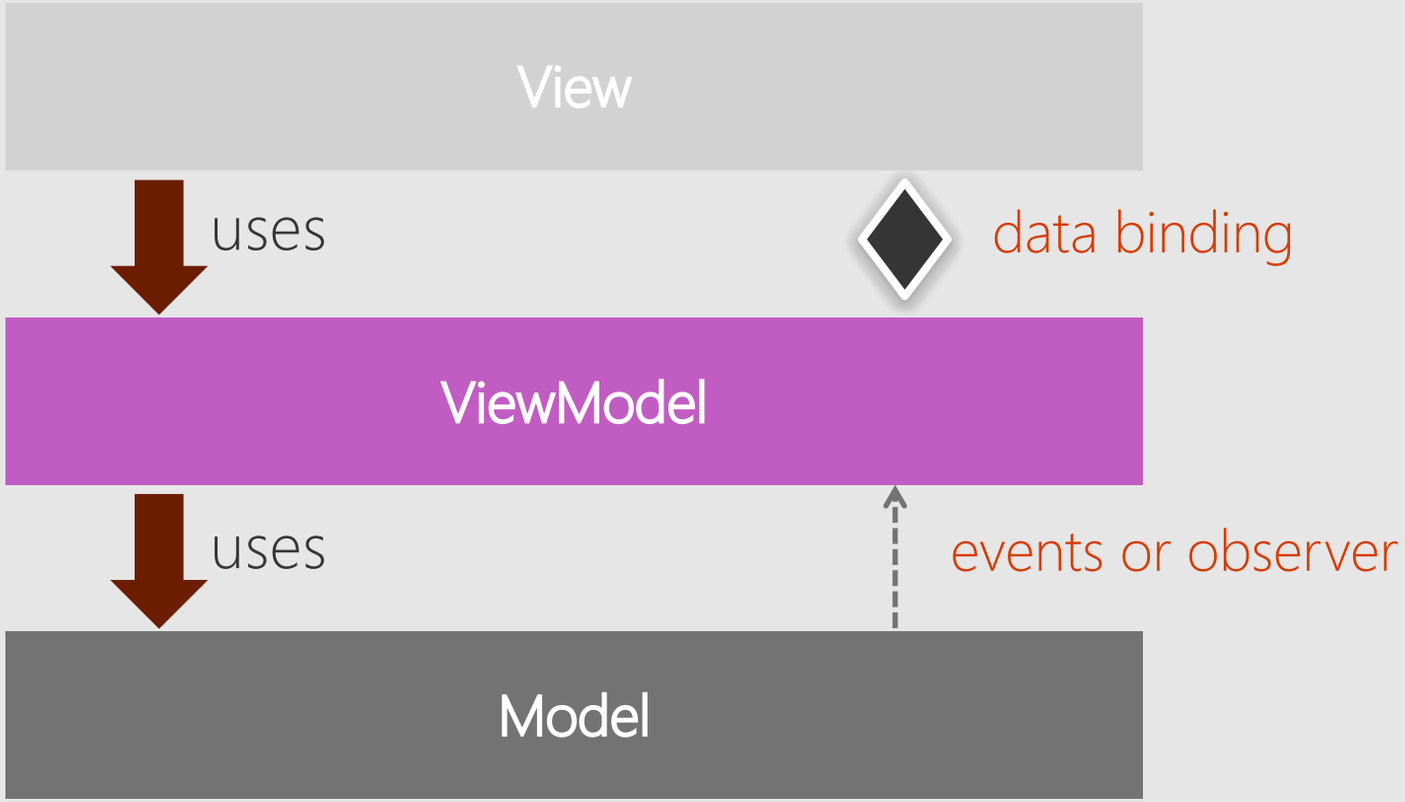


Exercise #13

Use data binding to display a question

Model-View-ViewModel (MVVM)

MVVM is a layered, **separated presentation pattern** made popular by XAML based UI where a data binding engine takes the place of the controller / presenter



What is the Model?

Models **manage the application data** and may include any combination of domain logic, persisted state and validation, not necessarily in one object

Models are intended to be **shared across platforms** and should not depend on platform-specific features

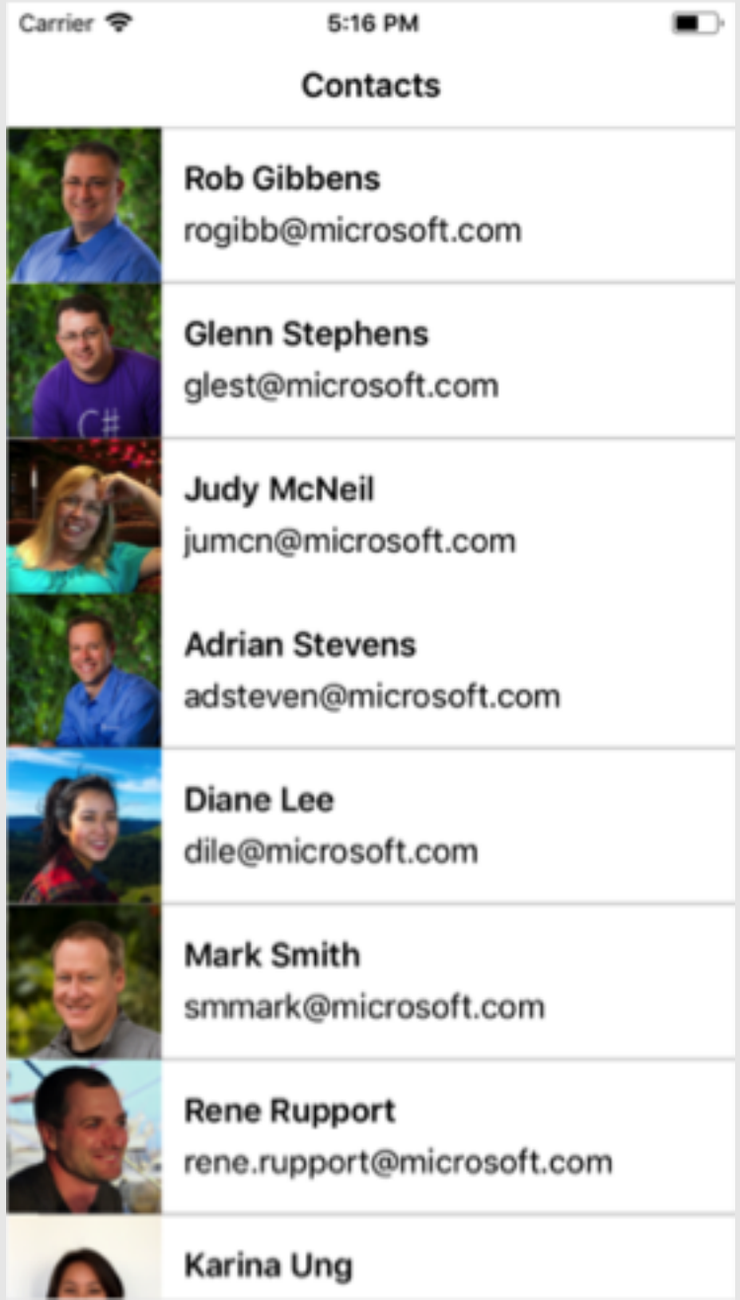
```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime Dob { get; set; }
    public string Email { get; set; }
    public static Person GetById(int id);
}
```

What is the View?

View presents the information to the user in a platform-specific fashion

Should <u>not</u> contain code you want to unit test

Everything *visual* should be managed here – fonts, colors, etc.



What is the ViewModel?

The ViewModel provides a **view-centric representation** of the data to display

```
public class PersonViewModel
{
    private Person model;

    public string Name {
        get { return model.Name; }
        set { model.Name = value; }
    }

    public PersonViewModel(Person model) {
        model = model;
    }
    ...
}
```

Exposes properties from the data contained in the model

Often has 1:1 relationship with a specific model (but this is not a strict rule)

Why use a ViewModel?

Having a wrapper around our models allows us to customize the data specifically for the view – we have an *interception* point to provide presentation logic outside the data layer

```
public class PersonViewModel
{
    public string Birthdate {
        get {
            return model.Dob
                .ToString("D");
                }
    }
}
```

As a simple example, we can format dates for locale and coerce the **`DateTime`** into a presentable **`string`**

Saturday, January 22 1999

Why use a ViewModel?

Apps often require *presentation logic* to manage runtime state, handle user interaction and validate input; we can centralize this logic into our ViewModels and make it testable and reusable

Which data item is selected/visible?

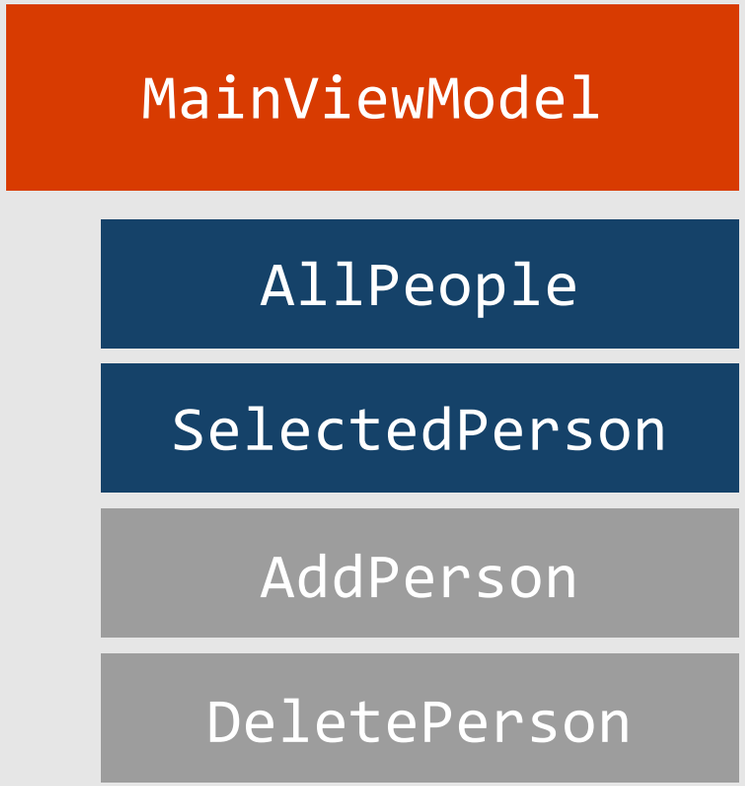
Have we filled in all the require values?

Is that background operation complete?

ViewModel relationships

Apps can have multiple view models – one for each unique "data-bindable" entity being displayed

Useful to implement a "main" or primary view model that can be set as the binding context and provides access to UI state and behavior



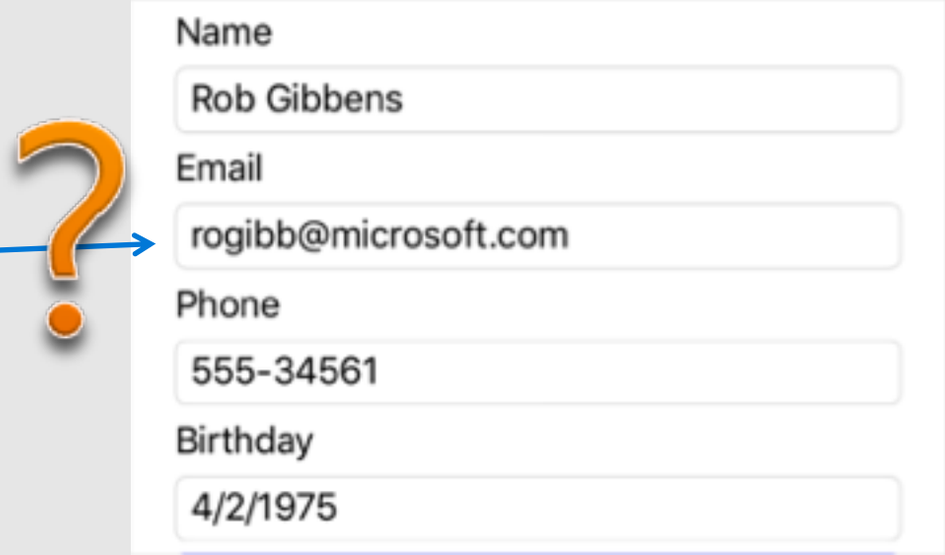
Exercise #14

Create and connect view models to drive the UI

Pushing changes to the UI

ViewModels are regular C# objects – Xamarin.Forms needs to know when properties are changed so that it can refresh the UI

```
person.Email = "rogibb@microsoft.com";
```



INotifyPropertyChanged

`INotifyPropertyChanged` provides change notification contract, should be implemented by your ViewModel objects

```
namespace System.ComponentModel
{
    public interface INotifyPropertyChanged
    {
        event PropertyChangedEventHandler PropertyChanged;
    }
}
```

Implementing INotifyPropertyChanged

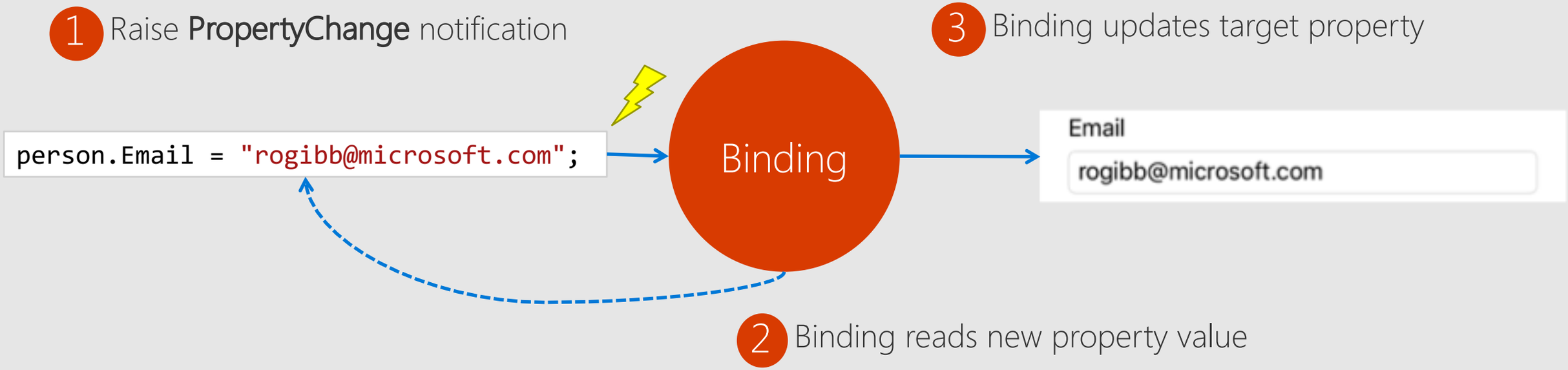
```
public class PersonViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged = delegate {};

    public string Email {
        get { return model.Email; }
        set {
            if (model.Email != value) {
               model.Email = value;
               PropertyChanged(this,
                  new PropertyChangedEventArgs("Email");
            }
        }
    }
}
```

Must raise the PropertyChanged event when any data bound property is changed – otherwise the UI <u>will not update</u>

INPC + Bindings

Binding will subscribe to the **`PropertyChanged`** event and update the target property when it sees the source property notification



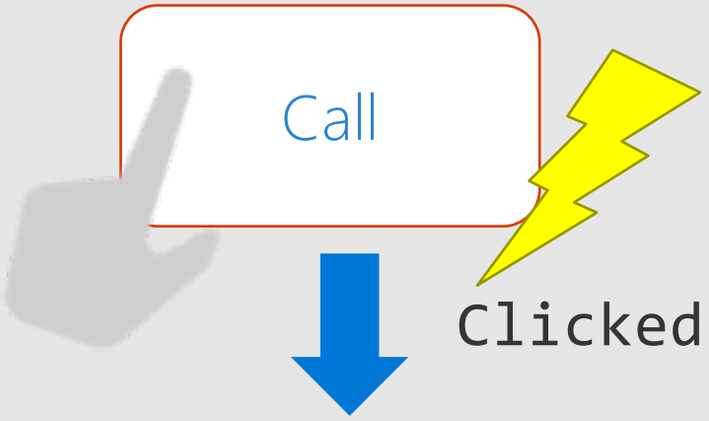
Exercise #15

Implement INotifyPropertyChanged

Event Handling

- UI raises events to notify code about user activity
  - **`Clicked`**
  - **`ItemSelected`**
  - …

- These events **must be handled** in the code behind file, can forward to a VM for centralized processing (and testing)



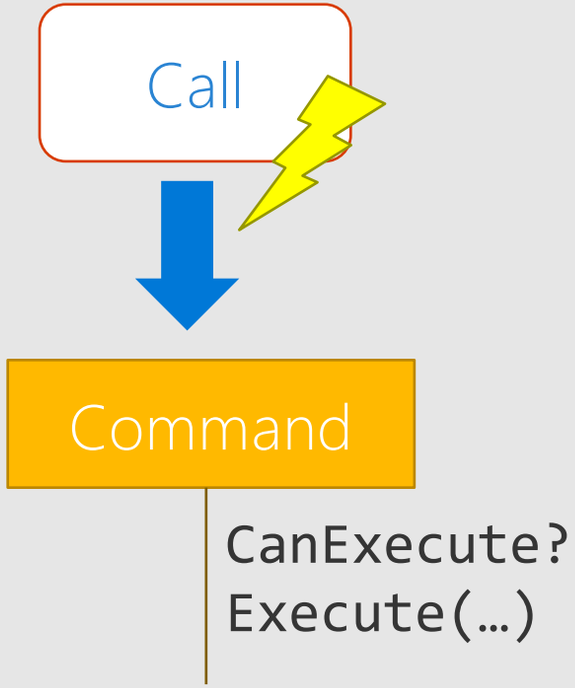

```
public MainPage()
{
    ...
    Button callButton = ...;
    callButton.Clicked += OnCall;
}

void OnCall(object sender, EventArgs e)
{
    ...
}
```

Commands

Microsoft defined the **`ICommand`** interface to provide a commanding abstraction for their XAML frameworks

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



Binding to commands in Xamarin.Forms

Several Xamarin.Forms controls expose a **`Command`** property for the main action of a control which can be assigned to an **`ICommand`** implementation

`Button`

`Menu`

`ToolbarItem`

`TextCell`

Commands in Xamarin.Forms

Can use data binding to push commanding logic into a ViewModel

```
public ICommand CallMe { get; }
```

```
<Button Text="Call"
        Command="{Binding CallMe}" />
```

Implementing commands in the VM

Command should be exposed as a public property from the ViewModel

```
public class PersonViewModel : INotifyPropertyChanged
{
    public ICommand CallMe { get; private set; }
    ...

}
```

Using the Command class

Xamarin.Forms provides a **`Command`** implementation which calls delegate methods in response to **`Execute`** and **`CanExecute`**

```
public class PersonViewModel : INotifyPropertyChanged
{
    public Command CallMe { get; private set; }
    public PersonViewModel(...) {
        CallMe = new Command(OnCallMe, OnCanCallMe);
    }

    void OnCallMe() { ... }
    bool OnCanCallMe() { return ... }
}
```

Commanding status

Must use the **`ChangeCanExecute`** method to notify a binding that a command's current state has changed at runtime

```
public class PersonViewModel : INotifyPropertyChanged
{
    public string PhoneNumber { ...
        set { ...
            model.PhoneNumber = value;
            CallMe.ChangeCanExecute();
        }
    bool OnCanCallMe() {
        return !string.IsNullOrEmpty(PhoneNumber);
    }
}
```

In response, any binding tied to **`CallMe`** will use the **`CanExecute`** method to check whether the command can be called

Exercise #16

Use Commands to respond to button taps

Summary

- Data Binding framework in Xamarin.Forms allows the UI to be decoupled from the data that populates it
- Centered around the Binding class which is ties a source object property to a target (UI) element property
- MVVM is a popular pattern used with data binding to further insulate the view from the data by introducing an intermediary View Model
- Commanding allows ViewModels to handle UI interactions directly