Building a mobile enterprise application with Xamarin.Forms, Docker, MVVM and .NET Core

Gill Cleeren
@gillcleeren – www.snowball.be
Gill Cleeren
Microsoft MVP & RD
Mobile Architect @ Ordina.BE

Founder of Techorama

@gillcleeren
www.snowball.be
gill@snowball.be
Xamarin, .NET Core, HTML5, WPF, social and Windows development

http://gicl.me/mypscourses
Agenda

• Overall application structure
• The Xamarin application architecture
  • MVVM
  • Dependency Injection
  • Loose-coupled Messaging
  • Navigation
  • Service communication
  • Testing
• Backend architecture
  • .NET Core-based Microservices
  • Docker
The Solution Structure
DEMO

Looking at the Solution Structure & the Finished Application
MVVM
MVVM

• Architectural pattern
• Based on data binding and commanding
• Popular for testable and maintainable XAML applications
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 MVVM

• Testable
• Developers and designers can work independently
• A new XAML view can be added on top of the view models without problems
• Changes can be made in view model without risking issues with the model
<Entry Text="{Binding UserName.Value, Mode=TwoWay}"></Entry>

<Label
    Text="{Binding UserName.Errors, Converter={StaticResource FirstValidationErrorConverter}}" />

<ListView
    IsVisible="{Binding Campaigns.Count, Converter={StaticResource CountToBoolConverter}}"
    ItemsSource="{Binding Campaigns}"
Sample View Model Code

```csharp
public class LoginViewModel : INotifyPropertyChanged
{
    public string UserName
    {
        get; set;
    }

    public string Password
    {
        get; set;
    }

    public ICommand SignInCommand
    {
        get
        {
            return new Command(async () => await SignInAsync());
        }
    }
}
```

We will use a different base class in the real application!
DEMO

The MVVM Pattern Applied
Responding to changes in the (view) model

• Handled through INotifyPropertyChanged interface
• PropertyChanged raised for changes of view model or model property value changes
  • (View)Model property changes
  • Calculated properties
  • Raise event at the end of a method that makes changes to the property value
  • Don’t raise PropertyChanged when the value didn’t change
  • Don’t raise PropertyChanged from the constructor
  • Don’t raise PropertyChanged in a loop
BindableObject

public abstract class ExtendedBindableObject : BindableObject
{
    public void RaisePropertyChanged<T>(Expression<Func<T>> property)
    {
        var name = GetMemberInfo(property).Name;
        OnPropertyChanged(name);
    }

    private MemberInfo GetMemberInfo(Expression expression)
    {
        ...
    }
}
DEMO

Looking at the ViewModels
Commanding

- Action is defined in one place and can be called from multiple places in the UI
- Available through ICommand interface
  - Defines Execute() and CanExecute()
- Can create our own or use built-in commands
The ICommand Interface

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

- Command property available only on ButtonBase-derived controls
- Other controls and interactions only possible through “behaviours”
  - Use of an attached behaviour
  - Use of a Xamarin.Forms behaviour
public class EventToCommandBehavior : BindableBehavior<View>
{
    protected override void OnAttachedTo(View visualElement)
    {
        ...
    }
}
Using EventToCommandBehavior

```xml
<ListView ItemsSource="{Binding Orders}"
        Behaviors>
    <behaviors:EventToCommandBehavior
        EventName="ItemTapped"
        Command="{Binding OrderDetailCommand}"
        EventArgsConverter="{StaticResource ItemTappedEventArgsConverter}" />
</ListView.Behaviors>
</ListView>
```
Using TapGestureRecognizer

```xml
<StackLayout>
    <Label Text="SETTINGS"/>
    <StackLayout.GestureRecognizers>
        <TapGestureRecognizer Command="{Binding SettingsCommand}"
                                NumberOfTapsRequired="1"/>
    </StackLayout.GestureRecognizers>
</StackLayout>
```
DEMO
Commanding Done Right
Who Knows Who?
Linking the View and the View Model

View-First

View Model-First
View-First (from XAML)

```xml
<ContentPage>
    <ContentPage.BindingContext>
        <local:LoginViewModel/>
    </ContentPage.BindingContext>
</ContentPage>
```
public LoginView() {
    InitializeComponent();
    BindingContext = new LoginViewModel();
}

View First (from code)
The View Model Locator

View Model Locator

View 1

View 2

View n

View Model 1

View Model 2

View Model n
DEMO
The ViewModel Locator
Dependency Injection
Dependency Injection

- Type of inversion of control (IoC)
- Another class is responsible for obtaining the required dependency
- Results in more loose coupling
  - Container handles instantiation as well as lifetime of objects
- Autofac is commonly used
  - Many others exist
Dependency Injection

ProfileViewModel

IContainer

IOderService

OrderService
DEMO

Working with Dependency Injection
Loose-coupled Messaging
View Model communication
Messaging Center built-in in Xamarin.Forms

- Publish messages
- Subscribe to messages
Messaging Center

• Implements pub-sub model for us already
  • Built-in in Xamarin.Forms
• Multicast supported
• Based on strings (not always perfect)
DEMO

Working with Messages and the Messaging Center
Navigation
Navigation and MVVM

- Navigation isn’t always easy to include in an MVVM scenario
  - No tight-coupling can be introduced
  - Who is responsible for navigation? View Model? View?
  - How can we pass parameters during navigation?

- Xamarin.Forms comes with INavigation interface
  - Will be wrapped as it’s too basic for real-life scenarios
Our Own NavigationService

• Must be registered in the Dependency Injection system

```csharp
public interface INavigationService
{
    ViewModelBase PreviousPageViewModel { get; }

    Task InitializeAsync();

    Task NavigateToAsync<TViewModel>() where TViewModel : ViewModelBase;

    Task NavigateToAsync<TViewModel>(object parameter) where TViewModel : ViewModelBase;

    Task RemoveLastFromBackStackAsync();

    Task RemoveBackStackAsync();
}
```
DEMO

Adding Navigation
Service communication
Take some REST

• REST: Representational State Transfer
• Based on open HTTP standards
  • Open for all types of applications
• Works with Resources
  • We’ll send requests to access these resources
  • URI and HTTP method are used for this
• Results in HTTP Status code
  • 200, 404... based on result of request
Communicating with a REST API

• Apps will typically use services for making the data request
  • Are responsible for communication with the actual API
  • Controllers on API microservices return DTOs
  • Are transferred to the application

• App can use HttpClient class
  • Works with JSON
  • Returns HttpResponseMessage after receiving a request
  • Can then be read and parsed
    • Json.NET
Loading data from the service

```csharp
public override async Task InitializeAsync(object navigationData)
{
    IsBusy = true;
    Products = await _productsService.GetCatalogAsync();
    Brands = await _productsService.GetCatalogBrandAsync();
    Types = await _productsService.GetCatalogTypeAsync();
    IsBusy = false;
}
```
DEMO

Accessing Remote Data
Backend architecture
ASP.NET Core
".NET Core is a general purpose development platform maintained by Microsoft and the .NET community on GitHub. It is cross-platform, supporting Windows, macOS and Linux, and can be used in device, cloud, and embedded/IoT scenarios."

source: https://docs.microsoft.com/en-us/dotnet/articles/core
ASP.NET Core

“ASP.NET Core is a new open-source and cross-platform framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backends.”

source: https://docs.microsoft.com/en-us/aspnet/core
ASP.NET Core

• Built on top of .NET Core
• Lightweight
• Cross-platform
  • Windows, Mac & Linux

→ Easy in combination with Docker and Microservices
Containerized Microservices
Monoliths

• Client-server often results in tiered applications
  • Specific technology used per tier
  • Known as monolithic applications
• Often have tight coupling between components in each tier
  • Components can’t be scaled easily
  • Testing individual components might also be hard
Monoliths

• Not being able to scale can be an issue for cloud readiness
  • All layers typically are required
  • Scaling is cloning the entire application onto multiple machines
Monolithic applications
Enter microservices

- Microservices are easier for deployment and development
  - Better agility
  - Better combination with cloud
- App will be decomposed into several components
  - Components together deliver app functionality
- Microservice = small app, independent concern
  - Have contracts to communicate with other services
- Typical microservices
  - Shopping cart
  - Payment system
  - Inventory system
Enter microservices

• Can scale out independently
  • If an area requires more processing power, can be scaled out separately
  • Other parts can remain the same

• Scale-out can be instantaneous
  • Web front-end for handling more incoming traffic
Enter microservices

• Microservices manage their own data
  • Locally on the server on which they run
  • Avoid network overhead

• Faster for processing

• Even eliminate need for caching

• Support for independent updates
  • Faster evolution
  • Rolling updates, onto subset of instances of single service support rollback
Benefits of using microservices

• Small
• Evolve easily
• Scale-out independently
• Isolate issues to the faulty microservice
• Can use latest and greatest
  • Not constrained to using older technologies
Disadvantages of microservices

- Partitioning a real application is hard
- Complex
  - Intercommunication between services
- Eventual consistency
  - Atomic transactions often not supported
- Deployment (initial) might be harder
- Direct client-to-microservice communication might not be a good idea
DEM0

Looking at the Microservices
“Containerization is an approach to software development in which an application and its versioned set of dependencies, plus its environment configuration abstracted as deployment manifest files, are packaged together as a container image, tested as a unit, and deployed to a host operating system”
Adding containers to the mix

• Container is isolated, resource controlled and portable operating environment
• Applications in containers run without touching resources of host or other containers
• Acts like a VM or physical machine
• Work great in combination with microservices
• Docker is most commonly used approach here
What’s a container really?

- Container runs an operating system
- Contains file system
- Can be accessed over the network like a real machine/VM
- Contain the application (and dependencies)
- In general, require less resources to run than regular VMs
- Allow for easy and fast scale-up by adding new containers
Adding containers to the mix
Using Docker to host the microservices

• Each container hosts a separate part of the application
  • Single area of functionality
• Each microservice has its own database
  • Allows for full decoupling
  • Consistency is eventual
  • Can be improved using application events
    • Service bus

- Microservice 1
  - Product catalog
- Microservice 2
  - Ordering
- Microservice 3
  - Identity
DEMO

Containerization
Summary

• MVVM is the go-to standard for building enterprise-level Xamarin.Forms apps
• .NET Core is a good choice for building microservices
• Docker helps with deployment of microservices
Slides available at www.snowball.be
Building a mobile enterprise application with Xamarin.Forms, Docker, MVVM and .NET Core

Gill Cleeren
@gillcleeren – www.snowball.be