Using MVVM Light with your Xamarin Apps

Paul Johnson

Apress®

Using MVVM Light with your Xamarin Apps

Paul Johnson

Apress®

Using MVVM Light with your Xamarin Apps

Paul Johnson
Merseyside, United Kingdom

ISBN-13 (pbk): 978-1-4842-2474-8 ISBN-13 (electronic): 978-1-4842-2475-5
https://doi.org/10.1007/978-1-4842-2475-5

Library of Congress Control Number: 2017962633

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Todd Green
Development Editor: Laura Berendson
Technical Reviewer: Abhishek Sur
Coordinating Editor: Jill Balzano
Copy Editor: April Rondeau
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please email rights@apress.com or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484224748. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Contents

About the Author

Paul Johnson is a 46-year-old mobile software developer. He has written code for many companies (including TfL, NHS, and Farm Apps) and most of the time has enjoyed it. He lives with his wife and daughter along with a variety of animals.

He loves to travel (Australia being a particular favorite destination) and has a long-term love of retro computing—in particular the old 8-bit home micros of the 1980s.

This is his fourth book (first with Apress), and he has plans for a fifth already.

Outside of developing software, he is an avid scuba diver and can frequently be found diving at local quarries. He loves reading books as well as listening to rock and metal.

According to another bio, he doesn't have a pet badger. He doesn't. He does have a lot of coffee stains on his desk.

About the Technical Reviewer

Abhishek Sur has been a Microsoft MVP since 2011. He is currently working as a Product Head with Insync Tech-Fin Solutions Ltd. He has profound theoretical insight and years of hands-on experience in different .NET products and languages. Over the years, he has helped developers throughout the world with his experience and knowledge. He owns a Microsoft User Group in Kolkata named Kolkata Geeks and regularly organizes events and seminars in various places to spread .NET awareness. A renowned public speaker, voracious reader, and technology buff, Abhishek's main interest lies in exploring the new realms of .NET technology and coming up with priceless write-ups on the unexplored domains of .NET. He is associated with Microsoft's Insider list on WPF and C# and stays in touch with Product Group teams. He holds a Master's degree in Computer Application along with various other certificates to his credit.

Abhishek is a freelance content producer, developer, and site administrator. His website www.abhisheksur.com, guides both budding and experienced developers in understanding the details of languages and latest technologies. He has a huge fan following on social networks. You can reach him at books@abhisheksur.com, get online updates from his Facebook account, or follow him on Twitter @abhi2434.

Acknowledgments

This book is dedicated to my long-suffering wife, Becki. Without her, this wouldn't have been written. It is her giving me the time to do this sort of thing that means I can write this sort of thing. Almost equally, I'd like to thank my kids, Richard and Ashleigh, for their good humor, coffee, and laughter.

I'd also like to say a big thank you to the Xamarin Developers group on Facebook for their constant pestering. Along with them, I am indebted to Steve Melia and Leigh Ciechanowski, former colleagues and now great friends whom I met at G2G3 Digital in Sheffield and who showed me the benefits of a technology that I was previously very skeptical about.

My final dedications go to the chaps over at Primordial Radio (an internet radio station dedicated to the rock and metal community: http://www.primordialradio.com) and, in particular, Russ Collington (better known as Dewsbury) for recording some of the samples used in the book. Good music always helps on those long nights of writing.

Finally, to Laurent Bugnion—the man who wrote the MVVM Light framework in the first place and who has helped in various parts of the book. A great man to know and someone with a limitless amount of patience when something doesn't actually sit correctly in my very muddled mind!

I'd like to acknowledge the denizens of Stack Overflow for their help when my code decided to go bang. Kulvir Gakhal at NELFT must also be mentioned, if for no other reason than the encouragement shown while I worked there.

During the writing of this book, I lost two very dear friends, Alan Paynter and Paul Haywood. They may have left the planet, but their memory remains. Rest in peace my friends.

Foreword

The Model-View-ViewModel pattern is the result of the work of a few people working at the same time on different problems and coming up with a similar solution. When Microsoft released its new application framework, Windows Presentation Foundation (WPF), it included a very flexible way to link the layers of an application, called data binding. Today, data binding is a pretty common way to keep source and target properties synchronized, but 12 years ago it was not that easy. We knew about the need to keep data neatly organized, but we were only starting to get the tools that allowed us to do so. The community was buzzing with new terms like *dependency injection*, *inversion of control*, *unit testing*, and *abstraction*. We were jumping through hoops to refactor our code and make it easier to read. Pages and pages of blogs were written about best practices that had just been invented. It was a vibrant time.

In 2006, I had the amazing chance to be sent by my employer at the time, Siemens, to Las Vegas, to witness the official release of WPF as well as the first application written with this new framework, Expression Blend. The atmosphere in the room was electric and excited. Everyone wanted to know more about this new desktop technology. The promise of a better workflow between designers and developers was something that many were longing for, after some frustrations in that regard with Windows Forms or even Visual Basic before that. Even on the web, designers were often annoyed with the limitations of CSS. Discovering XAML and Blend was like a breath of fresh air on a hot day.

As I came back home to Switzerland and started to explore, I realized the potential of this powerful technology and was especially impressed by Blend. Since we wanted to use WPF and XAML in Siemens' new desktop application, I started to explore its potential, especially how to enable designers to work on the user interface without disturbing the developers. This work (over the course of a few months!) led me to look for help, both at Microsoft and in the community. We ended up decompiling Blend, debugging how the views were displayed, and understanding how design-time data was working. We also came up with an architecture that Blend seemed to like very much. It's only later, after talking to Microsoft architect John Gossman, the father of MVVM, that I understood that what I had been looking at was the Model-View-ViewModel pattern, which had been there within WPF and XAML since the beginning.

Around the same time, a group of friends from the WPF community started chatting on a mailing list, exploring how WPF could be made even more powerful with helper classes and architecture. The WPF Disciples mailing list was very active and constantly came up with new ideas. At the same time, I was blogging a lot about WPF and wanted to create a framework allowing me to quickly build sample applications without having to repeat basic things over and over again. Also, I wanted to speed up the repetitive work of setting up an application with the proper architecture. This small collection of classes was published in 2009 under the moniker MVVM Light Toolkit. Like the name shows, I wanted this to be a toolkit and not a framework. In other words, it was a collection of tools that you could cherry-pick from, leaving what you didn't need and possibly replacing parts with other similar tools.

One of the early components was the so-called Messenger, which other WPF Disciples also implemented under different names, such as Event Bus or Mediator. I remember intense discussions around this component, which was characteristic of the interactions on the Disciples list. Sharing code with each other was a must, as was taking constructive criticism to improve our components. We also allowed each other to reuse and modify existing components, like, for instance, my good friend Josh Smith, who was extremely influential in those early days. His blog was a prime destination for MVVM content, and he was generous in donating code to other authors like myself. Many discussions with Josh and others led me to

think that there may be something more to MVVM Light than "just" some classes that could be used to build samples and blog articles. Maybe people wanted to use these components to build actual production applications. The thought was both exciting and scary!

After additional code reviews, I decided to go for it and to publish the toolkit in open source. I started promoting it in the community. The response was amazing! People started downloading it, and I was getting constructive feedback and suggestions. From the start, I have been blessed with a very positive community of users. This contributed to a better toolkit with new features and a lot of volunteer testers who gave amazing feedback. The interaction with this community is really what makes it all worth it. The question I get the most often at conferences is, "How many hours did you invest in MVVM Light?" To be honest, I really don't know the answer, and I probably wouldn't like to know it. This is a labor of love, and sometimes a little pain (mostly when I really want to implement something new or fix some bugs and don't have the time to do it). But it's always worth it when someone approaches me and tells me what they implemented using the toolkit, that it helped them in their task and allowed them to be more efficient. I love this community.

The download numbers were doing quite well before, but they really started taking off after the port to Xamarin. If you think about it, it makes sense: cross-platform applications are just perfect for the MVVM pattern. Putting as much code as possible in the shared Model and ViewModel layers leaves the way open to having a lean View layer, which can be interchanged on different platforms. And, if you have built-in support for databinding, like in Xamarin.Forms, you have all you need! (And if you don't, you can always benefit from external data-binding support like in MVVM Light.) The Xamarin community loves MVVM.

A few years ago, when my friend Stuart Lodge spun off MVVM Light and started modifying it into what was going to become MVVM Cross, I was quite intrigued by Xamarin, but I had other priorities at the time. Later, Xamarin became more stable, and I saw that it was time to port MVVM Light to it. In the meantime, MVVM Cross had grown to become a powerful framework. The learning curve was also rather substantial, and there was clearly a significant portion of the Xamarin community that was interested in MVVM Light on their platform too. The port was not very hard, but coming up with the data-binding framework as well as architectural guidance was definitely an interesting task. And it was really worth it—the adoption was great, as was the response to my two Xamarin Evolve talks (in 2014 and 2016) and the many meetups and conferences where I was able to show how to use MVVM Light on Xamarin.

I fully expect the same level of interest for Paul's book. Building cross-platform applications is a must these days, and it is quite hard to find an argument (other than inertia or fear of change) for not going that route. What cross-platform framework you select is up to you. We honestly believe that Xamarin is a great choice. The community is fantastic, the tools are powerful, and the productivity is excellent, all without sacrificing the rich, truly native user interface. I hope that you will find this book useful, and that it will help you make even more of MVVM Light and Xamarin. I am very impatient to see what you are going to build!

Happy coding,

Laurent Bugnion

Introduction

One of the advantages of having been involved in software development for 35 or so years is that I've seen technologies come and go. Languages rise and fall, programming ideas and processes explode without trace, and, well, this is the sort of thing you'd expect given the fluid and forever-evolving nature of the beast.

The emergence of software frameworks is nothing new; we've had plenty of them in PHP and other such languages. They aid in the development of software by providing features that aren't normally available, commonly used functions, and a fairly flexible but consistent approach to development.

For mobile development, frameworks would need to be platform specific, as the languages used are different on each platform (Objective C on iOS, Java for Android, and .NET on Windows Phone). It would not be possible—or at least easily possible—to create a framework that could be used once and used in many places.

When Xamarin first released their libraries, which used C# to create apps rather than ObjC or Java, the ability to create a framework for mobile became a reality, with the long-term goal of "write once, deploy many" now being easily available.

MVVM has often been compared to MVC on iOS and web where a data model is kept separate from the view controller, leading to some of the early examples of separation of concerns. MVVM takes it one step further by having an additional layer between the view and the data models: the view model. The view model does the majority of the work for the framework. It calls services (such as SQLite and web) as well as is code specific to the platform via Inversion of Control (IoC).

Data was provided back to the view using data binding. On Android and UWP, data binding was nothing new, but again, it was platform specific.

As developers, we're now getting spoiled by the choices of which MVVM framework to use. It would be unfair to compare one to another, as they offer different benefits and drawbacks. At the end of the day, the framework is up to you. The beauty of the MVVM programming paradigm is that irrespective of which one you use, the way they work at a base level is the same:

1. We always have a view model, which provides data to and from the UI to any sort of background service.
2. We always have some form of model; these can be as simple or complex as you need.
3. We always have some form of a view. The beauty is that the view can be anything and on any platform; the View Model doesn't care if you're on an X-Box, PlayStation, Mac, Android device, Windows device, TV, or even a car display—the same rules apply on all. The graphics layer for the UI is meaningless as well; it can be native or abstracted or use some other library. The binding system remains the same.
4. There is always some form of view model registration code.
5. View code is never in the View Model (this is important when developing for Xamarin Forms; a common mistake is to have UI code in the View Model, which defeats the purpose of true MVVM).

I hope that this book helps explain how it all works together in an easy-to-understand way. MVVM is not for everyone and not for every project, but in general using it will help in development.

CHAPTER 1

Installing MVVM Light

In order to use MVVM Light within your Xamarin application, you must first add it into the application. The process is different depending on whether you are doing this on a Mac (with VisualStudio for Mac) or Visual Studio. The reason for this is simple—on a Mac, you are not able to create for Windows Mobile. This may not *seem* like a big difference, but when setting up a project with MVVM Light, it is.

In this chapter, we'll create our project as well have a brief look at the two important files provided by MVVM Light: `ViewModelLocator.cs` and `MainViewModel.cs`.

Creating Your Shell Project

Irrespective of whether you are using Visual Studio on a Mac or PC, you will need to create a project, and for MVVM Light we need to create a shared project that provides us with a portable class library as well as the Android, iOS, and Windows Phone shell projects (the Windows Phone project will only be doable on a PC).

Visual Studio 2015 & 2017

To start, we create a new project (`File` ➤ `New` ➤ `New Project`). You will be presented with a window like that shown in Figure 1-1.

P. Johnson, *Using MVVM Light with your Xamarin Apps,* https://doi.org/10.1007/978-1-4842-2475-5_1

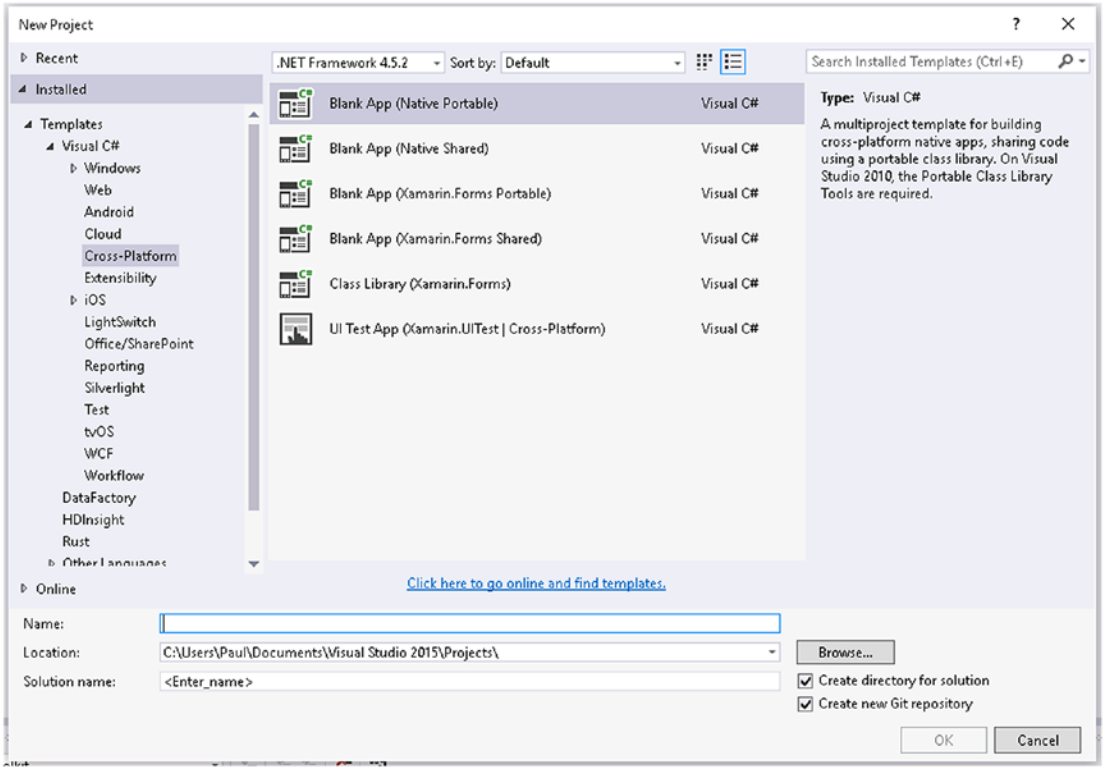


Figure 1-1. *Create a new project in Visual Studio*

Select "Blank App (Native Portable)." We give the application a name; for this project, I've called it `my_mvvm_app`.

Once the project has been created, you will see something similar to Figure 1-2 in Solution Explorer.

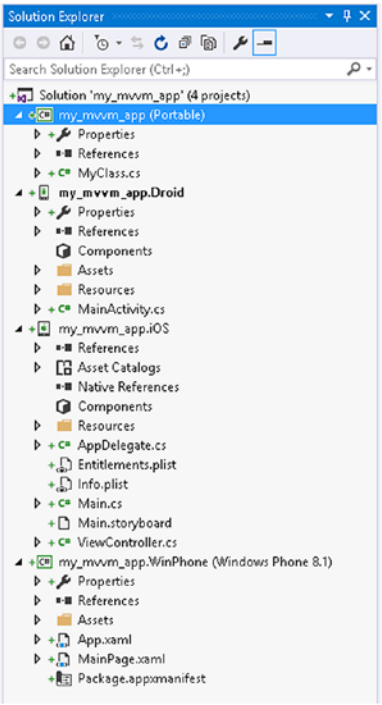


Figure 1-2. *Solution Explorer*

VisualStudio for Mac

As with Visual Studio on a PC, we start by creating a new application. This is done by selecting File ➤ New ➤ Project. When selected, you will be presented with a window similar to that shown in Figure 1-3.

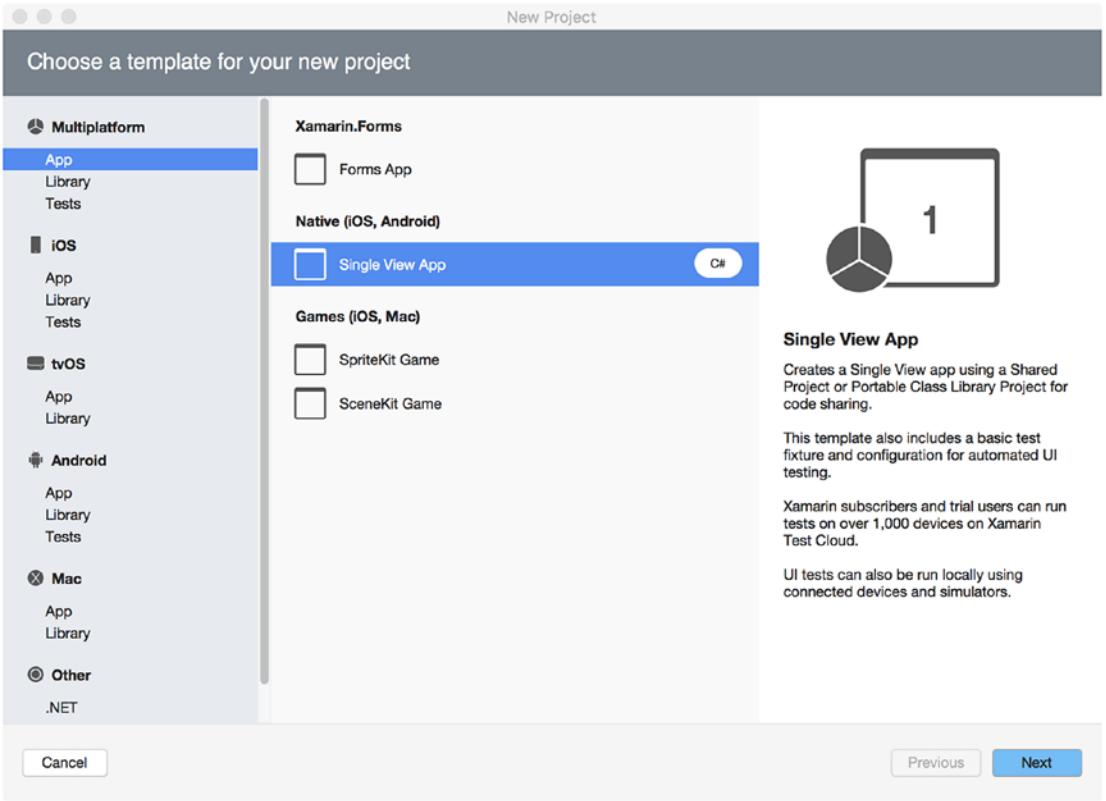


Figure 1-3. *VisualStudio for Mac. template selection*

You will need to select "Single View App" in the Multiplatform section. Once highlighted, click *Next*. The new project window will be replaced with the window shown in Figure 1-4.

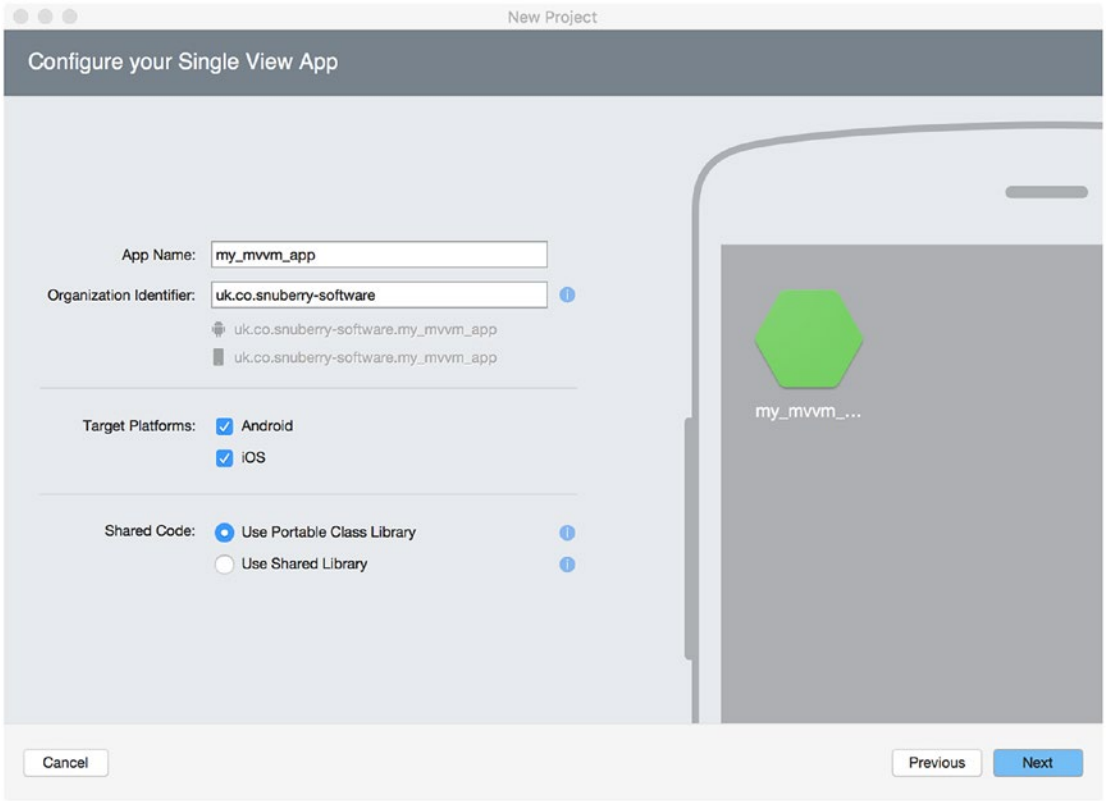


Figure 1-4. *Configuring your app*

Here, I have called the app `my_mvvm_app`. Once you have selected the target platforms and ensured "Use Portable Class Library" has been selected, click on *Next*. Again, the configuration window will close and be replaced with a new window, shown in Figure 1-5.

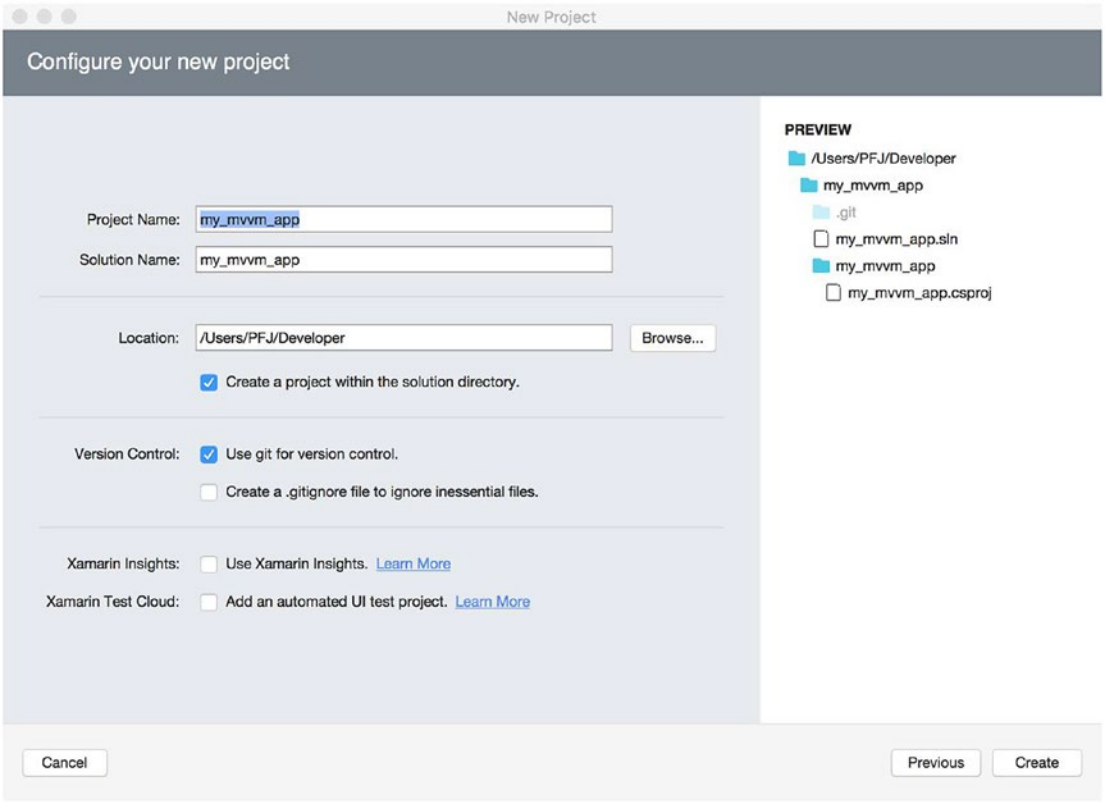


Figure 1-5. *The final new project window*

When you are happy with the settings, click on *Create* to create the project. Once created, you will see the following in Solution Explorer (Figure 1-6).

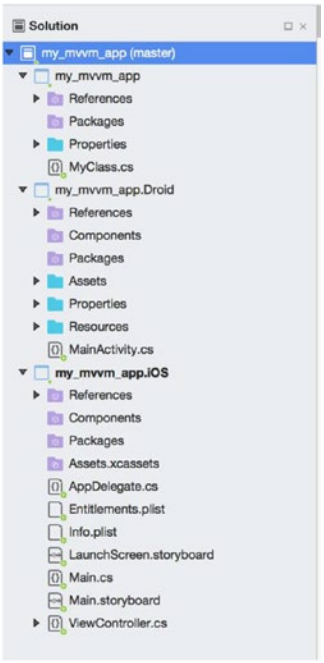


Figure 1-6. *Solution Explorer on VisualStudio for Mac*

■ **Note** If you create the project on a Mac, you can still add the Windows Phone project, but it will require that you copy the project to a PC and add to it a Windows Mobile project. The best way to do this is either to use a cloud-based service for where the source is stored (such as Dropbox, Google Drive, or Microsoft's One Drive) or to commit to a source repository service (such as GitHub). Remember, you cannot create the Windows Phone application in VisualStudio for Mac for Mac for the Mac.

Adding the MVVM Light Libraries

Now that we have created the projects, we need to add the libraries. By far, the simplest way to do this is by using the popular NuGet packaging service. Visual Studio on both PC and Mac has access to NuGet.

The only drawback to using NuGet is that the libraries must be added to each subproject and cannot just be added to, say, the PCL.

Adding MVVM Light to the Mac Project

You will notice that in each project there is a folder called `Packages`. Once a NuGet package has been installed, the name will show in this folder, with the library being shown in the `References` folder.

To add the library, select the packages, then press CTRL and click with the mouse (alternatively, highlight the `Packages` directory and, from the menu bar, select Packages ➤ Add Packages . . .). If you have used CTRL + click, you will see a window like that shown in Figure 1-7.

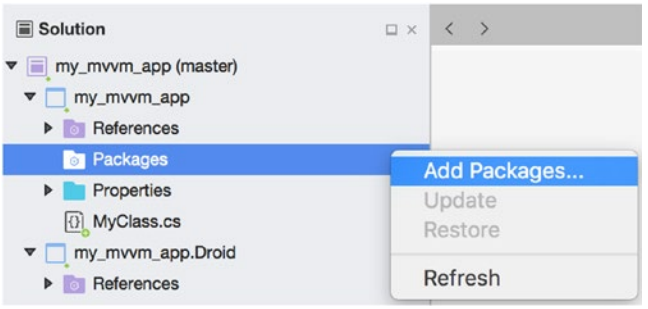


Figure 1-7. *Adding a new package to the PCL*

Once you have selected "Add Packages . . .," the NuGet package manager will show, as in Figure 1-8.

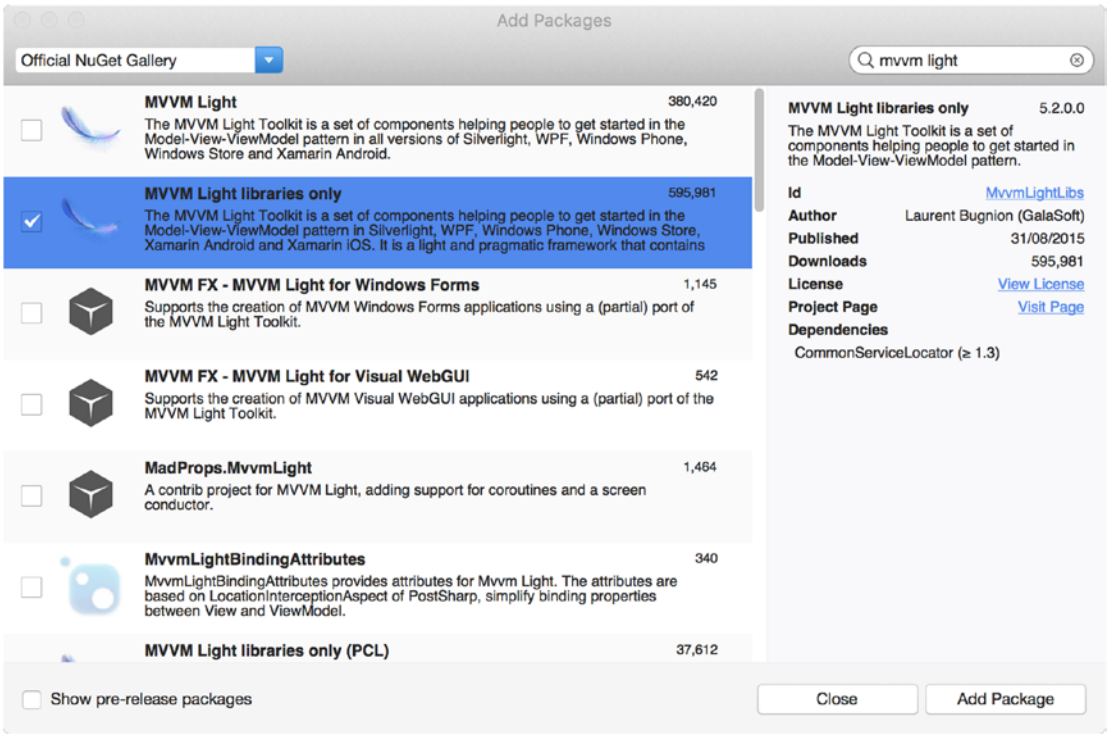


Figure 1-8. *Selecting MVVM Light*

In the top right (next to the magnifying glass), type *mvvm light*. The best matches will show. For the PCL, Android, and iOS, you will need to select "MVVM Light libraries only." Highlight and select "Add Package." This will install only the libraries, which contain nearly everything that you will need.

When the installation is complete, the project requires a small bit of extra work in the PCL.

What the installer does not provide are two files—the viewmodel and viewmodel locator. I have supplied these in the source code. These can be found the Chapter 1 source archive.

To add the `ViewModelLocator.cs file`, highlight the PCL project, press CTRL + click, and select "Add Files from Folder." Locate the `ViewModelLocator.cs` file and click *OK*. When asked if you want to move or copy, opt to copy the file.

Next, you will need to add the `ViewModels` directory and files. This can be performed by highlighting the PCL project, pressing CTRL + clicking, and selecting "Add Existing Folder." Locate the `ViewModels` directory and select "OK." You will be asked which files to import. There is a single file; ensure this file has a tickmark next to it. Click *OK*, and the folder will be imported.

Correcting the Namespace

As both the files are generic to the entire book, you will need to correct the namespace. First, we shall edit the `ViewModelLocator.cs` file. Double-click as you would normally to open the file. It is not difficult to see where the problem lies in the text editor, as typically all mistakes have a wobbly red line under them (Figure 1-9).

```
public MainViewModel Main
```

Figure 1-9. *The text editor indicates where the error is*

You will need to change the line

```
namespace mvvmlight1
```

to be

```
namespace my_mvvm_app.
```

Save the file. Next, open the `ViewModels` directory and open the `MainViewModel.cs` file. As with the `ViewModelLocator.cs` file, alter the namespace from `mvvmlight1` to `my_mvvm_app`. Save.

If you examine the `ViewModelLocator.cs` file again, the wobbly line under `MainViewModel` will now have disappeared. This means that the error has been corrected.

```
public MainViewModel Main
```

Figure 1-10. *The error has been fixed*

Unfortunately, you will need to do this for each new project in VisualStudio for Mac.

Add MVVM Light to the Visual Studio Project

Adding new packages to a Visual Studio project is very similar to doing so on a Mac. Select the `Packages` directory and right-click. You will be presented with a new menu over the `Packages` directory. Select "Manage NuGet Packages . . ." See Figure 1-11.

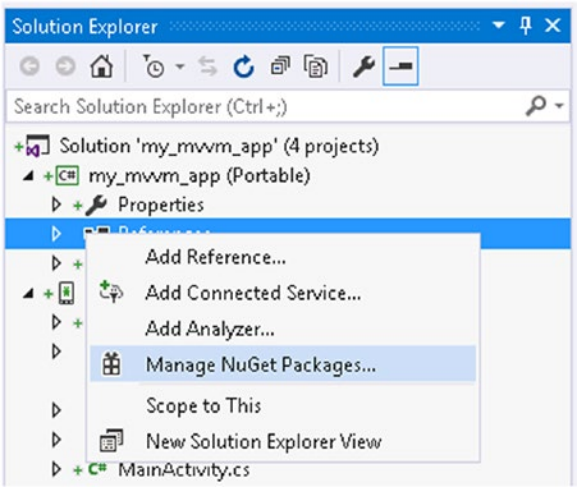


Figure 1-11. *Managing NuGet packages in Visual Studio*

Once selected, the NuGet manager window will open. It is different than the Mac NuGet window, but does the same job. Here though, you will need to select "Browse." Type *mvvm light* into the search box. For the Windows Phone package, you will need to install `MvvmLightLibs`. See Figure 1-12.

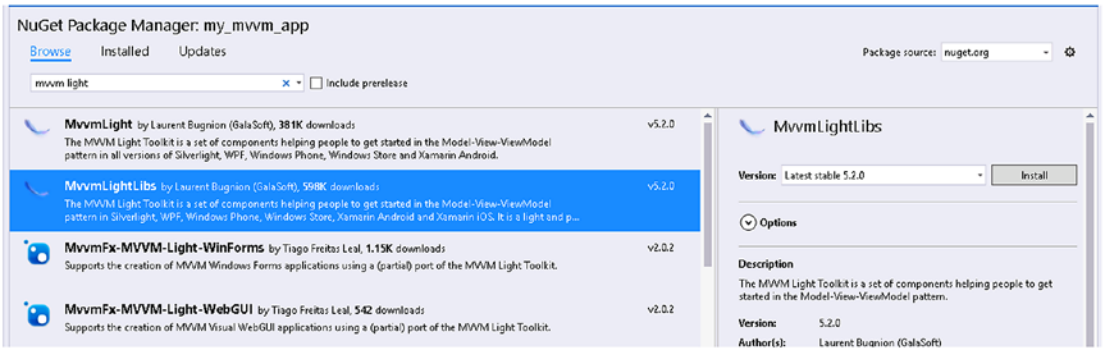


Figure 1-12. *Ensure you have selected the correct package*

Just before installing the package, if there are any licence agreements you are required to approve or turn down (which would stop the package's being installed), you will be presented with a window like that shown in Figure 1-13.

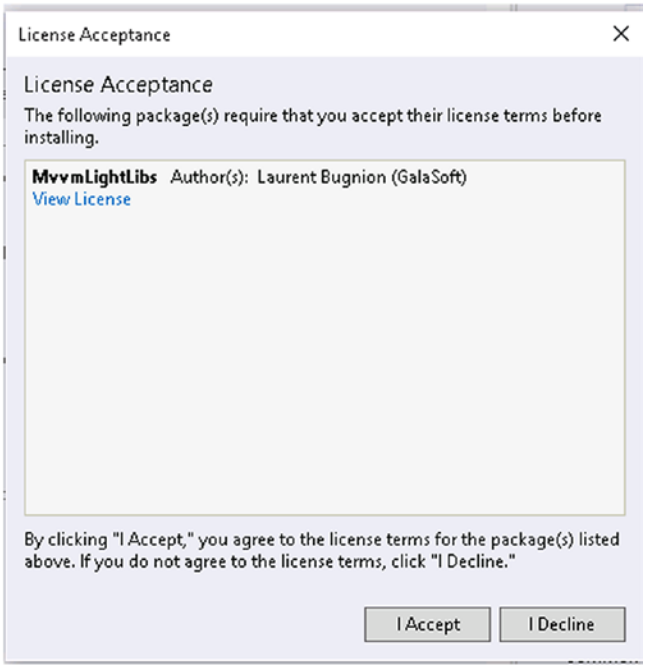


Figure 1-13. *Accepting the licence for the library*

Once you have accepted the licence, the package will install.

For the Windows Phone subproject, you will need to install the `Mvvm Light` package (not the libraries). See Figure 1-14.

Figure 1-14. *Use the full Mvvm Light package for Windows Phone & UWP*

The difference between the Libs and non-Libs NuGet packages is that the non-Libs package will install the `ViewModel` directory within the Windows Phone project. See Figure 1-15.

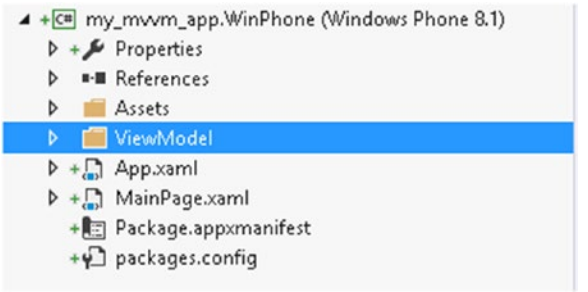
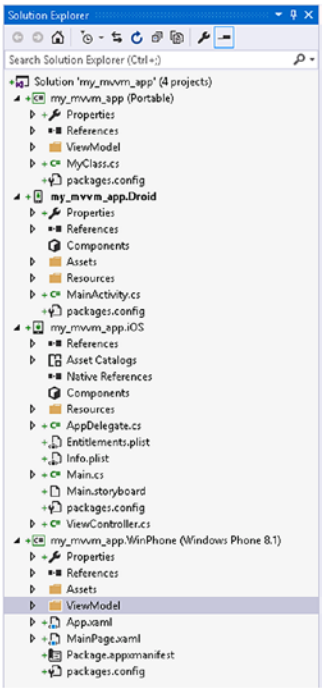


Figure 1-15. *The ViewModel directory is present in the Windows Phone package*

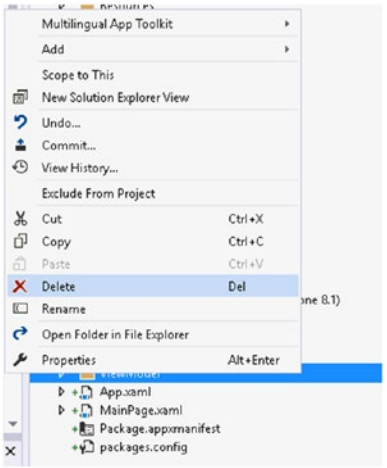
■ **Note** If you have installed the Libs version, you can either remove the Libs package or follow the instructions for adding the `ViewModel` directory as shown in the VisualStudio for Mac installation instructions.

We need to move the `ViewModel` directory from the Windows Phone project to the PCL. This is a very simple operation, as follows:

1. Highlight the `ViewModel` directory.
2. Drag the directory from the Windows Phone project to the PCL.

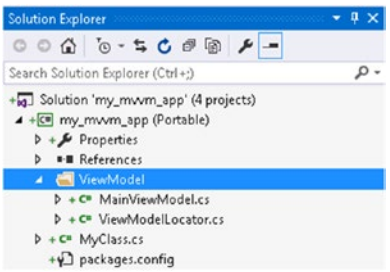


3. Delete the directory from the Windows Phone project.

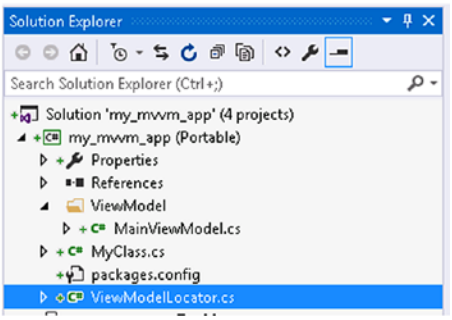


Once the directory has been removed from the Windows Phone project, we next need to move the `ViewModelLocator.cs` file from within the `ViewModel` directory to the base of the PCL. As with moving the directory, moving the file is simple.

4. Expand the `ViewModel` directory.



5. Highlight the `ViewModelLocator.cs` file and drag to the top of the PCL project (where it says `my_mvvm_app (Portable)`).



Correcting the Namespaces

As the folder was moved from one subproject to another, both the `MainViewModel.cs` and `ViewModelLocator.cs` files will be using the namespace of the Windows Phone package. Thankfully, this is a simple task to fix.

Open the `ViewModelLocator.cs` file and alter the line

```
namespace my_mvvm_app.WinPhone.ViewModel
```

to

```
namespace my_mvvm_app.
```

You will need to repeat this for the `MainViewModel.cs` file to ensure they are both using the same namespace.

Congratulations!

You've now set up your MVVM Light project—well, the basis of one at least! Before we finish this chapter, let's have a look at the two files provided by MVVM Light: `ViewModelLocator.cs` and `MainViewModel.cs`.

ViewModelLocator.cs

The `ViewModelLocator` file "registers" the `ViewModels` with the `ServiceLocator`. The service locator (by default, with MVVM Light) uses `SimpleIoC`. *IoC* stands for Inversion of Control. The easiest way to think of this is that normally the head (the mobile platform) controls the dog, but with IoC the tail (the PCL) controls the dog.

The registration service (`SimpleIoc.Default.Register<MainViewModel>();`) is also not as simple as it seems.

For example, we can register the `viewmodel`, or we can pass in an interface to a service as well as the service (such as `SimpleIoc.Default.Register<IMyTestService, TextService>();`). The reason for this is that it further separates concerns.

Separation of Concerns

If you have never used an MVVM framework before, then it is possible you have not heard of this phrase. The best way to think of this is to consider a simple app.

The app (to the user) is the user interface. They don't care what is running behind it, as long as it looks good and doesn't crash. However, to the developer, the app contains a number of blocks (user interface, business logic, server communications, etc.), and each block has its own purpose.

If the app has been created with a modicum of planning, then the UI won't be mixed with the business logic. The advantages of breaking each of these parts out are as follows:

- Easier to debug
- Easier to port between platforms
- Speed of development advantage
- Reusable code increased

The MVVM toolkits all work by letting the platform code deal with platform parts (user interface, known as the *view*). The *model* contains the data structures, and the *view model* deals with the data and events the view requires.

This is fine, but let's look at navigation between views. Consider, say, going between two activities on Android. To move between the activities, you would use something like this:

```
var intent = new Intent(this, typeof(MySecondActivity));
intent.PutExtra("pack", pack);
intent.PutExtra("CurrentStep", currentStep);
StartActvity(intent);
```

The problem here is that this navigation is handled completely differently between Android, iOS, and Windows Phone. Thankfully, we can further delineate the UI by removing the navigation from the view to a navigation service. This service is passed into the view model and can then be accessed directly. The service requires an interface and code that deals with the interface.

The `ViewModelLocator` registers the view models with the IoC system (MVVM Light can use any number of IoC libraries) as well as any interfaces and interface services. If a view model is not registered and the app tries to access it, the app will crash.

A pointer to the `ViewModelLocator` must also exist on each platform to be able to access the view models. Code such as

```
public static class App
{
   private static ViewModelLocator locator;

   public static ViewModelLocator Locator => locator ?? (locator = new ViewModelLocator());
}
```

will allow access to the ViewModelLocator.

MainViewModel.cs

The `MainViewModel` file is simply a stub file for the default view model. It inherits `ViewModelBase`. `ViewModelBase` contains a number of very useful methods for raising property-changing and -changed events that are used to trigger changes to the view. It also contains a single step method called `Set` that will set a property value as well as raise the property-changed event.

Conclusion

In this chapter, we've set up the project, installed MVVM Light from NuGet, and had a brief look at the `ViewModelLocator`, inversion of control, and separation of concerns. In the next chapter, we'll be creating our first application using the MVVM Light toolkit.

CHAPTER 2

Your First MVV MLight Mobile App

It's time that we jump in and start to create our first MVV MLight mobile app. We will show the planning, UI construction, separation of interests, and how it all comes together. As with any app, we need to know what it is we are going to write.

The source code for each part is included in the Chapter 2 source archive.

Welcome to Football Grounds

Football Grounds is a very simple application. In its first iteration, it will shuffle all of the FA Premiership teams and display the team at the top of the list. For the second iteration, a map with a pin for the grounds will be shown.

Let's consider the different parts of the application and how an MVVM application is "plumbed" together.

How an MVVM Application Is Constructed

As has already been stated, MVVM stands for Model–View–ViewModel. It is also fair to assume that, as the name suggests, the model feeds the view, which feeds the view model. This is not the case. Both the model and the view model exist within the common library (the PCL, in our case). The relationship can be seen illustrated in Figure 2-1.

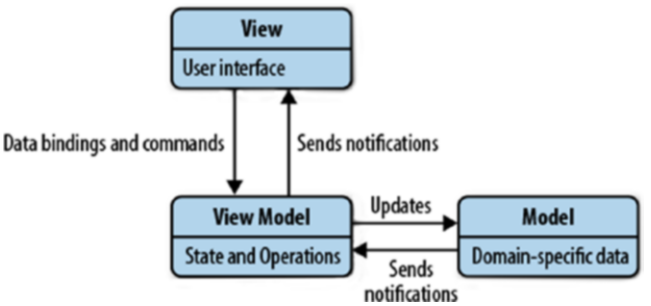


Figure 2-1. *The MVVM application "layout"*

The important part to understand is that the relationship between each concern and that in order for the model to talk to the view, it must go through the view model, and for the view to receive information, it too must go through the view model.

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_2

Notifications

The notifications are different for the model to view model than they are for the view model to view. For the model to view model, they are `PropertyChanged` events (from `System.ComponentModel`). The event can be created in one of two ways. The first is to use something akin to the following:

```
string myPropertyName;
public string MyPropertyName
{
    get
    {
        return myPropertyName;
    }
    set
    {
        myPropertyName = value;
        RaisePropertyChanged(“MyPropertyName”);
    }
}
```

When the view model sends a change to the model, the setter is called. Depending on how the view model is set up, it may emit an event that notifies the UI of a change to the property.

MVVM Light provides a simpler way to combine the two operations within the setter using an overloaded method called `Set`. For the preceding example, the following `Set` can be used:

```
set
{
    Set(() => MyPropertyName, ref myPropertyName, value);
}
```

There are a number of overloads for `Set`. All of the overloads assign the new value to the property. The `ObservableObject` is a class that inherits `INotifyPropertyChanged` and `INotifyPropertyChanging`, which are MVVM Light interfaces that wrap around the standard `PropertyChanged` and `PropertyChanging` events.

Name	Additional Description
`Set<T>(Expression<Func<T>>,T,T)`	Raises the `PropertyChanged` event if needed (inherits from `ObservableObject`)
`Set<T>(string, T, T)`	Raises the `PropertyChanged` event if needed (inherits from `ObservableObject`)
`Set<T>(T, T, string)`	Raises the `PropertyChanged` event if needed (inherits from `ObservableObject`)
`Set<T>(Expression<Func<T>>,T,T,bool)`	Raises the `PropertyChanged` event if needed and broadcasts a `PropertyChangedMessage` using the `Messenger` instance (or the static default instance if no `Messenger` instance is available)
`Set<T>(string, T, T, bool)`	Raises the `PropertyChanged` event if needed and broadcasts a `PropertyChangedMessage` using the `Messenger` instance (or the static default instance if no `Messenger` instance is available)
`Set<T>(T, T, bool, string)`	Raises the `PropertyChanged` event if needed and broadcasts a `PropertyChangedMessage` using the `Messenger` instance (or the static default instance if no `Messenger` instance is available)

Between the view model and the view, we notify the UI from the view model using `RaisePropertyChanged` and `RaisePropertyChanging`. This changes the value that is bound between the view model and the view. Bindings are simply a data channel between the view and the view model used to relay data. Bindings are typically two-way systems, but can also be one-way. The data is updated in the view model, and the UI responds to an event. The event (for example) will be analogous to `TextChanged`. However, as each platform handles the text-changed event differently, the event used will not be the same.

The difference in events used among the platforms is a very good demonstration of the beauty of the MVVM pattern. The UI is kept apart from the business logic. We are able to modify the user interface to fit the design pattern for that platform (for example, if we have a list of data, Android and Windows Mobile will use a `ListView`, while iOS will use a `UITableView`. How each platform constructs that user interface is up to that platform, but the code will be highly optimized for that platform. Each platform has different events that deal with a click event on a list item as well).

We will take a closer look at this while discussing the UI element of the app.

Let's Design the App

The app consists of 20 data elements, each of which contains five pieces of information: football team, ground name, maximum attendance, latitude, and longitude.

In the first iteration of the app, the user will be able to enter how many times they want the team data elements to be shuffled, and once completed the top data element will be displayed.

For the second iteration, the app will show a map for the grounds.

This is nothing groundbreaking in itself, but it will show first how simple using MVVM Light is and second how to access a platform-specific UI element.

Constructing the Model

The model itself is the following class:

```
public class Cards
{
    public string TeamName { get; set; }
    public string StadiumName { get; set; }
    public int Capacity { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
}
```

We can store the models in a `List`:

```
List<TeamModel> FootballTeams;
```

The data then needs to be entered. There are three ways to do this:

1. Have the user enter the data
2. Read the data in from an XML file
3. Have the data embedded within the model

For maximum flexibility, option 1 would be best. However, we're not in the 1980s when it was common to spend an hour or two typing things in from a magazine. If the app were extended to add and remove clubs from the Premiership, having the user enter data would keep things up to date.

The second option is to read the file in. The question here is, do we read the data in from the platform or from the PCL? The PCL contains a cut-down version of the standard .NET libraries that only contains the libraries and facilities guaranteed to be supported by all platforms. To this end, there is no access to `System.IO.File`, so to load the data in, we would have to read the XML file into a stream and then pass that stream into XmlReader. The main advantage here is that, once a year, the app could download an updated version of the XML file.

If we have the file within the app and have the app read the file and pass that data directly to the model, we would break the flow. We could also create a service with an interface and pass that in via the `ViewModelLocator`. Plenty of options.

The final option has a disadvantage similar to the first option in that the data has to be physically entered. The second downside is that with each new season, a new version of the app would have to be released.

Constructing the View Model

The view model has to provide the data from the model to the view and, when the button is clicked, shuffle the data objects a number of times and then send the result to the UI. The view model requires a single property to store the number of "shuffles" for the loop.

The Click event comes from the UI and is acted upon as a `RelayCommand. RelayCommands` can be considered the MVVM event (though it is closer to being an ICommand than what would be considered an event).

Setting the ViewModelLocator

The `ViewModelLocator` is probably the single most important file in the whole of your MVVM Light projects. It sets up the references to each view model and sets up any form of injection and injection services (such as navigation).

For our project, we have at the heart of the app the following code:

```
public ViewModelLocator()
{
     ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
     SimpleIoc.Default.Register<MainViewModel>();
}

public MainViewModel Main
{
    get
    {
        return ServiceLocator.Current.GetInstance<MainViewModel>();
    }
}
```

The `ViewModelLocator` constructor is used to register all of the view models and any other services required. The line `SimpleIoc.Default.Register<MainViewModel>()` creates a single instance of the `MainViewModel` for use in the `Main` property. The `Main` property returns a new instance of the `MainViewModel`.

We will see in later chapters how to add services and other view models to the locator.

Constructing the User Interface (View)

For the first iteration, we need the user interface to display the team, grounds name, and grounds capacity. The GPS coordinates aren't required for display. We also need a text input and a button.

■ **Note** The source code for the project is in Chapter2/FootballCards_01.

Let's have a look at the project layout. In the Solution Explorer, you'll notice a couple of extra directories. See Figure 2-2.

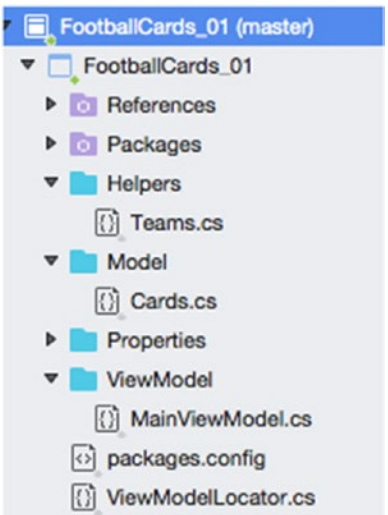


Figure 2-2. *Expanded PCL in the Solution Explorer (VisualStudio for Mac)*

With any of the MVVM toolkits, you are encouraged to break down your code into their own named parts. For example, the Model directory should contain only the models and not the propagation of the model. If you have defined any interfaces, they should be stored in the Interface directory. Helpers are the utility methods to aid common development modules.

We have already seen the model that we will be using. The Cards file simply returns a `List<Cards>` containing all of the teams in the 2015–2016 England Premiership.

```
public static List<Cards> GenerateCards
{
    get
    {
        return new List<Cards>
        {
            new Cards {TeamName="Liverpool", Capacity=45276, StadiumName="Anfield",
            Latitude=53.430833, Longitude=2.960833 },

            new Cards {TeamName="West Ham United", Capacity=35345, StadiumName="Upton
            Park",Latitude=51.531944, Longitude=0.039444 },
            /* other teams here */
```

```
            new Cards {TeamName="Aston Villa", Capacity=42682, StadiumName="Villa Park",
            Latitude=52.509167, Longitude=1.884722 },

            new Cards {TeamName="Tottenham Hotspurs", Capacity=36284, StadiumName="White
            Hart Lane", Latitude=51.603333, Longitude=0.065833 }
        };
    }
}
```

The View Model

The view model hooks the user interface together with the models. The view model sends the data to the UI as well as waits for the button to be pressed. The button's being clicked triggers a `RelayCommand` within the view model.

The data in and out are mostly handled by properties within the VM.

We start the view model with the constructor. For our simple app, we instantiate `List<Cards>` with our `Helpers.GenerateCards` property. We then take the "top" card and use that to propagate the other UI properties.

```
public class MainViewModel : ViewModelBase
{
    List<Cards> FootballCards;
    RelayCommand buttonClicked;

    public MainViewModel()
    {
        // propogate the Football cards List
        FootballCards = Helpers.Helpers.GenerateCards;

        // give the number of shuffles a default
        NumberOfShuffles = 0;

        // fill the UI properties
        var firstTeam = FootballCards.First();

        TeamName = firstTeam.TeamName;
        StadiumName = firstTeam.StadiumName;
        Capacity = firstTeam.Capacity;
        Longitude = firstTeam.Longitude;
        Latitude = firstTeam.Latitude;
    }

    // number of shuffles
    int numberShuffles;
    public int NumberOfShuffles
    {
        get {
            return numberShuffles;
        }
        set {
            Set(() => NumberOfShuffles, ref numberShuffles, value, true);
                // the final true means to broadcast the event
```

```
                if (numberShuffles > 0)
                    buttonClicked.RaiseCanExecuteChanged();
                // enables the click
            }
        }

        string teamName;
        public string TeamName
        {
            get {
                return teamName;
            }
            set {
                Set(() => TeamName, ref teamName, value, true);
            }
        }
```

Once the properties have been set up, we come to the main workhorse of the view model, the `RelayCommand`. This is the one we have in the view model:

```
public RelayCommand ButtonClicked
{
    get {
        return buttonClicked ??
        (buttonClicked = new RelayCommand(
             () =>
             {
// Shuffle the cards NumberOfShuffles times
                 FootballCards = Helpers.CardShuffle.Shuffle(FootballCards,
                 NumberOfShuffles);

                 // get the first card
                 var topCard = FootballCards.First();

                 // propogate the properties
                 TeamName = topCard.TeamName;
                 StadiumName = topCard.StadiumName;
                 Capacity = topCard.Capacity;
                 Longitude = topCard.Longitude;
                 Latitude = topCard.Latitude;
             }));
      }
}
```

The code first checks to see if `buttonClicked` is `true` (this equates to the UI element's being enabled). If it isn't, the command simply returns `false`. If it is `true`, the cards are shuffled, the top card is taken, and the properties are modified.

The shuffle routine is a recursive method in the Helpers directory:

```
public static class CardShuffle
{
    static Random rng = new Random();
    static int Shuffles = 0;

    public static List<Cards> Shuffle(this List<Cards> list, int shuffles)
    {
         if (Shuffles < shuffles)
         {
             int n = list.Count;
             while (n > 1)
             {
                n--;
                int k = rng.Next(n + 1);
                var value = list[k];
                list[k] = list[n];
                list[n] = value;
             }
             Shuffles++;
             Shuffle(list, shuffles);
         }
         return list;
     }
}
```

As the `RaisePropertyChanged` event is being raised (the final true parameter in the `Set(()=>)` construct), this will also update the UI.

For now, the PCL is complete. On to the UI.

The Platform User Interface

■ **Note** Code for this section is in Chapter2/FootballCards_02.

For each platform, the UI has been kept as simple as possible.

1. Android

Our UI for Android looks like that seen in Figure 2-3.

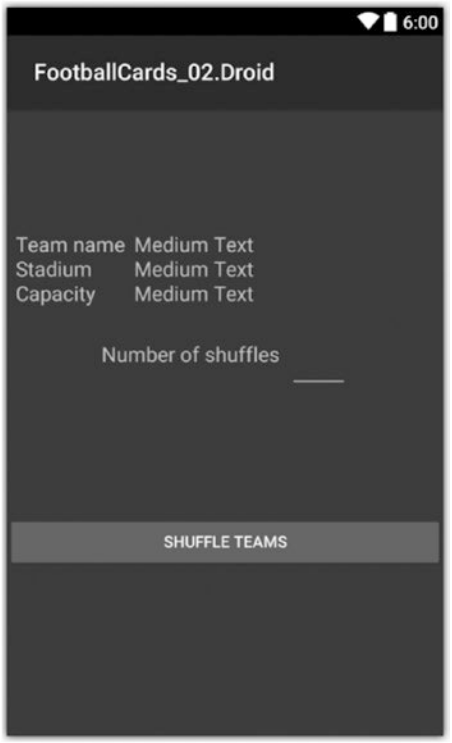


Figure 2-3. *The Android UI. The Medium Text elements will be replaced with data from the view model.*

We have three text views, one edit view, and one button.

Normally, the view objects would be instantiated using code such as `var myTextView = FindById<TextView>(Resource.Id.someTextView);` from within the `OnCreate` method. As we need to access these objects from outside of the `OnCreate` method, we need to create them slightly differently. Modify the activity to be a partial class, with the UI elements being created in a different partial class. While we could also just create global variables within the activity, using the partial classes not only keeps the code clean, but also reduces the amount of work being carried out in the `OnCreate` method, which speeds up the creation of the main view.

Let's see how to do this.

i. Modify `MainActivity` to be a partial class.

```
public class MainActivity : Activity
```

becomes

```
public partial class MainActivity : ActivityBase
```

■ **Caution!** Be aware that the inherited class is not `Activity` but `ActivityBase`. This base class is provided by MVVM Light for Android.

ii. Create the second part of the partial class to contain the UI elements.

```
public partial class MainActivity
{
    // create the private references to the objects

    Button btnShuffle;
    TextView txtTeamName, txtStadium, txtCapacity;
    EditText editShuffles;

    // create the public properties for the objects

    public Button BtnShuffle => btnShuffle ?? (btnShuffle = FindViewById<Button>(Resource.
    Id.btnShuffle));

    public TextView TxtTeamName => txtTeamName ?? (txtTeamName =
    FindViewById<TextView>(Resource.Id.txtTeamName));
    public TextView TxtStadium => txtStadium ?? (txtStadium =
    FindViewById<TextView>(Resource.Id.txtStadium));
    public TextView TxtCapacity => txtCapacity ?? (txtCapacity =
    FindViewById<TextView>(Resource.Id.txtCapacity));

    public EditText EditShuffles => editShuffles ?? (editShuffles =
    FindViewById<EditText>(Resource.Id.editShuffles));
}
```

There needs to be a reference to the `ViewModelLocator` from the PCL in order for the app to access the view model itself. The following class handles this:

```
public static class App
{
    private static ViewModelLocator locator;

    public static ViewModelLocator Locator => locator ?? (locator = new ViewModelLocator());
}
```

with the following line being added to the `Activity` class:

```
public MainViewModel ViewModel => App.Locator.Main;
```

We next need to deal with the event handling. We have two events: the button press and the edit text changing. The text-changing event is handled by the Android MVVM Light library directly. For the button press, we create a static class containing a string reference to the event name:

```
public static class Events
{
    public const string Click = "Click";
}
```

We next need to create the bindings. For this we create a call in `OnCreate` to the method or methods we use for them.

For `EditText` and `TextView`, the binding is set like this:

```
this.SetBinding(
            () => ViewModel.TeamName,
            () => TxtTeamName.Text,
            BindingMode.TwoWay);
```

Binding Modes

The binding mode refers to how the data flows. It can be two way (data goes both ways; this is the default), one way (data only goes from the view model to the view), one-time (the binding is only updated once, and once only), or default (two way).

Setting the Button Command

A `button` needs to set the following command:

```
void CreateButtonBinding()
{
     BtnShuffle.SetCommand(
     Events.Click,
     ViewModel.ButtonClicked);
}
```

We want to respond to the `Click` event, and when the event is raised we want to run the `ButtonClicked` code from the view model itself. Once that has executed, the UI will change.

2. iOS

As with Android, we have a very simple user interface, as seen in Figure 2-4.

Figure 2-4. *The iOS user interface. It has been designed using the universal class setting to ensure it works on all sizes of iPhone. The app has been set to be iPhone/iPad rather than universal.*

We need to create a class to handle the events in much the same way as we did for Android. Unlike for Android, we also need to define `EditingChanged`. The reason for this is `simply down` to how iOS operates.

Ahead of Time and Just in Time UI Systems

Android and Windows Mobile both use a system called "Just in Time." Everything is done just before it is needed, so when a value is updated in the view model just before the view is rendered (or a process is required), the UI is given the most up-to-date data. This greatly cuts down on the amount of work an app (and therefore the OS) is required to do to ensure the UI is correct.

iOS uses "Ahead of Time"; that is, the UI wants to know exactly what is going to be there way ahead of when it is going to be needed. This ensures that the user gets an ultra-responsive experience. The problem is that once the OS knows what is there, any further changes aren't going to be displayed. Here, we have to force an update of a UI element by raising an event.

Is there an advantage of one system over the other? Yes and no. There is a great advantage for the user experience in the Ahead of Time model. It is instant and will look great. For the developer, AoT is a pain to handle, as you're having to do the work for the application during the creation whereas JiT is done for you.

Wiring Up the Bindings

The UILabels and UIButton are created in the exact same way as for Android. For the UITextField, we need a slightly different approach:

```
void BindTextField()
{
    this.SetBinding(
            ()=>txtShuffles.Text).UpdateSourceTrigger(Events.EditingChanged).
            WhenSourceChanges(()=>ViewModel.TextNumberOfShuffles = txtShuffles.Text);
}
```

This may look slightly complex, but what it is saying is that when the text-field editing has changed (in other words, we've finished editing), we set the ViewModel.TextNumberOfShuffles to equal the value of the text field. Without this code, the view model data is not updated.

Referencing the View Model

The reference to the view model can be placed in one of two files: Main or AppDelegate. While there is nothing wrong with having the reference in Main, as your applications grow larger, Main too will become larger. At this point, I can hear you say, "So? Won't it make FinishedLaunching larger?" and the answer is yes, it will. Here, you have to consider what Main is for—it is simply the .NET code that launches the iOS application. You also need to consider that all iOS applications have around 15 seconds to launch fully before the operating system considers them "dead" and terminates them—the more in Main, the longer it takes.

In the AppDelegate, the ViewModelLocator is referenced like this:

```
static ViewModelLocator locator;

public static ViewModelLocator Locator
{
    get {
            return locator ?? (locator = new ViewModelLocator());
        }
}
```

which is then referenced in the UIViewController like this:

```
public MainViewModel ViewModel => AppDelegate.Locator.Main;
```

■ **Note** If your main project for building is not the one you are working on, you may see a red wavy line under Events.TouchUpInside. This is nothing to worry about. If you set the device to reflect the one you are editing (in this case, Debug ➤ iOS Device), the line will disappear.

---

3. Windows Mobile

As with the other two platforms, our user interface is very simple, as seen in Figure 2-5.

Figure 2-5. *Our Windows Mobile user interface*

■ **Note** If you are developing your code on a Mac, you will not be able to create a Windows Mobile version of the app unless you have a virtual machine of some type running on the Mac.

Accessing the ViewModelLocator

The best part of developing for Windows Phone is that most of the problems with binding and accessing the PCL don't exist as soon as the `App.xaml` is pointing at the correct place for the view model.

If you open `App.xaml` in VisualStudio, you will see a section of XML that reads similar to this:

```
<ResourceDictionary>
      <vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True"
      xmlns:vm="using:FootballCards_02.WinPhone.ViewModel" />
</ResourceDictionary>
```

This is telling the resource dictionary that all of the view models that we are using can be found in the `ViewModelLocator`, which is in the ViewModel directory (this is the one we moved to the PCL at the start).

We can point the `ViewModelLocator` to a different location by changing `xmlns:vm="using:FootballCards_02.WinPhone.ViewModel"` to `xmlns:vm="using:FootballCards_02"`.

As far as the locator goes, we need to do nothing else.

Binding on Windows Phone

Windows Phone creates all user interfaces using a form of markup called XAML (Extensible Application Markup Language). This XAML is compiled to generate the UI and is often referred to as being the "code in front" (it is in front of any controlling code that sits behind it). For this project, the code behind simply tells the compiler to generate the code; however, the degree of code behind depends on what is being called from the PCL (such as data passed from a different view).

Our XAML is held in the `MainPage.xaml` file.

As it stands, even though we have set the global `App.xaml` to point to the `ViewModelLocator`, we still need to tell each view that we want to use that locator. This is performed by setting the `Page DataContext` binding to point to the locator.

Let's have a look at the definition directly beneath the `<Page` opening tag:

```
x:Class="FootballCards_02.WinPhone.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:FootballCards_02.WinPhone"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d"
Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

For the most part, we don't need to worry about any of this. The important lines are the `x:Class` and `xmlns:local` lines as these tell the compiler the class the XAML file "belongs" to and the namespace in use for the XAML file respectively.

To set the `DataContext`, the following line is added anywhere before the closing `>` brace:

```
DataContext="{Binding Main, Source={StaticResource Locator}}"
```

Blend is now able to access the `ViewModelLocator`, and Intellisense will be able to pick up on the properties in the locator. `Main` is the static string we have used in the `ViewModelLocator` for the `MainViewModel`.

Binding UI Elements

Thankfully, binding on Windows Phone is extremely simple. Take the team name element. In code, when we define it, the markup looks like this:

```
<TextBlock x:Name="txtTeamName" FontSize="14" Margin="8,0" HorizontalAlignment="Stretch"
Grid.Column="1"/>
```

It defines the name we want to refer to it as, the column it sits in, and a bit of spacing. We want to bind the `Text` property to the view model. This is achieved by adding `Text="{Binding TeamName, Mode=TwoWay}"` to the markup for the `TextBlock`. All this means is that we want to create a two-way binding

to the TeamName property in the view model. In exactly the same way, we can bind the TextBox to the view model. We define the BindingMode to be two-way explicitly as by default it is one-way.

To connect the button, we need to bind the Command property to the view model:

```
<Button x:Name="btnShuffle" Content="Shuffle" Margin="0,32" Command="{Binding ButtonClicked,
Mode=TwoWay}" HorizontalAlignment="Center"/>
```

The button does have a Click event as well, which we will discuss in a later chapter. The Click event is handled in the code behind.

Removing Old Code

There is a small amount of old code in Main.xaml.cs to handle a button that no longer exists. It is in the OnNavigatedTo method. Above the MainPage constructor is a private int variable. It is safe to remove this as well.

Compiling and Running the Apps

Assuming that everything has gone correctly, you should now be able to compile and run the applications as you would any other mobile application and see the results shown in Figures 2-6a to 2-8b.

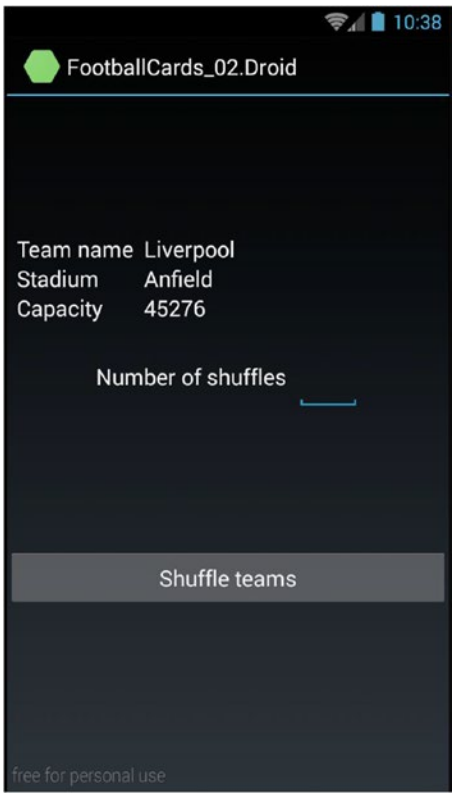


Figure 2-6a. *Android version on startup*

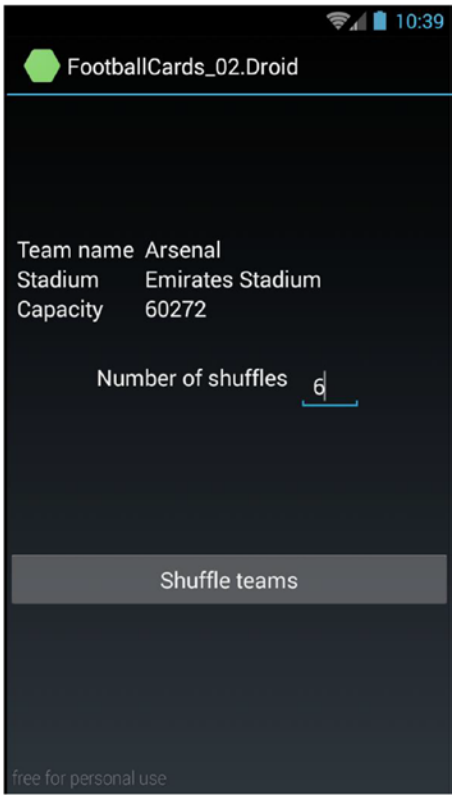


Figure 2-6b. *Android version after performing a shuffle*

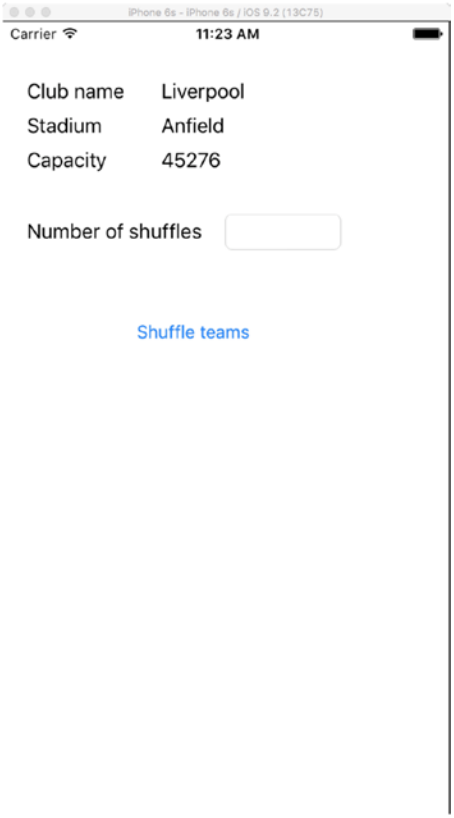


Figure 2-7a. *iOS version on startup*

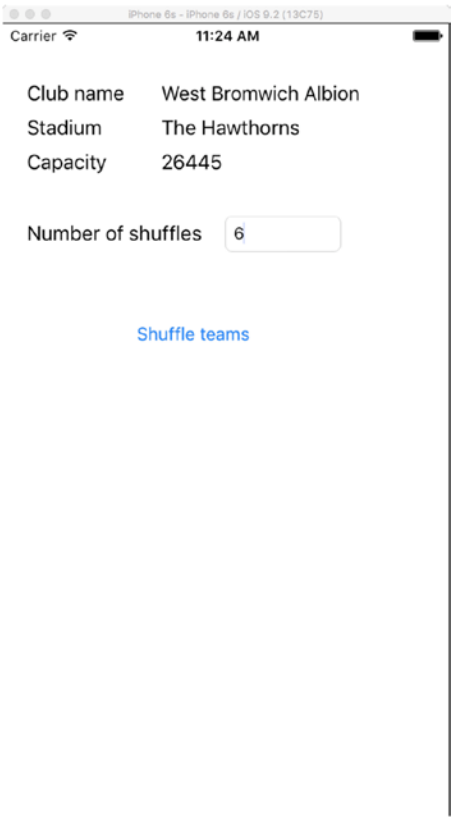


Figure 2-7b. *iOS version after a shuffle*

Figure 2-8a. *Windows Phone version on startup*

Figure 2-8b. *Windows Phone version after a shuffle*

It may have struck you by now that with very little effort, we have created a very simple application for three platforms. If something goes wrong with the UI, we only really need to look at the UI code. If something goes wrong with the logic, then we only need to start looking within the PCL, and as the MVVM design pattern works, we should know where to start the debugging, as each part of the PCL is split into its respective parts.

We can modify our UI directly to pretty up the user experience on, say, Android without interfering with our iOS or Windows Mobile applications by using all of the tricks Android developers use (such as using 9-patch, XML overlays, styles, and so on). Later in the book, we'll look at moving an existing app from being for one platform to being for three.

Extending Our Application

Up to this point, we've had an app with a single view that shuffles some cards. You may recall that within our model, we stored the latitude and longitude of each stadium. The reason for this is so that we could extend our application to include a map with a pin on the stadium.

Implementing this will mean that we will also need to add a new view (which will involve a new view model), and in order to do this, we need a mechanism by which to move between view models.

The problem is that we need to implement a navigation system, and this means that for each platform we have to create a mechanism to move. Or do we?

If you consider the purpose of any MVVM toolkit, it is to take as much control from the platform into the PCL so that the platform is really just there for the user interface. The processing and keeping track of the navigation stack can be safely abstracted away from the platform.

MVVM Light provides a navigation service that enables movement between views. The navigation service—despite being part of the PCL—has to be defined on the platform level.

Adding the Navigation Service

■ **Note** Code for this section is in Chapter2/FootballCards03.

Adding the navigation service is a two-step process: within the PCL and external to the PCL.

Adding to the PCL

1. Create a new view model within the PCL and ensure that it inherits `ViewModelBase`. For the time being, we can just leave the file blank.
2. Open the `ViewModelLocator.cs` file.

The navigation system works on a key/ID basis (the ID is an object and not a specific type). In order to create the keys, we define them within the `ViewModelLocator`. We can call the keys whatever we wish, as we only ever refer to them on the platform level through the static instance of the `ViewModelLocator` and never directly.

For our app, we have the following:

```
// set up keys for the pages
public const string MainPageKey = "FirstPage";
public const string MapPageKey = "MapPage";
```

We next need to register the second view model and create a property for access.

Within the `ViewModelLocator` constructor, the following line is added to register the view model that we created in Step 1:

```
// add the map view model
SimpleIoc.Default.Register<MapViewModel>();
```

The property is virtually as for the main view model:

```
// add the map page propery
public MapViewModel Map
{
    get {
        return ServiceLocator.Current.GetInstance<MapViewModel>();
    }
}
```

Adding to the Platforms: iOS

For iOS, we register with the navigation service within the `FinishedLoading` method within `AppDelegate`:

```
// create the instance of the navigation service
var nav = new NavigationService();

// iOS uses the UINavigtionController to move between pages, so will we
nav.Initialize(this.Window.RootViewController as UINavigationController);

// we configure the navigation service to take the key, with the second page being the
storyboard ID
// the storyboard ID is set in the storyboard and is not the filename
nav.Configure(ViewModelLocator.MainPageKey, "MainPage");
nav.Configure(ViewModelLocator.MapPageKey, "MapPage");

// finally register the service with SimpleIoc
SimpleIoc.Default.Register<INavigationService>(()=>nav);
```

We will also need to modify the storyboard ID for the existing UI to be the same as we have defined in our first `nav.Configure` line. See Figure 2-9.

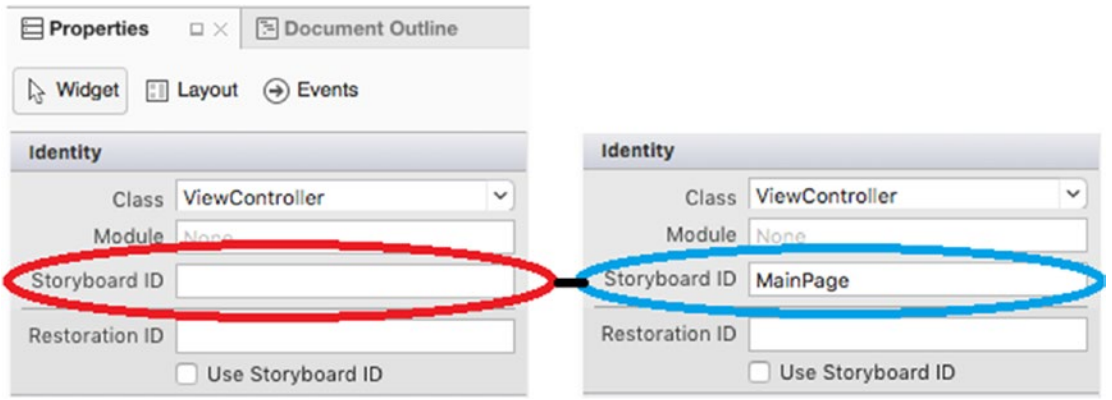


Figure 2-9. *Modify the storyboard ID*

Adding to the Platforms: Android & Windows Mobile

For Android and Windows Mobile, we use the same code. For both, we place the code in the `App` class.

For Windows Mobile, the following code is used:

```
// create an instance of the NavigationService
var nav = new GalaSoft.MvvmLight.Views.NavigationService();

// configure the service
nav.Configure(ViewModelLocator.MainPageKey, typeof(MainPage));
nav.Configure(ViewModelLocator.MapPageKey, typeof(MapPage));
```

```
// register the service
SimpleIoc.Default.Register<INavigationService>(()=>nav);
```

Android is different again. In `FootballCards_02`, our `App` class was not a particularly complex piece of code. We replace that code with a new class that inherits `Application` and is not a static class:

```
[Application(Icon = "@drawable/icon")]
public class App : Application
{
    static ViewModelLocator locator;

    public App(IntPtr h, JniHandleOwnership jho) : base(h, jho)
    {
    }

    public static ViewModelLocator Locator
    {
        get {
            if (locator == null)
            {
                // First time initialization
                var nav = new NavigationService();

                // configure the navigation service
                nav.Configure(ViewModelLocator.MainPageKey, typeof(MainActivity));
                nav.Configure(ViewModelLocator.MapPageKey, typeof(MapActivity));

                // register with the Navigation Service
                SimpleIoc.Default.Register<INavigationService>(() => nav);

                locator = new ViewModelLocator();
              }

              return locator;
          }
     }
}
```

■ **Note** Whenever a large change such as this is made, it is always advisable to test the code before moving on. If the storyboard ID (on iOS) was not changed to be the same as for the navigation service, the application will crash.

Using the Navigation Service Within the View Model

In order to use the navigation service within the view model, we need to pass in the `INavigationService` to the `ViewController` constructor. We also create a private copy of the `INavigationService` for use outside of the constructor.

```
INavigationService navigationService;

public MainViewModel(INavigationService navigation)
{
    // make a local copy of the interface
    navigationService = navigation;
```

We now have access to the service and can use it to navigate through our app.

Adding Access to Maps: iOS

Post–iOS 7, we need to add two additional lines to the `info.plist` file. This can be performed by editing the file in a text editor or by opening the plist file, selecting the source, and adding them directly.

```
<key>NSLocationAlwaysUsageDescription</key>
<string>Can we use your location</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>We are using your location</string>
```

Adding Access to Maps: Windows Mobile

We need to allow location services. This can be found in the file `Package.appxmanifest` under the `Capabilities` tab. Ensure "`Location`" is ticked.

■ **Note** If you intend to release an app using maps on the Windows Store, you will need to follow the instructions held here (for Windows Phone 8.1): https://msdn.microsoft.com/en-us/library/windows/apps/jj207033(v=vs.105).aspx.

Adding Access to Maps: Android

In order to use maps on Android, we need to enable a number of permissions for the app. This is achieved by opening the Android package properties and clicking on the Android Application sidebar tab. You will be presented with a window as Figure 2-10. Ensure the four location options are ticked and, further down, "`INTERNET`."

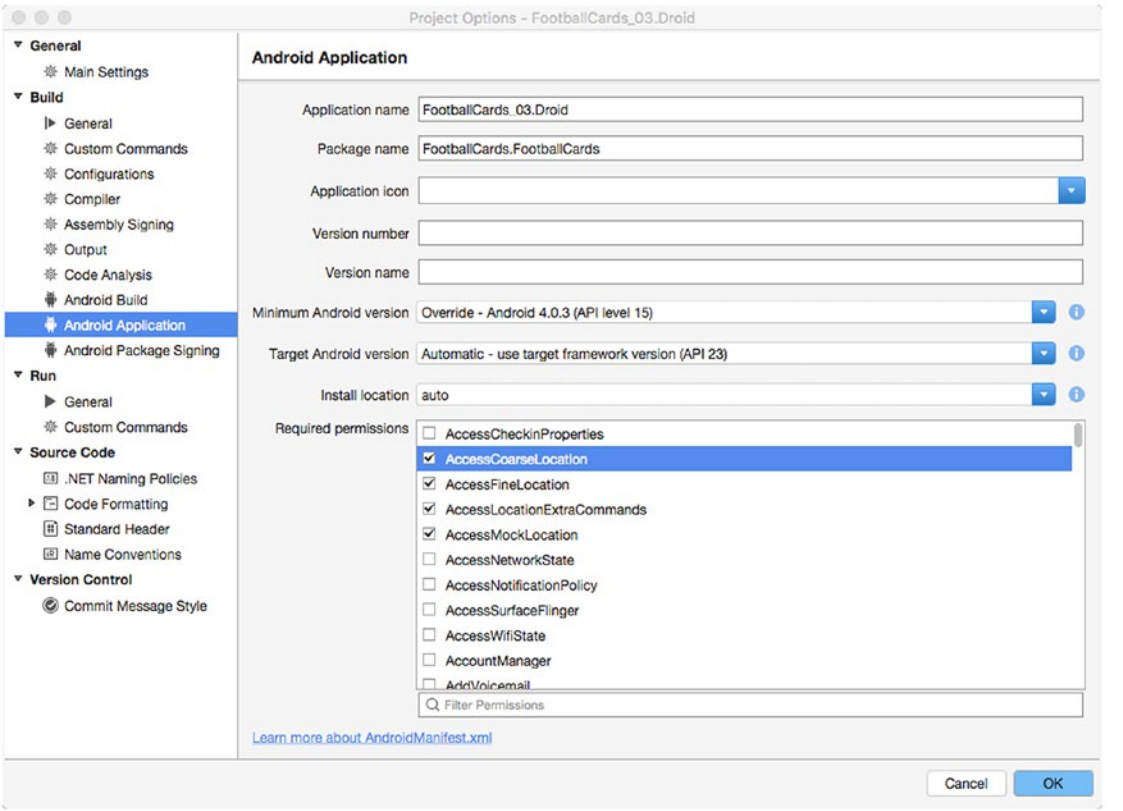


Figure 2-10. *Setting the permissions on Android*

The app will next need an API token, which can be obtained by logging in to your developer account, creating a new app, and from there generating a new API key and adding that into the `AndroidManifest.xml` file. It should look like this:

```
<meta-data android:name="com.google.android.apps.v2.API_KEY"
           android:value="MY_API_KEY"/>
```

Putting Maps into the Apps: iOS

The storyboard needs a new view controller containing a map view. The view controller has to be given the name used when defining the navigation service. An additional button is required for the initial view controller. Make sure it has a name (such as `btnShowMap`) and some suitable text. There is no requirement to link the new button to the new view controller via a segue as it was handled in the view model. See Figure 2-11.

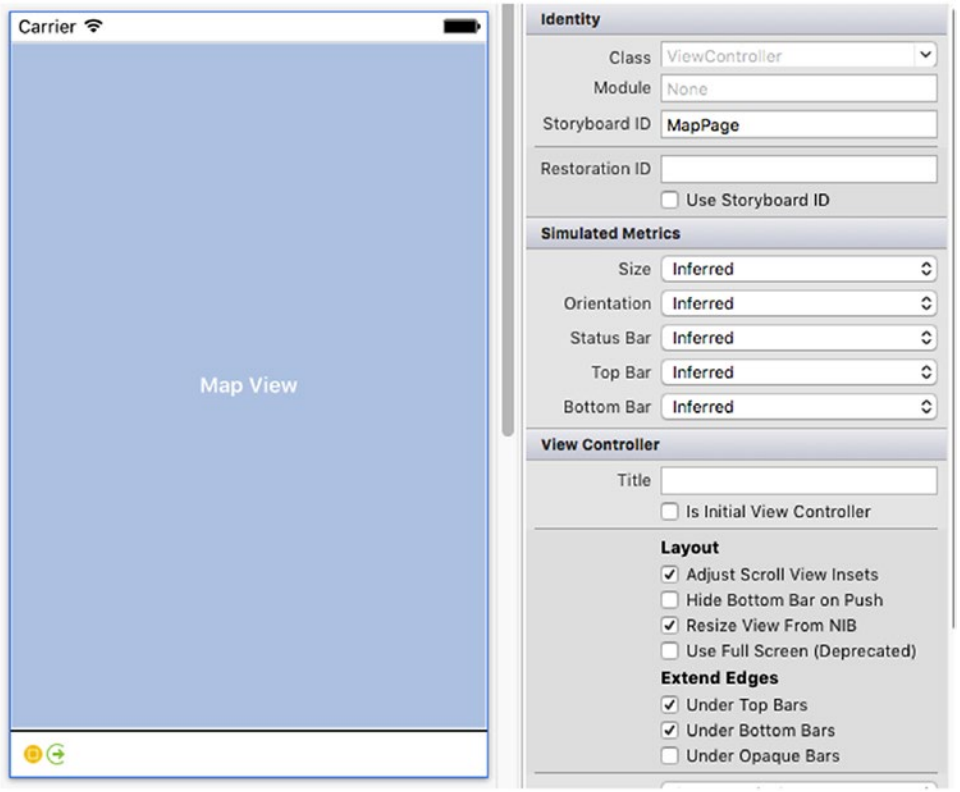


Figure 2-11. *The new ViewController for iOS to handle the maps*

Putting Maps into the Apps: Android

We need to create a new layout to hold the map. Once it has been created, replace the source with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <com.google.android.maps.MapView
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:enabled="true"
                android:clickable="true"
                android:id="@+id/mapView" />
</LinearLayout>
```

Once done, save the layout.

Putting Maps into the Apps: Windows Mobile

We first need to create a blank (or basic) page called `MapPage` (the name we defined when we created the navigation service). Once created, add the `DataContext` to point to the view model (in this case, it is the new `MapViewModel`). See Figure 2-12.

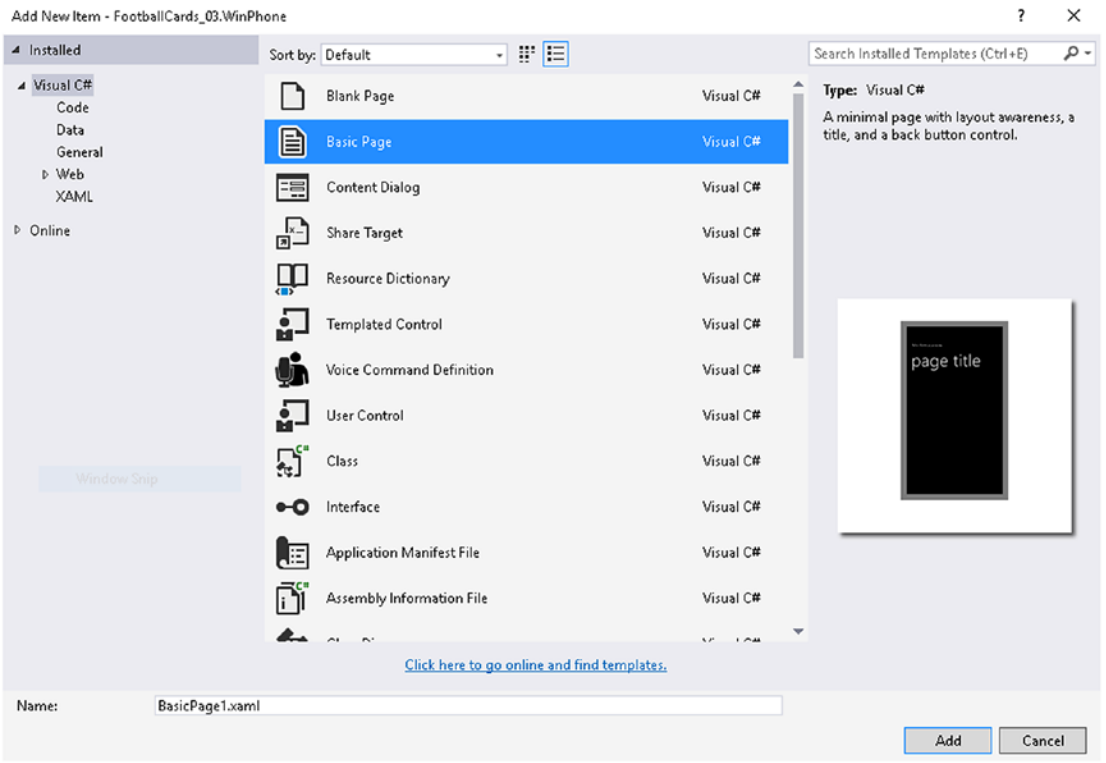


Figure 2-12. *Add new blank page on Windows Mobile*

Once we have the binding in place, we can create the `Map` object from the toolbox and associate a binding to it:

```
<Grid>
        <Maps:MapControl x:Name="mapView" />
</Grid>
```

In the code behind, we also have to create an `async Task` that will attempt to set the map view. Using an async approach is a better approach, as while we will still get the map (so we can see something), once the data is returned everything is in place for the map to zoom or move to the correct location.

```
async Task SetMapView(Geopoint point)
{
    await mapView.TrySetViewAsync(point, 15, 0, 0, Windows.UI.Xaml.Controls.Maps.
    MapAnimationKind.Bow);
}
```

Creating the Location from the View Model

In order for the map to be displayed, we first need some data. This data comes from the `MapViewModel`, which in turn will need to receive it from the top card generated from the shuffle. First, though, we need to link the *Show Map* button to navigate to the new view.

As before, the first step is to create the property the button will call within the `MainViewModel`:

```
public RelayCommand ButtonShowMap
{
    get
    {
         return buttonShowMap ??
          (buttonShowMap = new RelayCommand(
          () =>
             {
                  navigationService.NavigateTo(ViewModelLocator.MapPageKey);
             }));
    }
}
```

This code will simply get the instance within the navigation service of the `MapPage` and navigate to it. The problem is that this is the equivalent of just firing off a new, parameterless instance of a new class.

We can pass in a second parameter to the `NavigateTo` method, which can be sent to the next view model. It makes sense to pass over the GPS coordinates as either a `List<double>`, a `double` array, or as a new class (which will need to be added to the Models directory). We shall send over a `List<double>`, with the first parameter being the latitude.

The code for `NavigateTo` now looks like this:

```
navigationService.NavigateTo(ViewModelLocator.MapPageKey, new List<double>{Latitude,
Longitude});
```

For this to be of any use, the platform code has to take that data and deal with the code (for example, it can be sent back to the new view model, and the bindings on part of the UI then receive that data).

Each platform deals with the data sent from `NavigateTo` differently.

Dealing with Data Passed to the Platforms: Android

Android takes information passed between activities via the Intent. The same applies here, except we're not using `Intent.Extras.Get` as such. For Android, we first need an instance of `NavigationService`:

```
var NavService = (NavigationService)SimpleIoc.Default.GetInstance<INavigationService>();
```

Next, we need to obtain the data being passed in. Remember, this can be an object of any type (from a single `Bool` to a `Dictionary<T, U>` to a `class` and everything in-between). We will need to cast this object to the desired type. This can be performed in one of two ways:

```
var data = NavService.GetAndRemoveParameter<List<double>>(Intent);
```

where the casting is in the `< >` braces, or

```
var data = NavService.GetAndRemoveParameter<object>(Intent)as List<double>;
```

where the object is read into data as a List<double>.

Code-wise, these are both fine and are considered the same.

There is a final step, but as this is the same over all platforms, I'll deal with it after iOS and Windows Phone have been considered.

Dealing with Data Passed to the Platforms: iOS

iOS is easier than Android at obtaining the data passed from NavigateTo:

```
var data = NavigationParameter as List<double>;
```

NavigationParameter can be considered the same as the CommandParameter property, but it is handled by NavigationService.

Dealing with Data Passed to the Platforms: Windows Phone

Similar to Android, the first step is to obtain an instance of NavigationService:

```
var NavService = (NavigationService)SimpleIoc.Default.GetInstance<INavigationService>();
```

We next call the GetAndRemoveParameter method. This looks similar to the Android call, except instead of an Intent, the NavigationContext is used. Again, the object needs to be cast to the correct type:

```
var data = NavService.GetAndRemoveParameter(this.NavigationContext) as List<double>;
```

Dealing with Data Passed to the Platforms: Final Step

The final step in obtaining the data from NavigateTo is really just good practice; we check to ensure that data is not null before acting on the data:

```
if (data != null)
{
    // do something
}
```

If the data has been incorrectly cast or no data has actually been passed, then trying to do anything with our variable data will result in the app's crashing.

If you prefer, the if construct can be replaced with a try/catch using the NullExceptionError in the catch.

Plotting the Data

We have a final piece for the map jigsaw—we need to actually plot something on the map views. We aren't able to bind directly, but we can create the platform location object by accessing the Lat/Long properties from the view model.

With iOS, we create the `ViewController`, ensure the storyboard references it, and include the following to generate the map:

```
void GenerateMap()
{
      mapView.ZoomEnabled = mapView.ScrollEnabled = mapView.UserInteractionEnabled = true;
      mapView.MapType = MKMapType.Standard;
      mapView.Region = new MKCoordinateRegion(
         new CLLocationCoordinate2D(ViewModel.Latitude, ViewModel.Longitude),
         new MKCoordinateSpan(.5, .5));
}
```

Creating the map for Android is slightly more involved. We need to use NuGet to install `Xamarin.GooglePlayServices.Maps`, which allows access to the map libraries, and we also need create the `Activity`.

Conclusion

Hopefully, you will have seen in this chapter that using an MVVM framework can greatly improve the speed at which you can develop an app as well as debug one when things go wrong. Many consider MVVM to be a painful process, but if you adopt the way things work and interoperate, you can create code very quickly and, moreover, not have to replicate it in each project.

CHAPTER 3

Inversion of Control (IoC) & Messaging

For any form of MVVM system to work within your Xamarin application, it must rely on a pattern known as inversion of control. Without some form of IoC in place, the PCL part of the application will just act as a library. By definition, a library can only supply information—it cannot inject information.

Along with IoC comes dependency injection (or DI, as it is often referred to). Dependency injection is a technique often used to have the library injection information from another service (typically hosted on the platform). It then relies upon that information to perform an action (such as finding the current locale setting for localization of text information).

Messaging (on the other hand) is a simple method of passing information between the view models, typically without the view's being involved and without having to register an interface and implementation with SimpleIoc.

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_3

IoC Basics

Let's consider how a standard application works (Figure 3-1).

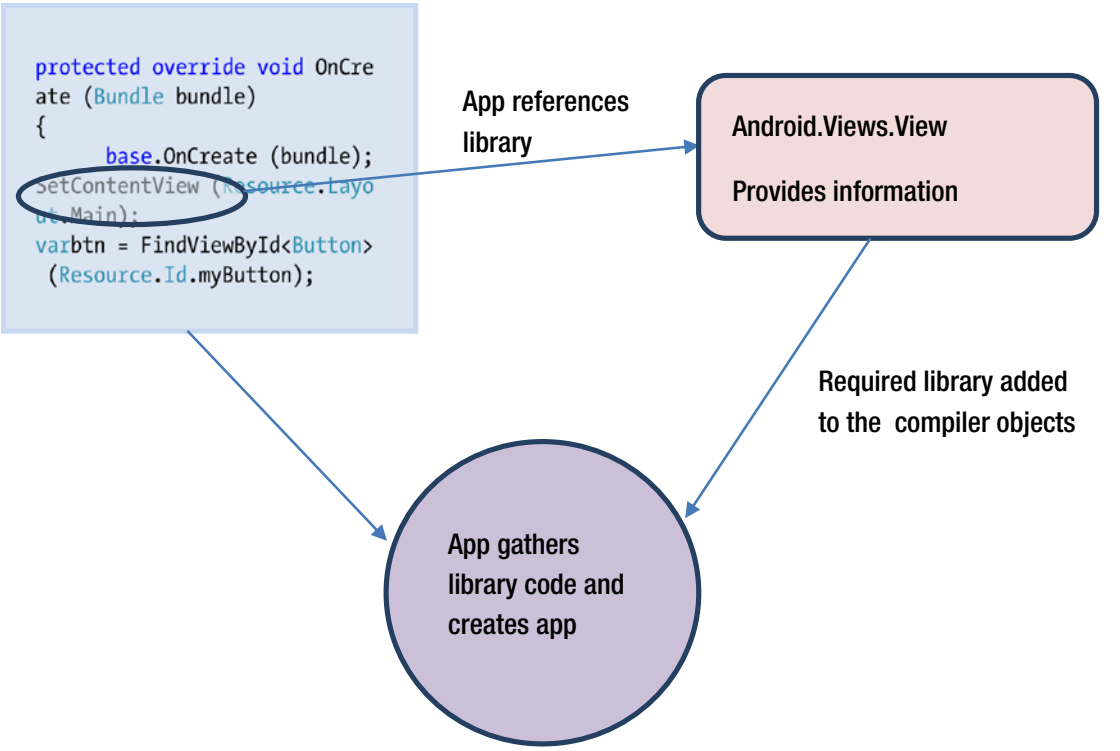


Figure 3-1.

Our application links to a system of libraries. The libraries SUPPLY the information the compiler requires to generate the binary. This is as true for Xamarin apps as it is for Java, ObjC, Swift, and any other language. To reduce the binary size, some application types use a system of dynamic linkable libraries (the often seen dll file). These libraries are still used in the same way as a library built into the binary and only supply information.

Without the ability to use injection, the PCL works in the same way as any other library; the big difference is that it is not a "black box"; you are able to see the workings of the library you are building. Of course, you can argue that you're still linking against the .NET standard libraries and as such that *is* a black box, but let's not go there.

IoC

Inversion of control means that the library takes control over the application (think of it as the tail wagging the dog). IoC relies on a system of services being provided (typically referenced via an interface).

MVVM Light uses this inversion system in a number of places:

1. Navigation
2. Messaging
3. Data binding
4. Alerts

We came across navigation in the last chapter; we create the view model, set up the navigation service on the platforms, and then, within the PCL, call the navigation service to navigate to the new activity/view controller/page. The service provides for two-way movement (the return to the previous view on mobile being provided by the `GoBack()` method, which essentially "pops" the current view controller and displays the new current view).

■ **Note** An example of this can be found in Chapter3/Navigation.

Data binding for the UI has also been seen. Here, the application references the library as it would any other library. However, when the property the UI object is bound to changes, the standard .NET `PropertyChanged` (or `PropertyChanging`) event is triggered. This "pushes" the update to the UI.

Alerts

As with navigation, it can be argued that the alert is part of the platform and not part of the PCL, and, on the face of it, this is true. That said, if an error occurs within the PCL, there has to be a way to alert the user. While it is certainly possible for the PCL to inject an alert onto the platform and have the platform respond, typically the response to the alert is required within the PCL, so there would need to be some form of "on the fly" binding to the object—an object that only exists for a fraction of the lifetime of the application.

Let's look at how this works for MVVM Light.

■ **Note** The full source for this example can be found in Chapter3/Alerts.

The alert system uses the `IDialogService` interface, which needs to be registered on each platform in the same way as you would the navigation service, as follows:

```
SimpleIoc.Default.Register<IDialogService, DialogService>();
```

We don't need to define the `DialogService` or the interface, as they are already provided by MVVM Light.

Once this has been registered, accessing the alert is performed via the `ViewController`:

```
public RelayCommand BtnAlert
{
    get {
            return btnAlert ??
            (btnAlert = new RelayCommand(
                 async () =>
                 {
```

```
                    var dialog = ServiceLocator.Current.GetInstance<IDialogService>();
                    await dialog.ShowError("A message from the VC", "Hello world!", "OK", null);
                }));
        }
}
```

Clicking on the button brings up the dialog native to the platform, as seen in Figures 3-2a and 3-2b.

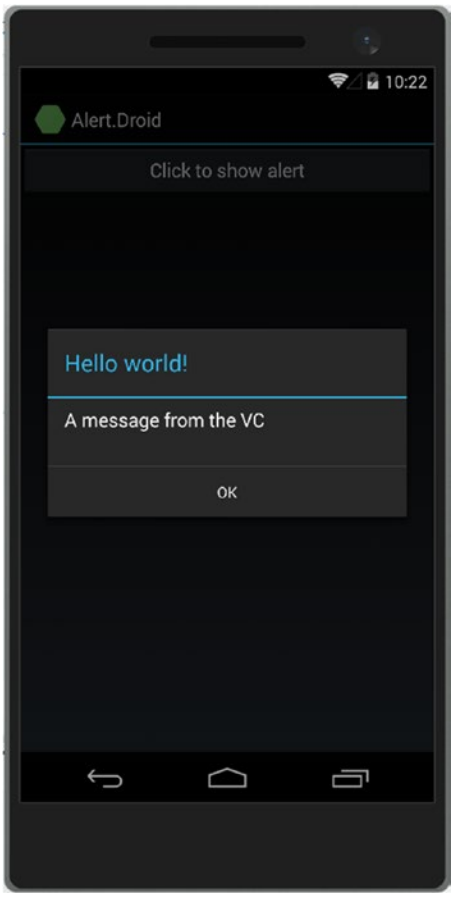


Figure 3-2a. *Android alert*

Figure 3-2b. *Windows Phone alert*

This alert is the simplest, and the only action possible dismisses the alert. To be able to make a decision from the dialog, we need to use the fourth parameter (currently null). This fourth parameter is an action taken after the dialog has been cleared.

■ **Note** Chapter3/Alert2 demonstrates using the fourth parameter.

In Alert2, the text in the button changes after the dialog box has been dismissed using the following code (Figures 3-3a and 3-3b):

```
public RelayCommand BtnAlert
{
     get
     {
         return btnAlert ??
 (btnAlert = new RelayCommand(
             async () =>
             {
                  var dialog = ServiceLocator.Current.GetInstance<IDialogService>();
```

```
                await dialog.ShowError("A message from the VC", "Hello world!", "OK", () =>
                    {
                        ButtonText = "You clicked me";
                    });
            }));
        }
}
```

Figure 3-3a. *Alert from view model*

Figure 3-3b. *Button text changed after the alert has been dismissed*

You may have by now noticed a limitation on the default alert box: you only get the *OK* button. This is very limiting—what if you want the alert to ask if the user wants to perform a process or cancel?

For this, we can use `dialog.ShowMessage`. This comes with a number of overloads:

`ShowMessage(string Title, string Message);` (has 1 button)

`ShowMessage(string Title, string Message, string ButtonText, Action afterDismiss);` (one button)

`ShowMessage(string Title, string Message, string ButtonConfirmText, string ButtonCancel Text, Action afterDismiss);` (two buttons)

The two-button overload of `ShowMessage` returns a Bool. If the *Cancel* button is clicked, the Bool returns false.

■ **Note** Chapter3/Alert3 demonstrates the two-button overload of `ShowMessage`.

In Alert3, we implement the two-button overload:

```
public RelayCommand BtnAlert
{
    get
    {
         return btnAlert ??
 (btnAlert = new RelayCommand(
         async () =>
         {
              var dialog = ServiceLocator.Current.GetInstance<IDialogService>();
              var result = await dialog.ShowMessage("Do you want to take the blue pill or
              red pill?",
                           "Reality time", "Red pill", "Blue pill", async(r) =>
              {
                    await dialog.ShowMessage(string.Format("You picked the {0} pill", r ?
                    "red" : "blue"), "Yummy");
              });
              if (result)
              {
                    await dialog.ShowMessage("Welcome Nero...", "Message from the Matrix");
              }
         }));
     }
}
```

■ **Caution** You need to take care when using this method as the code is waiting on the result. However, as it is an asynchronous call to the `DialogService`, the UI will continue. Therefore, if anything is reliant on the outcome of the choice within the UI, you will need to make allowance for this.

Using the Built-in IoC

The inversion of control system that comes with MVVM Light is aptly called SimpleIoC. This is a very lightweight IoC container and can be replaced by any other IoC library (such as ninject) should you need something less simple.

For the majority of the time, you will register whatever you need within the `ViewModelLocator.cs` file (the obvious exception to this is the navigation service, which is on the platform level).

SimpleIoC allows you to register both the view models and any other interfaces that may need passing to the view models like this:

```
SimpleIoc.Default.Register<IMyInterface>(()=> new MyInterface());
```

or

```
SimpleIoc.Default.Register<IMyInterface, MyInterface>();
```

When you then need to access the registered interface, it can be recalled using the following:

```
SimpleIoc.Default.GetInstance<IMyInterface>();
```

This is the simplest way in which you can use SimpleIoC. You can, however, have multiple instances of the interface:

```
SimpleIoc.Default.Register<IMyInterface>(()=> new MyInterface(), "InstanceName");
```

`InstanceName` can be whatever name you want to give it. To retrieve that instance of the interface, you would use the following:

```
SimpleIoc.Default.GetInstance<IMyInterface>("InstanceName");
```

If SimpleIoC is unable to find an instance with our `InstanceName`, it will return the default interface instance.

Another handy shortcut SimpleIoC provides is the ability to instantiate the instance upon creation:

```
SimpleIoc.Default.Register<IMyInterface>(()=> new MyInterface(), true);
```

Once we have registered with SimpleIoC, we can also pass the interface to a view model in the constructor (as has been seen with the `INavigationService`):

```
IMyInterface myInterface;
INavigationService navigationService;
public class SomeViewModel(IMyInterface intface, INavigationService navService)
{
    myInterface = intface;
    navigationService = navService;
}
```

Dependency Injection

Dependency injection is a term you will hear quite frequently in association with IoC. As the name suggests, you are injecting information from one place to another. Within IoC, this will typically be from the platform to the library. The library will have no knowledge of, say, a file path (for example, when creating an SQLite database, one of the first operations to occur is to create the database, which is stored in a file, and each mobile device stores the database in a different location). When it comes to writing to the database, the database needs to know where that file is, so the location is injected in. It is not passed in when the database code is called, as you would a normal method.

DI is used greatly within any form of IoC application (including if you are using Xamarin Forms).

Diagrammatically, DI is often represented as seen in Figure 3-4.

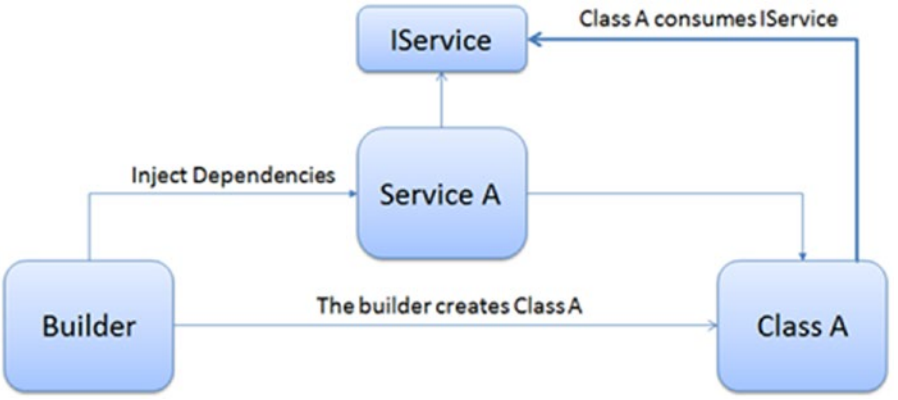


Figure 3-4. *A simple representation of how DI works*

Using DI Within an MVVM Light Application

■ **Note** Chapter3/Source/DepInjection has an example of DI in action.

DI is fairly simple to use within an MVVM Light application. Let's take the following example and see what is going on.

```
public class Class1 : IClass1
{
    public IInterface1 Interface1 { get; set; }

    public IInterface2 Interface2 { get; set; }

    public IInterface3<T> Interface3 { get; set; }

    public Class1()
    {
    } // base constructor

    public Class1(IInterface1 iface1)
    {
        Interface1 = iface1;
    }

    public Class1(IInterface1 iface1, IInterface2 iface2)
    {
        Interface1 = iface1;
        Interface2 = iface2;
    }
```

```
    [PreferedConstructor]
    public Class1(IInterface1 iface1, IInterface2 iface2, IInterface3<T> iface3)
    {
        Interface1 = iface1;
        Interface2 = iface2;
        Interface3 = iface3;
    }
}
```

This is a simple class. The `[PreferredConstructor]` tells the compiler that this is the higher priority constructor in the code (in other words, the most important and likely to be used).

Each interface looks like this:

```
public interface IInterface1
{
     string MyString1 { get; set; }
}
```

Once these have been set up, as with everything else in MVVM Light, they are registered with SimpleIoC:

```
SimpleIoc.Default.Register<IInterface1, myFirstImplementation>();
SimpleIoc.Default.Register<IInterface2, mySecondImplementation>();
SimpleIoc.Default.Register<IInterface3<int>, myThirdImplementation<int>>();
SimpleIoc.Default.Register<IClass1, Class1>();
```

To then use DI, you would use within your code the following:

```
SimpleIoc.Default.GetInstance<Class3>();
```

There is a downside, however, which is that, out of the box, you are unable to use an implementation with a key (such as the one called `InstanceName` used). This is a limitation of SimpleIoC.

There are various solutions to help you work around this limitation available online. The following is one I have used quite a few times:

```
public static class SimpleIOCExtension
{
    public static T GetInstanceDI<t>(this SimpleIoc simpleIoc, string key = "")
    where T : class
    {
        var ctors = typeof(T).GetConstructors(BindingFlags.Instance | BindingFlags.Public);

        if (!ctors.Any())
            throw new Exception("No constructor found");

        var ctor = ctors.OrderBy(d => d.GetParameters().Count()).Last();

        var param = ctor.GetParameters().Select(d => simpleIoc.GetInstance(d.ParameterType,
        key)).ToArray();

        return ctor.Invoke(param) as T;
    }
}
```

This works by getting the constructors for the requested class via reflection. It then takes the one with the most parameters and obtains the instance from SimpleIoC. The constructor is then invoked with the parameters. Quick and easy, as all good code should be!

Messaging

MVVM Light contains a built-in messaging system. It can be used for passing information between view models as well as for passing information back from the view model to the view, and also for passing messages to any other part of the application as required.

It works as seen in Figure 3-5.

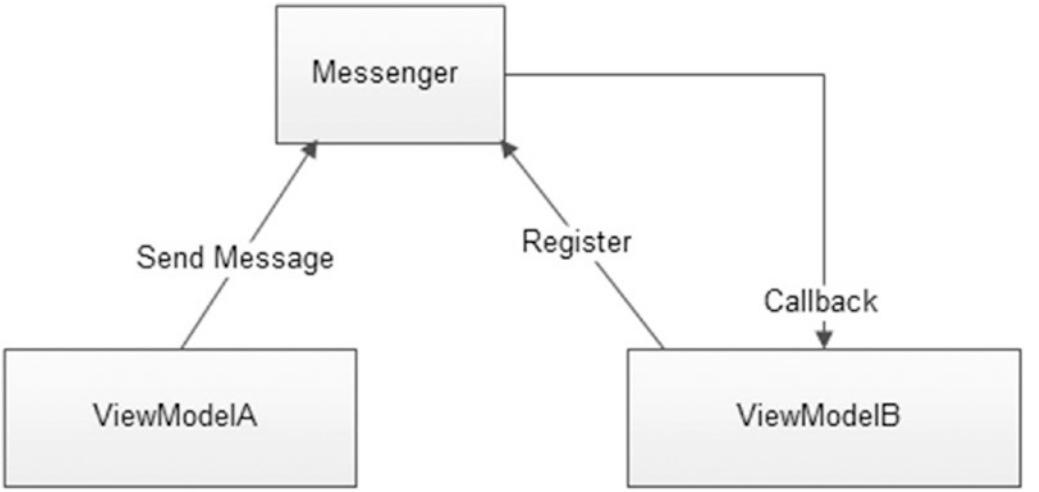


Figure 3-5. *The MVVM Light Messenger class*

The class provides both the interface and the implementation of it. Unlike other services, this does not need to be registered with SimpleIoC within the `ViewModelLocator` class. This is because each view model inherits `ViewModelBase`. This provides an instance of Messenger via the provided `MessengerInstance`.

There are three important methods the class provides: `Register`, `Unregister`, and `Send`.

Register

```
void Register<TMessage>(object recipient, object token, bool receiveMessages,
Action<TMessage>action);
```

This registers a recipient for a message of type `TMessage`. This type can be any type (`int`, `string`, `class`, or any other type).

Parameters

recipient: this is the object receiving the message. This can be set as the current object (using `this`) or any other specified object.

token: a keyword that the recipient "listens" out for. This means that if you have three objects listening for the token "foo," and two of these objects also listen for "bar," and we broadcast the token "bar," then the two objects listening for "bar" will receive the message.

receiveMessages: if this is `true`, the recipient also receives message types that inherit from `TMessage`. For example, if arms and legs implement the interface `IBody`, then registering with the type `IBody` and setting `receiveMessages` to `true` will result in the recipient getting arm and leg messages too.

action: The action that will be executed when the message is sent from the `Send` message.

Unregister

```
void Unregister<TMessage>(object recipient, object token, Action<TMessage>action);
```

Unregisters a recipient for the type of message `TMessage`. The parameters are the same as for `Register`.

The use of `Unregister` is important to the performance of your application. Consider the following. My application has 50 view models 30 register to message "A" and 30 register to "B". I then broadcast message "A". The 30 models listening for "A" all wake up and try and do something. Depending on what the something is depends on how long it will take to execute. In any case, the more classes registered, the more time it will take to process the message. If `receivesMessages` is true, this greatly increases the amount of data being passed to the recipient.

It is therefore important that, when possible, the recipient should unregister for messages.

Send

```
void Send<TMessage>(TMessage message, object token);
```

This sends messages to those objects registered.

Parameters

message: The instance of type `TMessage` to be sent
token: This is the token the registered class listens out for

Other Methods in the Messenger Class

The messenger class also provides three other methods: `Default`, `Reset`, and `Cleanup`.

Default

Provides the default instance of `Messenger`. This is a `static` method.

Reset

```
static void Reset();
```

Deletes the default instance of `Messenger`. If the default property of `Messenger` is used again, it creates a new instance of `Messenger`.

Cleanup

```
void Cleanup();
```

This scans the recipient list for dead recipients. When registered, the recipients are stored as `WeakReferences` so they can be claimed by the garbage collector. The `Cleanup` method removes the recipients that have been collected.

Other Important Classes

There are a number of other important classes provided with the messenger service:

MessageBase: Base class for all message types used by `Messenger` for sending messages. There are only two properties (of type `object`) that exist in this class: `Sender` and `Target`. These can be set (or not) in the constructor and exist only for sending indications to recipients.

NotificationMessage: Used for sending `string` message notifications to recipient(s).

NotificationMessage<T>: Used for sending `string` message notifications and a generic value `<T>` to recipient(s).

NotificationMessageAction: Used for sending `string` message notifications with a callback action. After the recipient processes the message, it can execute the callback.

NotificationMessageAction<TCallbackParameter>: Used for sending `string` message notifications with a callback with a parameter of type `TCallbackParameter`. When the recipient processes the message, it can execute the callback with the parameter.

PropertyChangedMessage<T>: Used for sending the property name along with both the old and new values.

GenericMessage<T> Used for sending generic values to a recipient. Recipient can access the generic value using the `Content` property.

Using the Messenger Class in action

As with most services and techniques, to better understand how the `Messenger` class works, it is best to see some code in action.

■ **Note** Chapter3/Messages contains the source for the following snippets.

To start with, data is being passed between the two view models. In order to facilitate this, a model class is created:

```
namespace Messages.Models
{
    public class MessageData : MessageBase
    {
        public MessageData(string d1, string d2, string d3, string d4)
        {
            Data1 = d1;
            Data2 = d2;
            Data3 = d3;
            Data4 = d4;
        }
```

```
        public string Data1 { get; private set; }
        public string Data2 { get; private set; }
        public string Data3 { get; private set; }
        public string Data4 { get; private set; }
    }
}
```

Once we have this, we create a `RelayCommand` to be bound to on the platform. This is in the `MainViewModel` file:

```
RelayCommand btnClick;
public RelayCommand BtnClick
{
    get
    {
        return btnClick ??
 (btnClick = new RelayCommand(
 () =>
        {
             // create the instance of the model
             var dataModel = new MessageData("This is line 1", "Line 2 contains this",
             "MVVMLight", "Xamarin rocks!");

             // Send the message. We are going to specify a key. We don't need it for this
             code
             // but if we had more view models and didn’t want them all to listen for this
             broadcast,
             // the key will tell those view models who aren't listening for this key to
             ignore it.
             Messenger.Default.Send(dataModel, "SelectData");

              // go to the next viewmodel
              navService.NavigateTo(ViewModelLocator.SecondViewKey);
          }));
     }
}
```

Finally, we need to wire this up into the receiving view model. This is achieved by placing the `Messenger.Default.Register` within the constructor (see Figures 3-6a and 3-6b):

```
public SecondViewModel()
        {
            Messenger.Default.Register<NotificationMessage<MessageData>>(this, (message) =>
            {
                // Gets the message object.
                var data = message.Content;

                // Checks the associated action.
                switch (message.Notification)
                {
                    case "SelectData":
```

```
                    TextData1 = data.Data1;
                    TextData2 = data.Data2;
                    TextData3 = data.Data3;
                    TextData4 = data.Data4;
                    break;
                default:
                    break;
            }
        });
    }
```

Figure 3-6a. *Click to send the messages*

Figure 3-6b. *Results of sending the message*

Conclusion

We've covered inversion of control and the messenger system used by MVVM Light. They are both powerful and simple to use. In the next chapter, we'll start to add functionality to a simple MVVM Light application by using the facilities each platform provides.

CHAPTER 4

Adding Functionality

Your mobile phone is a powerful piece of equipment. Not only does it contain an extremely fast processor, but it also has the ability to determine your location, access external services that can be used to extend your applications' functionality, make calls (obviously), and store data as well as a pile of other useful facilities.

In this chapter, we shall look at the integration of a simple SQLite database within your application as well as at a webservice and using some of the common phone functions and functionality (such as accessing the address book and accessing files).

Adding a Database

The real beauty of using a database within MVVM Light is that with the exception of providing the device platform, absolutely everything else can be performed within the PCL. The packages used are available from NuGet.

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_4

For the Microsoft platforms, you will need to do the following (this can be performed before even starting a project)

1. Select Tools ➤ Extensions and updates (Figure 4-1).

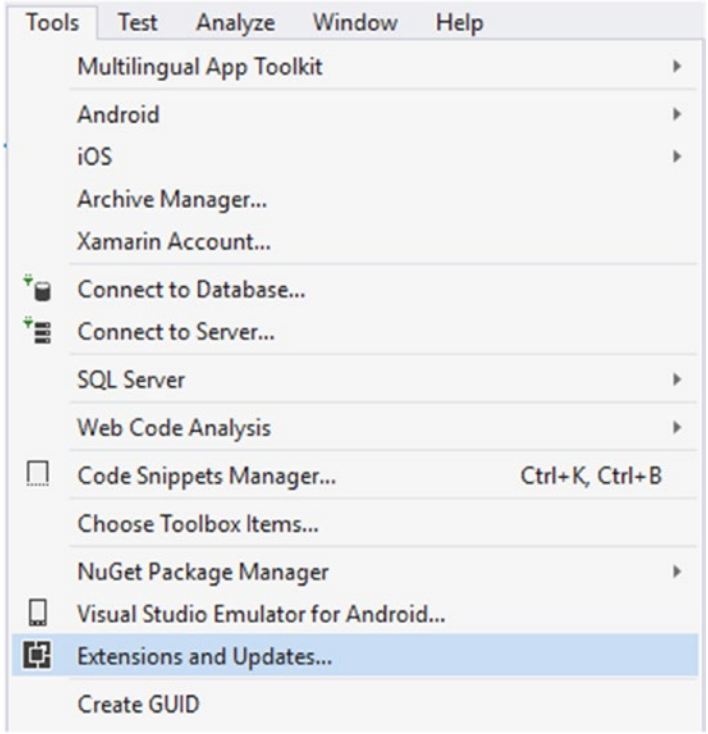


Figure 4-1. *Selecting the Extensions & Updates menu option in Visual Studio 2015*

2. On the left-hand-side menu, select Online. A new menu will appear under it. Select the Visual Studio Gallery option (Figure 4-2).

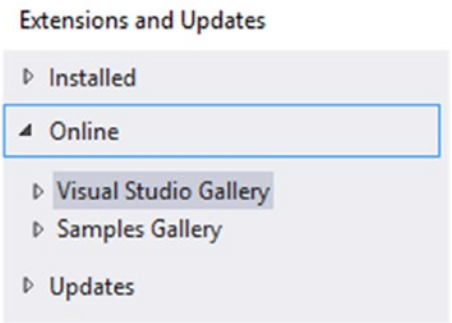


Figure 4-2. *Selecting the Online gallery menu option*

3. In the search box on the right-hand side, type *SQLite*. A number of packages will appear in the middle once found (Figure 4-3).

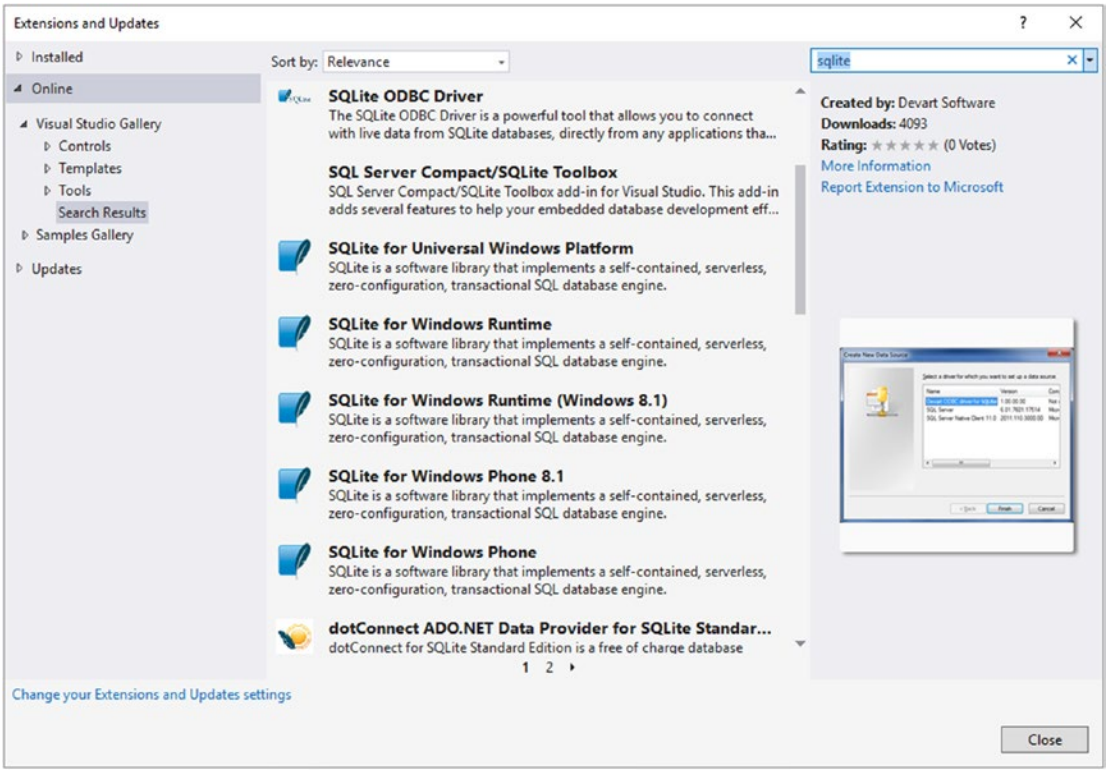


Figure 4-3. *SQLite packages available through the extension installer*

4. For Windows 10 (including mobile), click on the Universal Windows Platform (known as UWP). Clicking *Download* will start the installation (Figure 4-4).

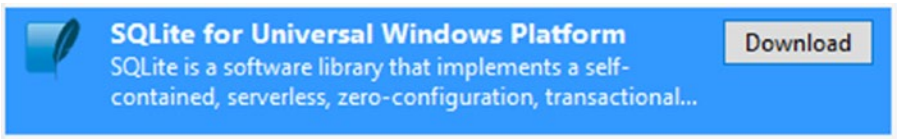


Figure 4-4. *UWP SQLite package selected*

5. For Windows 8.1 Phone support, click on the Windows Phone 8.1 option. As with the UWP package, clicking *Download* will start the installation (Figure 4-5).

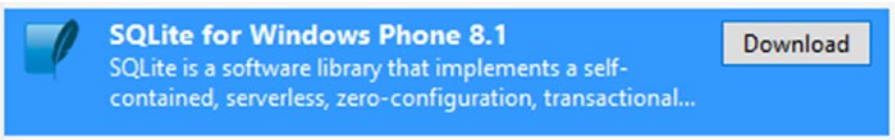


Figure 4-5. *Windows Phone 8.1 SQLite package selected for installation*

6. Once the package has downloaded, you will be presented with a license window. To accept the license, click *Install*. The installer installs the package globally to your system so it is available to all applications requiring it (Figure 4-6).

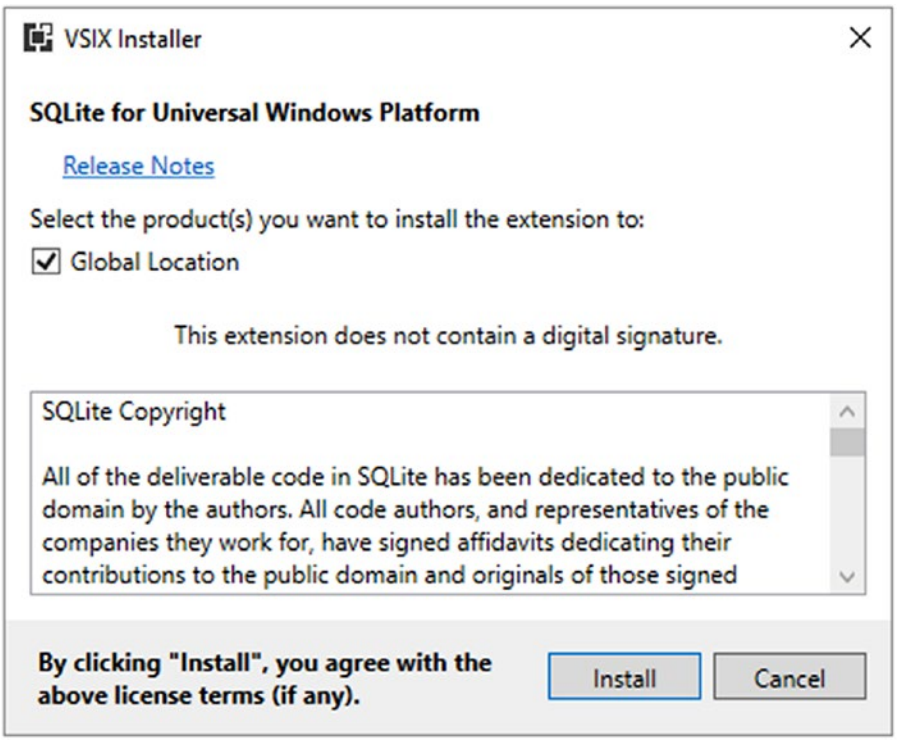


Figure 4-6. *Acceptance of the SQLite package license*

7. Once installed, you will be presented with a message prompting you to restart Visual Studio. This appears at the bottom of the window containing the search results. If you are installing both SQLite packages, it is safe to restart after both have installed (Figure 4-7).

Figure 4-7. *Restart message once installed*

You now have SQLite installed on your PC. You will still need to install the subpackages from NuGet.

■ **Note** Be aware that there are different NuGet packages available for SQLite. While they may be all based on the same source code, it is safer to stick with the SQLite.Net-PCL 3.1.x ones as these are also the ones packaged for the Windows platforms.

Creating Your SQLite Application

Let's begin by creating a project in the same way as before and installing the MVVM Light framework into each. Once we have that done, add the `SQLite.NET-PCL` package to the PCL and then to the platform subpackages (except for Windows).

Adding the SQLite References to Your Windows Projects

In a similar way to adding a NuGet package, you can add a reference to an existing .NET assembly that is installed on your computer. To do this, follow these steps:

1. Select References and click on the Add Reference option (Figure 4-8).

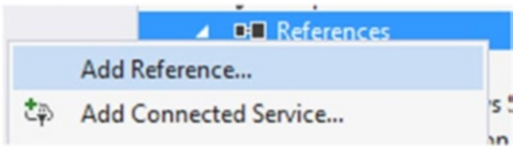


Figure 4-8. *Adding a reference to an existing Windows package*

2. On the left side of the Window (Figure 4-9 shows it for Windows Phone 8.1, but it is almost identical for UWP), select the Windows Phone 8.1 ➤ Extensions option. In the middle, you will be shown the extensions available. Click on the SQLite option (ensure that it is ticked) and then press the *OK* button. The SQLite assembly references will be added to your project. Note that the version number should be similar to the NuGet version installed in the PCL.

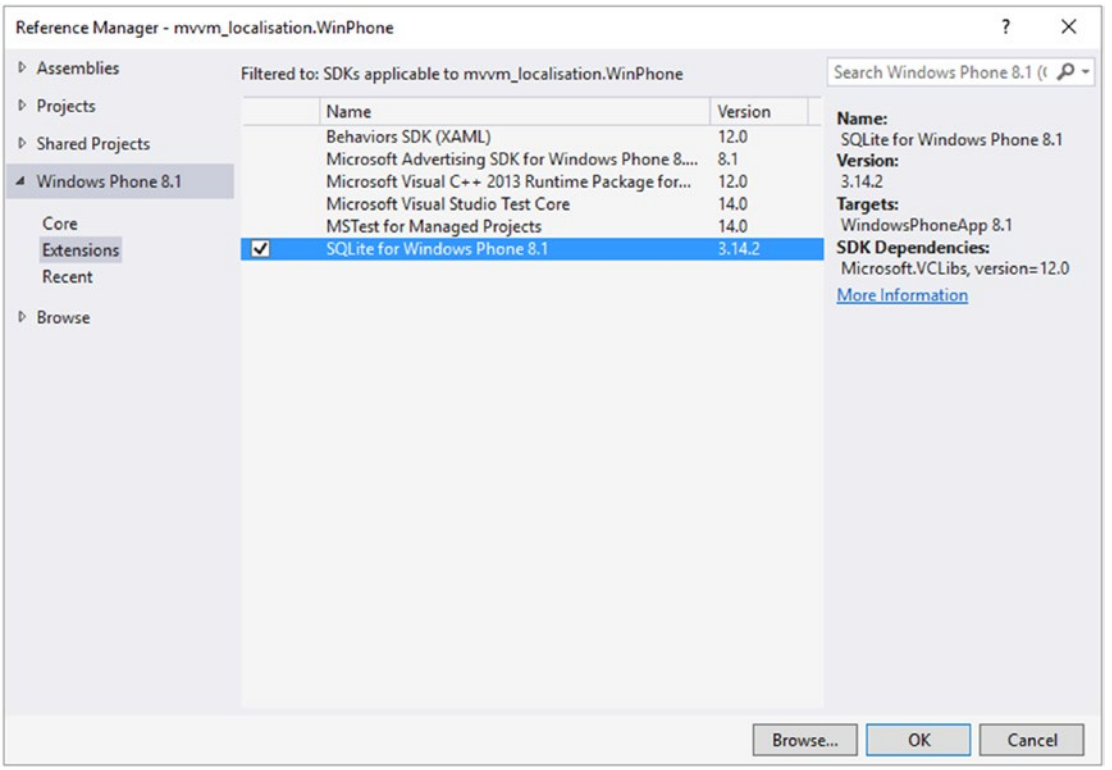


Figure 4-9. *Adding the SQLite assembly reference for Windows Phone 8.1 (there is a similar package for UWP)*

■ **Note** It is a requirement now for Android that you install a package called sqlite3 on the platform (compiled with v110_xp) (or something similar to this) into the PCL. If you don't, the first call to the database will kill the app irrespective of the platform. See Figure 4-10 for what it looks like.

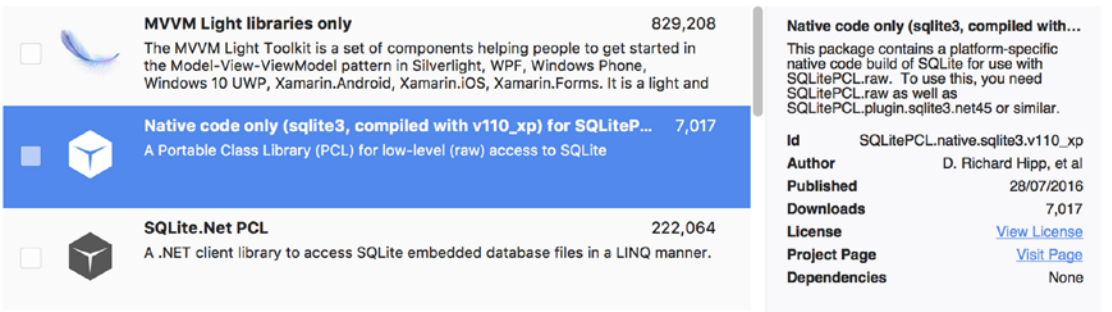


Figure 4-10. *SQLite3 native package for .NET*

Now that the pre-requisite files have been added, we can start creating the initial database.

Models, Interfaces, and Helpers

Key to any database are the data models and the helper methods associated with them. Typically, the helpers will do the data interrogation and the data insertion into the database. The only piece of information that the database will need is the location of the database file created, which is retrieved via IoC (which we covered in Chapter 3). The PCL cannot create the database file, but is able to read/write to and from it.

Let's create a simple model for the storing of names, addresses, and telephone numbers, as follows:

```
using System;
using SQLite.Net.Attributes;

namespace SQLiteExample
{
    public class PersonalInfo
    {
        [PrimaryKey]
        public string id { get; set;}
        public string FirstName { get; set;}
        public string FamilyName { get; set;}
        public string LandlinePhone { get; set;}
        public string MobilePhone { get; set;}
        public string AddressLineOne { get; set;}
        public string AddressLineTwo { get; set;}
        public string PostCode { get; set;}
        public DateTime LastUpdated { get; set;}
    }
}
```

Nothing earth-shattering about that. Next, two other model classes for their pets and hobbies are added (these are in the SQLite source project). Both of these other models start like this:

```
[PrimaryKey]
public string id { get; set; }
public string ParentId { get; set; }
```

This sets up the relationship between the primary table (`PersonalInfo`) and the other tables (`Hobbies` and `Pets`). It is known as a one-to-many relationship (one parent that can have any number of children). The children are linked back via the `ParentId` property.

While there is nothing wrong with having the view models do the insert/update/reads for each data point, it becomes confusing over time and will become difficult to maintain. To circumnavigate this, it is customary to create a helper class to do the legwork for you.

Let's look at a simple way to do this. It may look complex, but it's simple enough once you understand a bit about generic programming.

Generic Programming in a Nutshell

C# is known as an object-oriented programming language. In other words, at the lowest possible level, everything is just an object with some properties associated with it (more or less). An object can be anything—either included with the language (such as the `string` object) or something we have created (such as our `Pets` model). As far as the language is concerned, it's an object.

Often, we can use this to our advantage. Say we want to write a class that will output an object via `StreamWriter`. We can do this one of two ways. The first is to write a class for each different object that we want to output to `StreamWriter`. This is long winded and convoluted and prone to error. It also goes against the good programming principle of code reuse. The second way is to tell the writer that it is not going to be outputting, say, `Pets` and `PersonalInfo`, but rather some generic object of type `T`. As we are only outputting a single object, we only need to write the code once.

Generics is a very long subject and not really within the scope of this book; suffice to say that there are many excellent texts and websites covering the topic in much greater depth than I can here.

The clever part comes from the compiler. If you think about it, how does the compiler know what the possible objects are? The simple answer is that it doesn't. However, it is able to "read" the pre-linked code to see what calls the `StreamWriter` class in question and the types that will make use of it. The compiler then writes its own code for those possible types and links it all together, and from that we get the final binary.

This type `T` is extremely useful, but to use it within our models, we need to create an interface, `IInterface`. This will contain a "link" to each data model via the `id` property:

```
public interface IInterface
{
    public string id { get; }
}
```

We then add an interface reference to each model:

```
public class PersonalInfo : IInterface
```

We can now use the helper with generics. Let's work through the basics of the helper class:

```
SQLiteConnection connection;
public SqLiteRepository(ISqLiteConnectionFactory connectionFactory)
{
     connection = connectionFactory.GetConnection();
     CreateTables();
}
```

We first establish the database connection. This is provided through an interface that uses IoC to obtain the connection string from the platform. We then create the tables in the database. If they already exist, they are not created again.

Next, we have a very simple insert or replace method to save data. The name "insert or replace" does exactly what it says it will do. If there is a row that matches the primary key, the row is replaced with the new data. If it doesn't exist, the data is inserted.

```
public void SaveData<T>(T toStore)
{
    connection.InsertOrReplace(toStore);
}
```

As well as saving a row into the database, we need some method to obtain data from the database. Remember how we said we could use type T for our model classes? Here's where it really comes into its own:

```
public T GetData<T, TU, TV>(string para1, TU val1, string para2, TV val2) where T : class, new()
{
     var sql = string.Format("SELECT * FROM {0} WHERE {1}=? AND {2}=?", GetName(typeof(T).
     ToString()), para1, para2);
      var list = connection.Query<T>(sql, val1, val2);
      return list != null ? list.FirstOrDefault() : default(T);
}
```

Let's break this down to understand it:

```
public T GetData<T, TU, TV>(string para1, TU val1, string para2, TV val2) where T : class, new()
```

This method is going to return something of type T and take three generic types as parameters: T, TU, and TV. In C#, we have to tell the compiler that we're using generic types, and these are always in <> braces before the parameter list. The parameter list is the same as you would normally have, except that we have TV and TU as types before the identifier names. This leaves the question as to what T is.

At the end of the line, we have `where T : class, new()`. Here, we are telling the compiler that T is of type `class` and that it must have a default constructor. Note that we don't tell the compiler which class, just that it is one.

```
var sql = string.Format("SELECT * FROM {0} WHERE {1}=? AND {2}=?", GetName(typeof(T).
ToString()), para1, para2);
```

We create a string containing our SQL query. The `GetName` method returns the name of just the type instead of the full version. For example, our `Pets` class will have a full name of `SQLite.SQLiteExample.Pets`. However, we only created a table called `Pets` (which is from the model `Pets`). When we pass in T it will have the full name, so we need to reduce it to the name the database recognizes.

```
var list = connection.Query<T>(sql, val1, val2);
```

This takes the connection to the database and creates a query that will return all objects of type T, with `val1` and `val2` being the parameters that replace the ? in the SQL string. The result is returned into the variable `list`.

```
return list != null ? list.FirstOrDefault() : default(T);
```

Finally, we return `FirstOrDefault` from the returned values in a `list` (assuming there are results); otherwise, return the default value for T (this is the created instance made when the class was first called).

You should now realize the power and (coding) speed of using generic types. The rest of the class is made up in a similar way.

Wiring the Database Up for the View Models

With the helper, models, and various interface classes in place, we now need to be able to use the database within the view models.

In order to do this, we need to add some code to the `ViewModelLocator.cs` file:

```
public IRepository Repository
{
    get
    {
        return ServiceLocator.Current.GetInstance<IRepository>();
    }
}
```

You may be thinking this looks exactly the same as when you register a view model inside of the locator. That's because it is. However, we are now able to access the SQLite database through the view models, like this:

```
public class MainViewModel : ViewModelBase
{
    readonly INavigationService navService;
    readonly IRepository sqliteService;

    public MainViewModel(INavigationService nav, IRepository sql)
    {
        navService = nav;
        sqliteService = sql;
        LoadAllPersonalInfo();
    }

    List<PersonalInfo> personalInfo;
    public List<PersonalInfo> PersonalInfo
    {
        get { return personalInfo;}
        set
        {
            personalInfo = value;
            RaisePropertyChanged("PersonalInfo");
        }
    }

    void LoadAllPersonalInfo()
    {
        if (personalInfo == null)
            personalInfo = new List<PersonalInfo>();
        PersonalInfo = sqliteService.GetList<PersonalInfo>();
    }
}
```

Note that we need to load the `PersonalInfo` into the list—we can't just assume it will be there. We should update the database on either a UI button click (via a `RelayCommand`) or when we navigate off the view.

Getting the Connection Information from the Platform

On each platform, the `ISQLiteConnectionFactory` interface will need to be implemented. On Android, it will look like this (it is similar on the other platforms):

```
public class SqLiteConnectionFactory : ISqLiteConnectionFactory
{
        readonly string Filename = "personalinfo.db";

        public SQLiteConnection GetConnection()
        {
            var path = System.Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            path = Path.Combine(path, Filename);
             SimpleIoc.Register<SqlConnectionFactory, ISqLiteConnectionFactory>();
            return new SQLiteConnection(path);
        }
    }
```

All this does is create the database (assuming that it doesn't already exist) and returns the connection with the path to the database file.

A fully working example of this is in the source code for this chapter.

Let's Extend This a Bit: Using the Device Address Book

Now that we have a basic model, let's create a new address book application, but this time based on the address book on each phone. We will start off in the same way as usual and create a new project called `AddressBook`. Import SQLite and MVVM Light as usual. Remember to also install the sqlite3 package described earlier in this chapter. Create in the PCL a directory called Interfaces, a directory called Models, and a directory called Helpers.

Add a directory into the platforms called Events and Interfaces. You can copy these from the SQLiteExample sources, but remember to change the namespace.

Our final step is to create our data models and amend the `DBHelper` class to reflect these new model names.

There are two ways to obtain information from the address book—directly through the mobile platform's API or via a plugin component. For our example here, we will be going the component route as it means we are able to unify the models and also speed things up a bit.

For this, we'll need to install the `Xamarin.Mobile` component from the Xamarin Component store. To reach this, double-click on the component folder (see Figure 4-11).

The Xamarin Component store will be closing at some point in 2018. Most packages available on there will be available via Nuget.

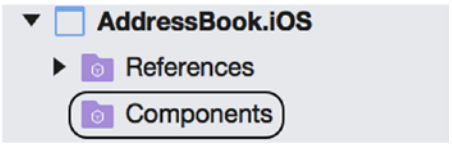


Figure 4-11. *The Components menu under the project. The box is my addition*

You will now be presented with a view. In the editor, click on the button asking you to add new components. As long as you are connected to the internet, you will see a new window. In the top left, type *mobile*. You will see results similar to those in Figure 4-12.

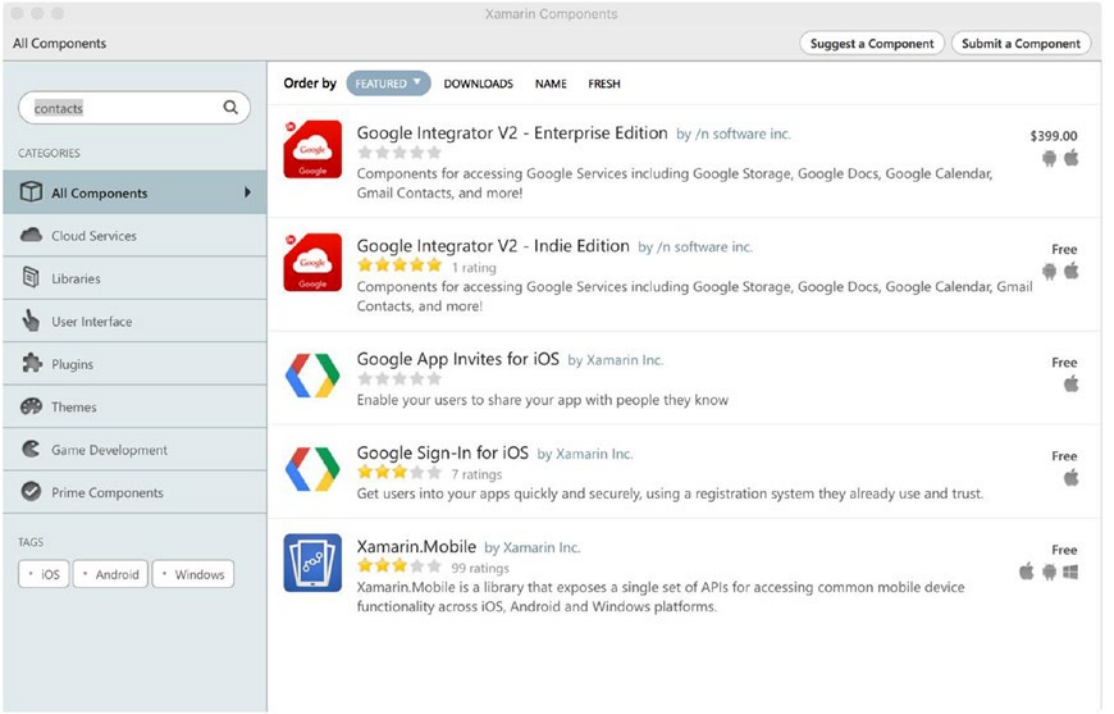


Figure 4-12. *Results from component store search*

Select Xamarin.Mobile and click on the *Add to App* button. Make sure it is version 0.7.7 or above. See Figure 4-13.

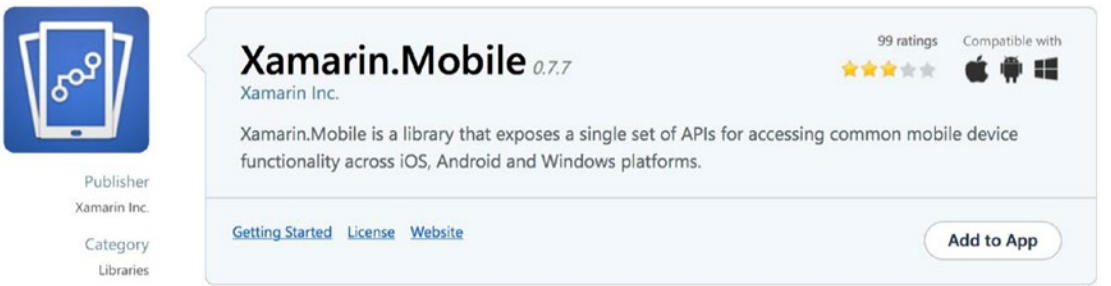


Figure 4-13. *Adding Xamarin.Mobile to the app*

Once you have added the component to the store, the Components menu will change to reflect the added component (Figure 4-14).

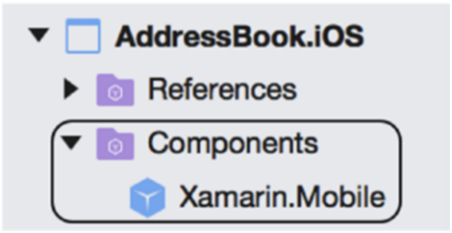


Figure 4-14. *Components menu showing Xamarin.Mobile has been added*

The interesting part of this project is that we are able to create our initial database in the PCL without very much work. Let's look at how to do it.

For iOS, permission has to be given to any application that is not part of the Apple application suite found on an iPhone in order for it to be able to access the address book. On Android, the `READ_CONTACTS` permission has to be set in the Manifest, with something similar required on Windows Phone.

Once we have permission, we can either just "throw" the data into the PCL or we can make the gathering of data part of the view model itself. To keep the integrity of the view/view model/model system, we will do this through the view model approach.

In order to store the data, we need a couple of models. As with the `SQLiteExample` earlier in this chapter, we have a model to store the person's address as well as the one we will abstract from the platform. If you have used any form of abstraction before, you will already know that you can only abstract something that is commonly available from all targets. For example, in Xamarin Forms, all targets have some way of displaying a text-only UI element (`TextView/UILabel/TextBox`) that is abstracted to the Forms `Label` UI element. In the case of the contacts, we are able to create a class from elements available on all platforms:

```
public class PhoneDetails : IInterface
{
    [PrimaryKey]
    public string id { get; set; }
    public string FullName { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Mobile { get; set; }
    public string Landline { get; set; }
    public string EmailAddress { get; set; }
    public string ImageFilename { get; set; }
}
```

The component allows the acquisition of the image associated with the contact, so we can make use of that, but we will only be able to store the filename to pass back to the UI.

We first need to get the data. This is performed using a `RelayCommand` attached to the button. The command is actually quite interesting. Remember, we can't access the data from the PCL. However, we can attach and send. Why do we want the button when we could just start on a different part of the UI?

The answer is simple: we need data to display data, and everything here is being performed synchronously (in other words, the UI cannot get on with something else until this data has arrived). When the UI starts, the data is grabbed. Once the data is in, the button is enabled, and we can carry on.

Letting the User Know Something Is Happening

An issue with anything performed synchronously is that when something happens, the UI slows (it may seem as though it has hung due to the slowdown). In this case, we should really employ a busy spinner graphic. This can be achieved by creating a base class both in the view model and on the platforms.

In the view model, we create a class called `BaseViewModel` that inherits `ViewModelBase`. For the other view models, we replace the `ViewModelBase` with `BaseViewModel`. We don't need to declare this class in the `ViewModelLocator`.

```
public class BaseViewModel : ViewModelBase
{
    bool isBusy;
    public bool IsBusy
    {
        get { return isBusy;}
        set { Set(() => IsBusy, ref isBusy, value, true);}
    }
}
```

In the platforms, we have to implement this and then let the UI classes inherit it. For example, the iOS base UI class will be:

```
public class BaseViewController<TViewModel> : UIViewController where TViewModel :
BaseViewModel
{
    private LoadingOverlay loadingOverlay;
    public BaseViewController(IntPtr handle) : base(handle)
    {
        ViewModel = SimpleIoc.Default.GetInstance<TViewModel>();
    }

    protected TViewModel ViewModel { get; set; }
    protected NavigationService GlobalNavigation
    {
        get
        {
            return (NavigationService)SimpleIoc.Default.GetInstance<NavigationService>().
            GetAndRemoveParameter(this);
        }
    }
    public override void ViewDidLoad()
    {
        PerformDataBind();
    }

    void PerformDataBind()
    {
        this.SetBinding(() => ViewModel.IsBusy).WhenSourceChanges(HandleLoadingNotifications);
    }
```

```
    void HandleLoadingNotifications()
    {
        if (ViewModel.IsBusy)
        {
            var bounds = UIScreen.MainScreen.Bounds; // portrait bounds
            if (UIApplication.SharedApplication.StatusBarOrientation ==
            UIInterfaceOrientation.LandscapeLeft || UIApplication.SharedApplication.
            StatusBarOrientation == UIInterfaceOrientation.LandscapeRight)
            {
                bounds.Size = new CGSize(bounds.Size.Height, bounds.Size.Width);
            }

            loadingOverlay = new LoadingOverlay(bounds);
            View.Add(loadingOverlay);
        }
        else
        {
            if (loadingOverlay != null)
            {
                loadingOverlay.Hide();
            }
        }
    }
```

The source for `LoadingOverlay` is in the iOS project. Similar code for Android and Windows Phone can also be used to affect the spinner. When we now inherit `UIViewController`, we also have the ability to include the spinner.

If we use the `UIViewController` though, we will need to have a link back to the view model in use. However, using

```
public partial class vcAddDetails : BaseViewController<AddDetailsViewModel>
```

means that we already have the view model that we want to use ready and available without a link back to the locator. This example is for iOS; for Android and Windows Phone, please refer to the sample code.

Ensuring that the button is correctly enabled is not handled on the UI level. Instead, it is handled in the `ViewController` using the `RaiseCanExecuteChanged();` method on the `RelayCommand` associated with the button. We still bind the button as we would normally.

How Do We Know When All of the Data Has Been Read?

Let's look at how the component is works. The following is a cut-down example (the full source is in the application):

```
List<Contact> list;
var book = new Xamarin.Contacts.AddressBook();

book.RequestPermission().ContinueWith(t =>
{
    if (!t.Result)
    {
```

```
        var alert = new UIAlertView("Permission denied", "User has denied this app access to
        their contacts", null, "Close");
        alert.Show();
    }
    else
    {
        list = book.OrderByDescending(c => c.LastName).ToList();
    }
}, TaskScheduler.FromCurrentSynchronizationContext());
```

We seek permission to access the address book. When we have it, the data is read into the list in descending order based on the `LastName` property.

When that has been done, it is a case of converting the `List<Contact>` into our internal format and sending that to the view model to store in the database. Once stored, `RaiseCanExecuteChanged()` is called and the button is enabled. The `IsBusy` property should also be set to `false` to switch off the spinner.

Passing the Data ID Between the View Models

As we saw in the previous chapter, it is possible to use the `Messenger` class to send information between the view models. There is another way to pass information around, however. It is possibly not as simple to use, but it is worth taking a look at anyway.

We have just clicked one of the items on the `List` user interface element. This will correspond to an ID. It is possible to take that ID from the model and pass the ID via the `RelayCommand`.

The `RelayCommand` would look something like this:

```
public RelayCommand NavigateToListItem
{
    get
    {
         return navigateToListItem
?? (navigateToListItem = new RelayCommand(
 () =>
        {
        navigationService.NavigateTo(Locator.ViewDetails
, listItemId);
        }
));
     }
}
```

The problem here is that we need to be able to get that ID from the navigation service. MVVM Light provides a method to retrieve that data from the view model. Unfortunately, it's not the same on all platforms.

For iOS

```
var id = GlobalNavigation.GetAndRemoveParameter(this) as string;
```

For Android

```
var id = GlobalNavigation.GetAndRemoveParameter<object>(this.Intent) as string;
```

For Windows Phone

```
var id = GlobalNavigation.GetAndRemoveParameter(this.NavigationContext) as string;
```

The type at the end can be a full object or a specific type (in the case of the preceding example, a `string`). `GlobalNavigation` is set up in the view superclass.

Adding the Component to the Windows Phone Project

For this part, I will be assuming that you have created the app on a Mac, so the Windows Phone project will not have been created. If you have created the app on a PC, the component will have been added with the correct references.

After you have created the WinPhone project on the PC, you will then need to add the component. The component is copied to the project root in the Components directory.

To begin, we need to add the reference to the library held within the component. Highlight the References folder in Project Explorer and select "Add Reference." You will be presented with a window similar to that shown in Figure 4-15.

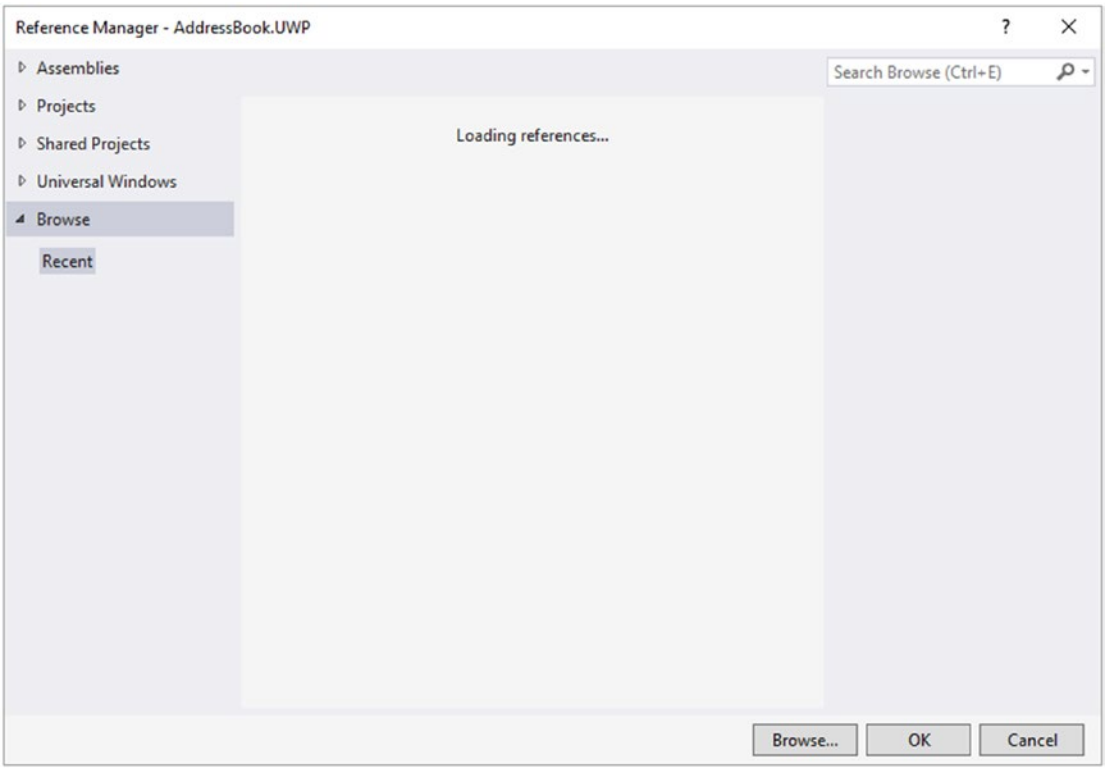


Figure 4-15. *Add Reference*

Click the *Browse* button. You will be presented with a standard filer window. Navigate to the root of the project until you see the Components directory (shown in Figure 4-16).

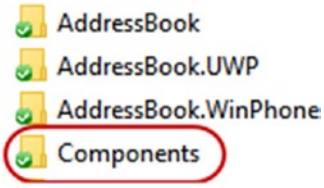


Figure 4-16. *Components directory*

Double-click the Components directory and navigate to the lib directory within the xamarin-mobile package. For Windows Phone 8.1, select the WP8 directory, and for WinPhone 10, the WinRT directory, shown in Figure 4-17.

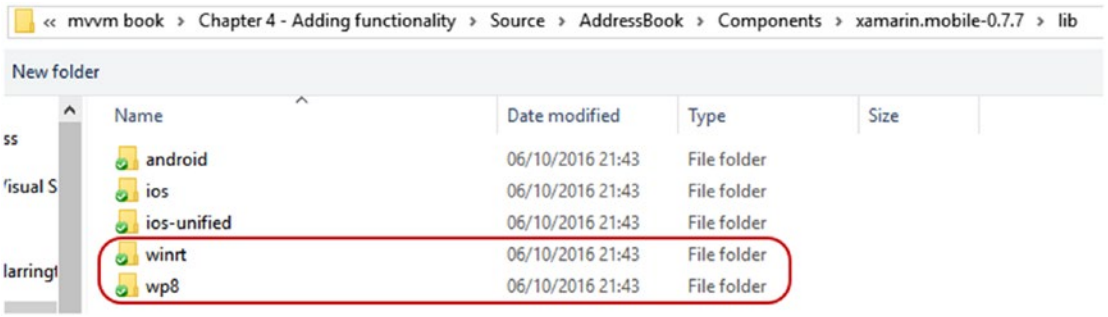


Figure 4-17. *The two Windows Phone directories*

Open the appropriate directory, and you will find a dll file (Figure 4-18).

Figure 4-18. *The Xamarin.Mobile.dll file*

Double-click this file. The filer window will close, and you will be returned to the Reference Manager. The selected package will show, as in Figure 4-19.

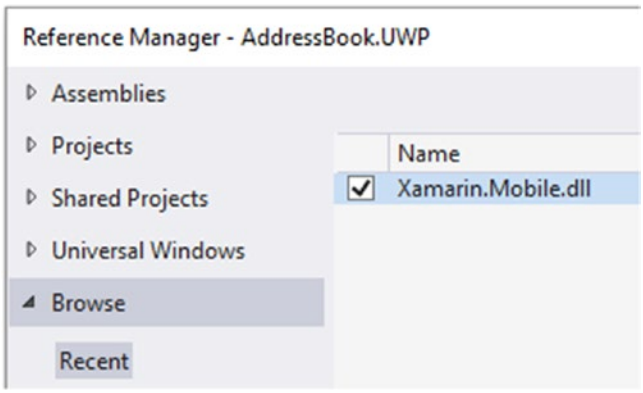


Figure 4-19. *Xamarin.Mobile.dll has been added to the project*

Once you have finished, click the *OK* button. The Reference Manager window will close, and the References folder will reflect the added dll (Figure 4-20).

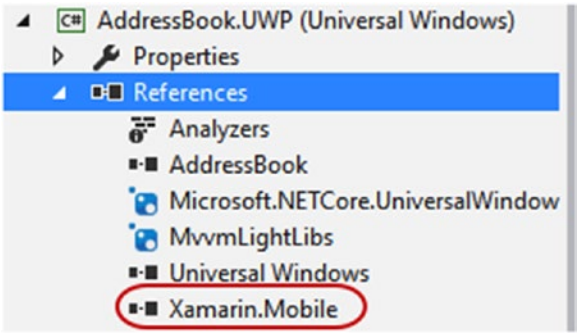


Figure 4-20. *The package has been added to the references and can now be used*

Displaying Data

Now that we have the data in the database, we need to display it. Key to all of the processes is the display of the list. Android and iOS use a `ListView` and `UITableView`. Windows Phone needs to do things slightly differently. While the `ListView` is a standard toolbox widget, the `ListViewItem` has to be created via the `ListView`.

To access the `ListViewItem`, right-click on the `ListView` in the designer window. You will be presented with the following menu (it has been cut down here).

When you select the menu option shown in Figure 4-21, the designer will add a small box at the top of the `ListView`. This is the `ListViewItem`. See Figure 4-22.

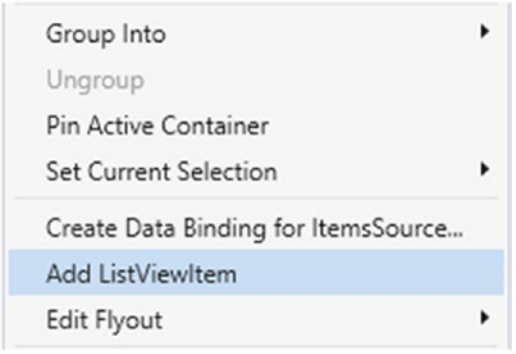


Figure 4-21. *Add ListViewItem on Windows Phone*

Figure 4-22. *ListViewItem*

If you recall, we have an image to the left for the user image and then the name and mobile number to the right.

The `ListViewItem` can be considered a mini view in itself, and so we can treat it as such and add whatever formatting we wish. In this case, we will have a grid and a `stack panel`. Our full `ListView` (with `ListViewItem`) looks like this:

```
<ListView x:Name="listView" HorizontalAlignment="Stretch" Margin="0,0,0,0"
VerticalAlignment="Stretch" ItemClick="listView_ItemClick">
    <ListViewItem>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Image x:Name="image" Height="32" Width="32" Grid.Column="0"
            VerticalAlignment="Center"/>
            <StackPanel Orientation="Vertical" Grid.Column="1" HorizontalAlignment="Stretch"
            Margin="8,0,0,0">
                <TextBlock x:Name="tbkName" Text="Some text" HorizontalAlignment="Stretch"/>
                <TextBlock x:Name="tbkMobile" Text="Some text" HorizontalAlignment=
                "Stretch"/>
            </StackPanel>
        </Grid>
    </ListViewItem>
</ListView>
```

The `TextBlock` text will contain the text from the `MiniContact` model.

■ **Note** If you are using both a Mac and a PC for development and have created the project on the Mac with the thought to later add the WinPhone/UWP projects, you may find the `InitializeComponent` within the codebehind will come up with an error that it can't be found. To fix this issue, close the project and open the folder containing the project. At the root of the project (where the `.sln` file resides), enable the showing of hidden files. You will see a directory called `.vs`. Delete this directory as well as the `.bin` and `.obj` files in the PCL and in the freshly created Windows Phone directories. When you reload the project, rebuild the entire project, and you should find the `InitializeComponent` will now no longer show an error.

Connecting to a Webservice

In this section, we will be creating a user interface to go along with a public webservice. In this case, we will be using the OpenWeatherMap service. OpenWeather is a weather aggregator service that takes data from many thousands of mini-weather stations from around the country you are in to provide a very accurate weather service.

For more details on the OpenWeatherMap service, please refer to their website: https://openweathermap.org.

The app contains three additional NuGet packages: `Xam.Plugin.Connectivity` (checks for some form of network connection), `Xam.Plugin.Geolocator` (obtains the geolocation of your phone), and `Newtonsoft.Json` (a very fast and efficient JSON serializer and deserializer library).

See the code flow in Figure 4-23.

Figure 4-23. *Data progression from geolocation to UI*

The reason it cannot be any other way is that, because the request for data is asynchronous, we cannot create the UI until we actually have data. Using MVVM Light makes this fairly simple, however, as we have the UI elements hang on the property value–changing event.

The webservice is very simple:

```
public static class Webservice
{
    readonly static string API_KEY = "8701bd54bcc76072ef55857b9c892a3f";

    public static async Task<WeatherItem> GetWeather(double lat, double lon)
    {
        var param = string.Format("lat={0}&lon={1}&APPID={2}", lat, lon, API_KEY);
        var request = WebRequest.Create("http://api.openweathermap.org/data/2.5/
        forecast?" + param) as HttpWebRequest;
        request.Method = "GET";
        request.Accept = "application/json";

        var Weather = new WeatherItem();
        string responseContent;
        try
        {
            var response = (HttpWebResponse)await Task.Factory.
            FromAsync<WebResponse>(request.BeginGetResponse, request.EndGetResponse,
            null);
            if (response.StatusCode == HttpStatusCode.OK)
            {
                using (var reader = new StreamReader(response.GetResponseStream()))
                {
                    responseContent = reader.ReadToEnd();
                }
                Weather = JsonConvert.DeserializeObject<WeatherItem>(responseContent);
            }
            return Weather;
        }
        catch (WebException ex)
        {
#if DEBUG
            Debug.WriteLine("Exception thrown : {0}--{1}", ex.Message,
            ex.InnerException);
```

```
#endif
                return Weather;
            }
        }
    }
```

We create and perform the request, and if the data comes through, we deserialize it into the `WeatherItem` object. This will also deserialize into the other classes, so we only need perform a single call to the service.

The app will also be "single shot"; if you wish to include a refresh, please feel free to improve the code sample.

Checking for Connectivity

It is a golden rule of thumb that before we go online, we check to see if we have a network connection. As the connection has to sit on the platform and not the framework, we need to be able to inform the framework of the change. The problem is that we can't do this directly, as typically we deal with a change of connection in an event that reflects onto a Bool, say, that the app then interrogates. This will be set up in a piece of code that does not have a view model associated with it.

In order to track the connection state, we need to do the following:

1. Check the connection and set the flag for online or offline.
2. Ensure that when the flag is changed, `OnPropertyChanged` is also broadcasted.
3. We either have a base class that each `ViewController`/activity/page inherits from that has a listener for `PropertyChanged` and sets the framework flag accordingly (this too will be in a `viewmodel` base class, therefore all view models have access to the property) **or** we simply have each `ViewController`/page/activity listen, which passes to the `viewmodel` base class.
4. When the app requires a webservice, the call goes through to the view model, which in turn checks for connection. If there is a connection, do the call, else display an error.

For the example, the code will load a website based on a click. Assuming there is a connection, you will see a site.

Note for iOS users: Due to Apple changing how apps can access arbitrary websites, the following was added to the `info.plist` file. For further information on this, please check https://developer.xamarin.com/guides/ios/platform_features/introduction_to_ios9/ats/; the following code allows all sites to load:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

What is important is how the binding works:

```
void DoBindings()
{
     Button.SetCommand(
           "TouchUpInside",
           ViewModel.WebsiteCommand);
```

```
        ViewModel.PropertyChanged += (sender, e) =>
        {
            if (e.PropertyName == "GoWebsite")
                if (ViewModel.GoWebsite)
                    InvokeOnMainThread(() =>
                                            wvWeb.LoadRequest(new NSUrlRequest(new
                                            Uri("http://www.liverpoolfc.net"))));
                else
                    InvokeOnMainThread(() =>
                                            wvWeb.LoadHtmlString("<html><body><p><b>Connecti
                                            on error</b></p></body></html>", null));
        };
}
```

We don't directly bind to the webview as there isn't a simple way to bind a URL to the website loader (`LoadRequest`). We therefore have to sit on the `PropertyChanged` event and wait for that to fire to retrieve the website.

GPS

In our final section, we will have a look at using the system GPS and getting some useful information from that about where you are. We will then extend it to show how fast you're going and the direction you're heading.

We will cover this in two ways: first using the `Xam.Plugin.Geolocation` NuGet plugin and then using GPS natively.

GPS can be an inaccurate system; how close you are to either an IP network or the telephone network towers as well as how many buildings are surrounding you will all influence how accurate your positioning will be. At best, GPS will be accurate to around 5m of your position.

GPS with the Plugin

We create our package as we would normally and install `Xam.Plugin.Geolocation` in each platform solution. To access the location service in the MVVM framework, we will create a simple service in much the same way as we create the service for SQLite.

The Problem with GPS, Cameras, Settings, and Quite a Few Other Things

Let's consider for a moment what MVVM stands for: Model-View-ViewModel. In terms of mobile, the view model and model sit with the PCL and provide and receive information to (or from) the view that sits firmly on the phone. The likes of the camera, settings, and GPS are firmly on the platform.

Can these fit into the MVVM paradigm? Unfortunately, the answer is not exactly clear. For a picture, we can store the image name and access it from within the view model, so in part yes. The same applies for geolocation; we can certainly store the data within the model, but we have the issue of getting the data into the model without incurring a problem known as a circular dependency.

A circular dependency looks like Figure 4-24.

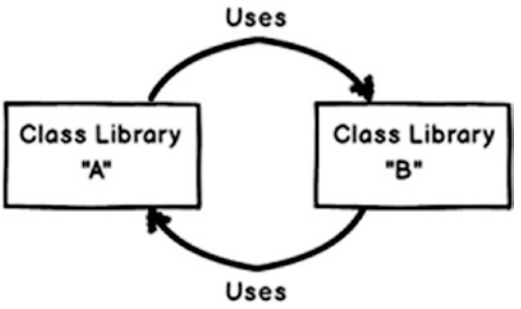


Figure 4-24. *Circular dependencies*

Class A is the PCL. The app is Class B. Typically, Class B uses the information from Class A. It supplies information back to Class A. This is normal and not a circular dependency; it is a two-way data flow. However, if Class A uses something specifically from Class B, we have a problem. As soon as Class B has been compiled, Class A is out of date. Class A therefore needs to be updated. If Class A relies on something from Class B that no longer exists, then Class A can't be recompiled, and the chances are that Class B will break because of it.

In our case, we have the platform supplying data to the model while the view model supplies information back to the platform. In other words, Class A relies on information from a function of Class B, while Class B relies on data from Class A. Look familiar?

Is There a Solution to This?

To remove this issue, we need to either create a service (the beauty of which means that, as with SQLite, the service can be used anywhere within the PCL) or use an event that the PCL listens into.

The problem with using an event to broadcast a change is that when you broadcast, everything can listen in. Is there a problem with that? Let's think. If only the PCL listens and only one class listens for it, then we don't have a problem. If we have more than one class listening, then we have the potential for conflicts. If we have classes in both the PCL and platform listening, we run into even more problems. If the event is on an active view, then that view feeds back to the view model, which feeds back to a model, which (you guessed it) possibly has had the same information, leading to the PCL having to do twice the work than it did originally.

Will a service work any better?

Let's look at the SQL service we've used. We have an interface defined in the MVVM framework that is implemented on the platform with the service, then registered. Easy enough. For something like geolocation, we have the potential for data to become stale. How can we prevent that from happening?

Thankfully, MVVM Light comes with a simple answer—the messenger service.

For the location service, we first need a model:

```
public class LocationService
{
     public double Longitude {get;set;}
     public double Latitude {get;set;}
     public double Heading {get;set;}
     public double Altitude {get;set;}
}
```

This is how we expect to see a model. The difference is that we need a reactive service. For that, we add an action that is a callback:

```
public Action LocationChanged {get;set;}
```

We can then add the service into the `BaseViewModel` (or, if we only use the location service in one view model, in there):

```
LocationService locationService;
public RelayCommand LocationChangedCommand {get; private set;}
public BaseViewModel(...)
{
     LocationChangedCommand = new RelayCommand(()=>Messenger.Default.Send(locationService));
     locationService = new LocationService();
     locationService.LocationChanged = UpdateLocation;
}
double latitude;
public double Latitude
{
     get {return latitude;}
     set {Set(()=>Latitude, ref latitude, value, true);}
}
// same for the other members of the service class
void UpdateLocation()
{
     Latitide = locationService.Latitude;
}
```

In the view, once the location has changed, we call the following:

```
T myMethod(LocationService locService)
{
    locService.LocationChanged.Invoke(null);
}
```

which fires off the callback within the PCL.

Conclusion

Problem solved? Yes and No. Sorry. For the majority of the time, the change of location is set in the `App.xaml.cs` (on Windows Phone), `Application.cs` (Android—I'm assuming that a singleton has been set for Android rather than just firing off via the `MainActivity`), or `AppDelegate.cs` (iOS). Typically, the singletons won't be bound to a view model. There is nothing for the service to be sent back to. Do we need to use a base page for all other views/activities/pages to inherit that listens for changes in the location?

The answer is no. If we implement `INotifyChanged` and have the framework listen for the change, the view model is then able to listen for a change and assign it to a property within the view model.

CHAPTER 5

Converting Your Existing Apps

By now you should be far more confident in how MVVM Light works, how to organize your code, and how to create apps that run on all three platforms and make use of their own user interface designs. Now, however, it is time to convert your existing apps to work with MVVM Light.

Why would you want to do this? For a start, it means that you can release your application on the other platforms, and also it can be argued that it makes for a simpler life when it comes to upgrading the codebase, adding new functionality, and fixing bugs.

In this chapter, we will convert a very simple Android application to run on all of the mobile platforms by use of MVVM Light. Once we have the code working, the functionality of the app will be extended to produce an internationalized version as well as to store the results for later retrieval.

The original source code is provided with this chapter for you to look at.

The app in question is a very simple app (it was actually the very first one I wrote), which makes it perfect for a makeover as it allows for additional functionality to be included.

I must thank my former student at Warrington Collegiate, Stacey Spiers, for providing the original artwork for the app and giving permission to use them here.

The Original App

The app is a simple three-tabbed app designed for forensic science students (Figures 5-1a to 5-1c). It used the Hessenge Nonogram to calculate the time of death for a body. It's a fairly well-known method of calculation, so it was simple enough to convert. The user interface was very simple.

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_5

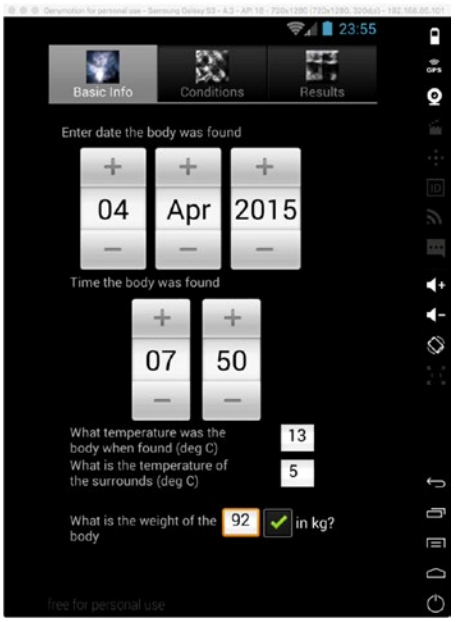


Figure 5-1a. *Enter the date*

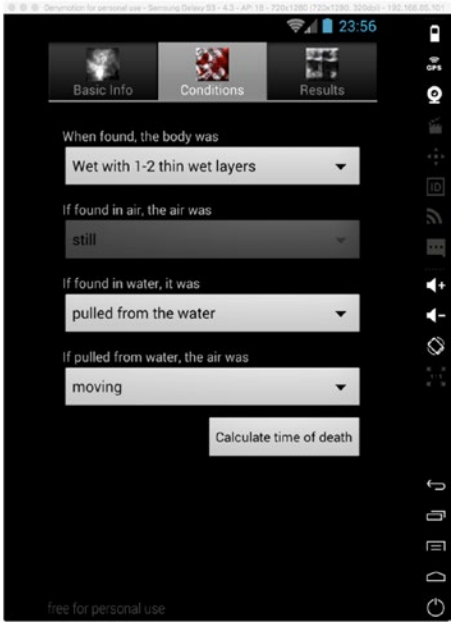


Figure 5-1b. *Enter the conditions*

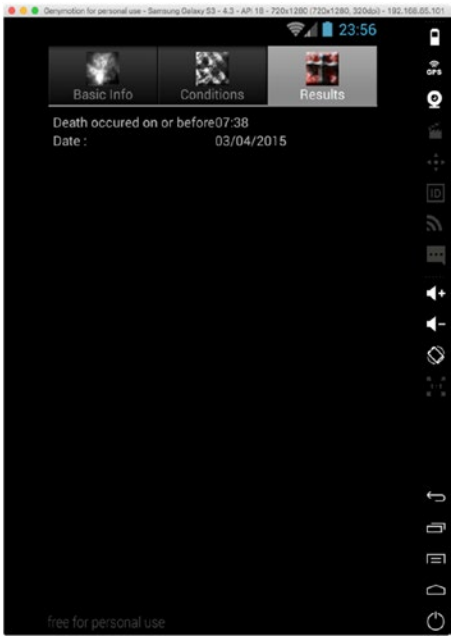


Figure 5-1c. *And here are the results*

As you can appreciate, the UI has a couple of drawbacks:

1. The results are a waste of a view.
2. There is little in the way of error checking.
3. The datetime-picking facility on the first view is massive.
4. The text is a mix of static (within the designer files) and the strings file.

That said, this was a first attempt, so this sort of *faux pas* can be forgiven.

The Redesign Process

Moving code over to a completely different framework is not a simple task. For example, the PCL uses a subset of the .NET framework. This means that only the parts of the .NET framework that exist within the subset are guaranteed to be implemented on all platforms. Thus, things like saving and loading files have to be implemented on the platform because they are excluded from the subset within the framework.

Other parts of the .NET framework you may take for granted (such as `ToShortDateString()` and some of the string formatters) are also not implemented.

This is not to say you cannot write an implementation of the missing functionality yourself. There are a number of NuGet packages that have been created solely to add back in the functionality missing from the subset (*PCL Storage* by Daniel Plaisted and *ACR File System* by Allan Ritchie both provide a plugin replacement for the missing `System.IO` libraries)

The User Interface Elements

One thing to remember about the user interface is that not everything is implemented on every platform. For example, there aren't checkboxes or radio groups on iOS, but then there isn't the range of image and video elements available on Windows Phone as there is on iOS. Some things don't directly move across.

Table 5-1 is a user interface element equivalence list. If something doesn't exist, you'll not see anything in that box.

Table 5-1. *User Interface Elements Equivalence List*

Android	iOS	Windows Phone
TimePickerDialog	UITimePicker	TimePicker
DatePickerDialog	UIDatePicker	DatePicker
Spinner	UIPicker	ListPicker
AlertDialog	UIAlertView	MessageBox
GridView	UICollectionView	LongListSelector
ScrollView	UIScrollView	ScrollViewer
SearchView	UISearchBar	PhoneTextBox
ProgressBar (Horizontal)	UIProgressView	ProgressBar
ProgressBar (Circle)	UIActivityIndicatorView	ProgressBar (Horizontal)
ListView	UITableView	LongListSelector
ActionBar Tabs	UITabBar	Pivot
SeekBar	UISlider	Slider
Switch	UISwitch	ToggleSwitch
RadioButton		RadioButton
CheckBox		CheckBox
WebView	UIWebView	WebBrowser
Button	UIButton	Button
TextView	UILabel	TextBlock
EditText	UITextField	TextBox
EditText (multiple lines)	UITextBox	TextBox
ImageView	UIImageView	Image
GoogleMap	MKMapView	Map
LinearLayout		StackPanel

This list isn't exhaustive, but it will help in porting the user interface between platforms.

Thinking in MVVM

An important aspect of moving an existing app over to the MVVM Light framework is to think in the MVVM pattern. It is a pointless exercise to just bring code over piecemeal from a platform to the PCL and end up with myriad problems for debugging. What you need to remember is that the whole point of MVVM is that it is designed to provide something known as *separation of concerns*.

For a standard app, this would mean untangling the UI from business logic. For an MVVM app, it is not only this, but also separating each part within the PCL into its own little "block." For example, we already saw in Chapter 3 that we had a class in its own directory called Models that was used for the single purpose of storing an instance of the message sent across. This is a prime example of a separation of concerns within the PCL. That file could have been placed anywhere (including in another file).

If it could be in another file, would it not have made sense to put in that other file?

No.

The whole point of the exercise is not only to make the code more readable (and therefore easier to debug) but also to structure it in such a way that it becomes easier and quicker to either add additional functionality or simply see what is going on.

With this in mind . . .

Step 1: Code Analysis

Here, we break down the code into four parts:

1. Models [PCL]
2. View models [PCL]
3. Helpers [PCL & Platform]
4. UI [Platform—except for error messages]

Models are any classes used purely for handling data.

Helper classes are classes used for the manipulation of data.

Why can we not just put the manipulation of the data within the view model? Recall what the view model is there for. It is the interface between the models and the view; in other words, it is a routing class to other functions (such as calling network APIs). The view model should contain just that. While this may seem messy, it actually makes a great deal of sense in terms of having clarity regarding what does what within the app (also, you don't have to go hunting for the manipulation code within the view models; this is especially important if you have a very complex app). See Figure 5-2.

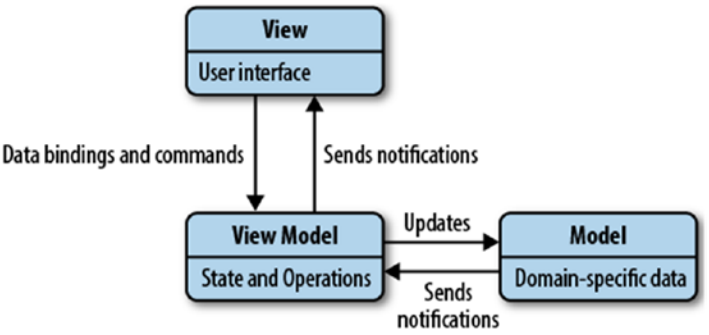


Figure 5-2. *A reminder of the areas each part of the MVVM Model deals with*

Step 2: Create the Models and Helpers

For these, remember the golden rule—for models, one class per file. The exception is if one or more models inherit from the same base class, in which case these can be kept in the same file.

When you move these files over to the PCL, you may notice that the built-in Intellisense will pick up methods or extension methods that do not exist.

Step 3: Recreate Any Missing Methods

This can be performed in one of two ways, and the choice should always depend on the complexity required to implement.

1. Write your own version of the missing method or extension method.
2. Create an interface and use dependency injection to retrieve from the platform the result of the missing method.

Step 4: Create the View Models

These should include any properties being used, including any `RelayCommand` properties; if the navigation service or any interfaces are being passed, a local copy that is instantiated from the values passed in to the constructor; and any methods required to process data required by any other code the view model may call.

Step 5: Refactor the UI

The UI should now be just the user interface elements. All data handling will be controlled via the view model (including error handling and error messages). While the UI on each platform will differ, they will still all need to handle events locally as well as bind to the view models.

The UI code will also now be far less cluttered and therefore will be simpler to optimize and fix.

Let's Get the Show on the Road

Code Analysis

The app is a tabbed-view app. This means that we have tabs that are handled by the parent view, with the tabs being child views that are rendered within the parent.

The `Activity.cs` file contains the data storage class (public static class `common`), seven activities (including one for the splash screen, one for the "About this App" menu option, and one that inherits `Activity` but is really a helper class for calculations).

There is also a platform-specific `DateChangedListener` class. This is there to pick up when the date changes.

The Activities

TimeTemp: Deals with the time and temperature view. Includes some data manipulation. These modify properties from the `common` data class.

StateFound: Deals with the state of the body when found. As with the `TimeTemp` activity, modifies the values of the `common` data class. Includes the *Calculate* button, which then triggers the error-checking routines before calling the `Calculate` activity class. UI includes spinners.

Results: Displays the results of the calculation. Requires two properties from the `common` class.

Information: Menu option displaying the "about us" details.

MainLauncher: Splash screen

AndroidTimeOfDeath: Parent activity. Includes the code for the menus.

Calculator: Deals with the calculations based on user input. Contains PCL missing methods `ToShortTimeString()` and `ToShortDateString()`.

The UI

With the exception of filling the spinners, all of the UIs are in standard XAML files. Fragments aren't in use. There is an issue though, as it is based on a `TabPage`.

Let's look at the potential issue with this. If we have a button that we press to move to another view, we can intercept using `Click`/`TouchUpInside` and then use the `RelayCommand` within the view model, which then moves the UI to the next View.

For iOS, the `UITabPageViewController` has a `UITabView` that itself has `UITabBarItems` attached to it. These too have a `Click` event (`UITabBarItemEvents`) that can themselves be intercepted and acted on via the `RelayCommand`.

We are now left with Android and Windows Phone.

If we look at the Android code, we set up the tabs like this:

```
spec = TabHost.NewTabSpec("timedate");
spec.SetIndicator("Basic Info", Resources.GetDrawable(Resource.Drawable.ic_tab_weather));
spec.SetContent(intent);
TabHost.AddTab(spec);
```

The UI is created before we see anything, so there isn't any navigation (well, none to really speak of). We have something similar for Windows Phone in that we have a `Pivot` (or similar) that has a `UserControl` to display the view. The same applies though; the UI is there, and we don't really have a navigation to hang on to. We also don't want to have to create something specific for iOS within the view model.

Thankfully, we don't have to. If a platform doesn't subscribe to an event, then even though the code is in the view model, it is essentially ignored by the platforms that aren't subscribed to the event.

Let's look at some code to see how this will all work (the source for which is in Chapter 6/TabExample)

The Setup

The application is set up to have a single-tab layout with three views coming from it. The List page contains a simple list, the Data page contains some static text, and the Rotate page allows an image to be rotated. Each platform has its own UI and is attached to the view model as it would be for any other app.

If we look at how the platforms link up to the view models—and specifically, how they navigate—we can see that allowances have to be made for the `TabHost`/`Pivot` layouts.

The data page pulls in the model and creates a list view/table view for the platform and populates it, while the rotate page places an image on screen and allows for the image to be rotated.

Dealing with Events

As a base, MVVM Light can deal with `System.Event` types, such as `TextChanged`, `Click`, and `TouchUpInside`. Unfortunately, there are many more different events that need to be dealt with. In our application, we will need to deal with the spinner item selected event.

To work with events that aren't a simple `System.Event` type, we need to change how we do things.

For example, the `ActionBar` has `ActionBar.TabEventArgs`. This event cannot be covered using the standard `SetCommand`, but we can still bind to the `SetCommand`.

`SetCommand` itself is actually `SetCommand<T>(string eventName, ICommand command) where T is EventArgs`. We can therefore replace `T = EventArgs` to be `ActionBar.TabEventArgs`. We then just need to pass in the event name as the first parameter, and the `ICommand` can be placed within the source or within the VM.

For moving the `ActionBar` tab, we can use the following:

```
tab.SetCommand<ActionBar.TabEventArgs>("TabSelected", TabClicked);
```

The `ICommand` would be as follows:

```
ICommand TabClicked
{
    get
    {
         return new RelayCommand(() =>
         {
             tab.TabSelected += (object sender, ActionBar.TabEventArgs e) =>
             TabOnTabSelected(sender, e);
         });
    }
}
```

We can use this form of non-primitive event irrespective of the event and platform.

Moving the Code Over to the PCL

One of the first aspects we need to consider is how to deal with the model. In the original, a public static class is used, meaning that the class can be accessed by all of the activities. Within a standalone app, this is fine; however, with MVVM we need to ensure the view is kept apart from the model and only have that accessed through the view model. With a public static class, all of the properties will be available not only to the PCL, but also to the platform. This can cause issues where the UI is able to directly alter the model properties, leading to potential issues where the view model also has access to the model properties.

Within MVVM Light, we are able to pass an interface as a parameter to the view model (in the same way as we do for `INavigationService`). In order to do this, not only is an interface needed, but we also need to create and register the service.

Creating the Service, Interface, and Registering

For the interface, it is best to have a method that will act as a property to the class (in other words, get and set the class).

```
namespace TimeOfDeath.Interfaces
{
    public interface ICalculateData
    {
        void SetCalcData(CalcData data);
        CalcData GetCalcData();
    }
}
```

The service implements the interface. A check is placed in to ensure we have a valid instance of the `CalcData` class:

```
public class CalcDataService : ICalculateData
{
    CalcData calcData;

    public CalcData GetCalcData()
    {
        return calcData == null ? new CalcData() : calcData;
    }

    public void SetCalcData(CalcData data)
    {
        if (calcData == null)
            calcData = new CalcData();

        calcData = data;
    }
}
```

Unfortunately, as it stands, this service won't allow direct access to the properties of the class. For that, we need to use a generic method to get and set an individual property. Of course, we could simply add a raft of methods to set each property, but this becomes untenable as the number and size of a class grows.

To get and set a property, we make use of reflection:

```
public void SetData(string propertyName, string value)
{
    try
    {
         var property = calcData.GetType().GetRuntimeProperty(propertyName);
         property.SetValue(calcData, Convert.ChangeType(value, property.PropertyType), null);
    }
    catch (Exception ex)
    {
         Debug.WriteLine("Exception thrown - {0}", ex.Message);
    }
}
```

```
public T GetData<T>(string propertyName)
{
    try
    {
         var property = calcData.GetType().GetRuntimeProperty(propertyName);
         return (T)property.GetValue(property, null);
    }
    catch (Exception ex)
    {
         Debug.WriteLine("Exception thrown - {0}", ex.Message);
         return default(T);
    }
}
```

Finally, the interface and service has to be registered within the `ViewModelLocator`:

```
SimpleIoc.Default.Register<ICalculateData, CalcDataService>();
```

We can now pass the interface into the view model constructor, like this:

```
public MyViewModel(INavigationService nav, ICalculateData calc)
```

Adding the Model

The model can be moved into the PCL, but it needs to be changed to be a non-static class with non-static members. It is up to you if you wish to use a method to set a property, like this:

```
public string MyString {get; private set;}
public void SetMyString(string value = "")
{
    MyString = value;
}
```

This will involve a bit of extra work in the view model to generate the `RaisePropertyChanged` event when `SetMyString` is called.

Adding the Calculation Helper

The `Calculate` activity can have the `Activity` inheritance removed and used directly as the helper class. The `ToShortDataString` and `ToShortTimeString` extensions can be rewritten, or you can use DI to have the platform provide the facility (see Chapter 3 on how to perform this sort of dependency injection).

As the extensions are small enough, the following can be used to replicate the missing functionality:

```
public static class ShortDates
{
    public static string ToShortDateString(this DateTime date)
    {
        return date.Date.ToString("dd/MM/YYYY");
    }
```

```
    public static string ToShortTimeString(this DateTime time)
    {
        return string.Format("{0}:{1}", time.Hour, time.Minute);
    }
}
```

The View Models

As a basic template, each activity is a view model. We don't need a view model for the parent or the results tab (the results tab will be an alert box, so we now have two views to generate). We will need to pass in `INavigationService` and `ICalculateData` to each view model.

We do not (and cannot) implement any UI elements, so we should only include any processing or properties from the view. Let's consider the `TimeTemp` activity:

```
[Activity]
public class TimeTemp : Activity
{
    DateTime today;

    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        SetContentView(Resource.Layout.datepicker3);
        today = DateTime.Now;
        EditText surround = FindViewById<EditText>(Resource.Id.tempSurround);
        surround.TextChanged += new EventHandler<Android.Text.TextChangedEventArgs>
        (alterValue);

        EditText body = FindViewById<EditText>(Resource.Id.tempBody);
        body.TextChanged += new EventHandler<Android.Text.TextChangedEventArgs>(alterValue);

        EditText bweight = FindViewById<EditText>(Resource.Id.weightBody);
        bweight.TextChanged += new EventHandler<Android.Text.TextChangedEventArgs>(alterValue);

        TimePicker timeFound = FindViewById<TimePicker>(Resource.Id.timeFound);
        timeFound.SetIs24HourView(Java.Lang.Boolean.True);
        timeFound.TimeChanged += new EventHandler<TimePicker.TimeChangedEventArgs>(alterTime);

        DatePicker dater = FindViewById<DatePicker>(Resource.Id.dateDisplay);
        int monthFix = today.Month - 1;
        dater.Init(today.Year, monthFix, today.Day, new DateChangedListener((DatePicker,
        year, month, day) =>
        {
             DateTime c = common.date;
             DateTime d = new DateTime(year, month + 1, day, c.Hour, c.Minute, 0);
             common.date = d;
        }));
```

```
        CheckBox weight = FindViewById<CheckBox>(Resource.Id.weightBodyKG);
        weight.CheckedChange += new EventHandler<Android.Widget.CompoundButton.CheckedChange
        EventArgs>(convertUnits);
    }

    private void alterTime(object sender, TimePicker.TimeChangedEventArgs e)
    {
        DateTime to = common.date;
        DateTime t = new DateTime(to.Year, to.Month, to.Day, e.HourOfDay, e.Minute, 0);
         common.date = t;
    }

    private void alterValue(object sender, EventArgs e)
    {
        EditText et = (EditText)sender;
        if (et.Text != "")
        {
            switch (et.Id)
            {
                case Resource.Id.weightBody:
                    common.bodyWeight = Convert.ToDouble(et.Text);
                    break;
                case Resource.Id.tempBody:
                    common.tempBody = Convert.ToDouble(et.Text);
                    break;
                case Resource.Id.tempSurround:
                    common.tempSurround = Convert.ToDouble(et.Text);
                    break;
            }
        }
    }

    private void convertUnits(object sender, EventArgs e)
    {
        CheckBox cb = (CheckBox)sender;
        EditText et = FindViewById<EditText>(Resource.Id.weightBody);
        if (et.Text != "")
        {
            double ow = Convert.ToDouble(et.Text);
            if (ow != 0)
                common.bodyWeight = cb.Checked == false ? ow * 6.35029318 : ow;
            else
                cb.Checked = true;
        }
     }
}
```

What can we remove from this? The easy answer is just about everything. We need to be able to set anything that starts with `common`. Anything else is UI based, so it needs to stay on the platform.

Let's construct the view model for this:

```
public class TimeTempViewModel : ViewModelBase
{
    INavigationService navService;
    ICalculateData calcData;

    public TimeTempViewModel(INavigationService nav, ICalculateData calc)
    {
         navService = nav;
         calcData = calc;
    }

    public DateTime SetDate
    {
         get { return calcData.GetData<DateTime>("date"); }
         set
         {
             calcData.SetData("date", value);
             RaisePropertyChanged("SetDate");
         }
    }

    public double TempBody
    {
        get { return calcData.GetData<double>("tempWeight"); }
        set
        {
            calcData.SetData("tempWeight", value);
            RaisePropertyChanged("tempWeight");
        }
    }

    public double TempSurround
    {
        get { return calcData.GetData<double>("tempSurround"); }
        set
        {
            calcData.SetData("tempSurround", value);
            RaisePropertyChanged("tempSurround");
        }
    }

    public double BodyWeight
    {
        get { return calcData.GetData<double>("bodyWeight"); }
        set
        {
            calcData.SetData("bodyWeight", value);
        }
    }
}
```

We will, of course, need to be able to navigate between views. We have seen how to do this a number of times.

Something similar will need to be created for the other activities. Most of these will just be setting the properties.

Returning the Results

Earlier in the chapter, it was identified that the `Calculation` activity wasn't actually an activity and that in the MVVM world it would be considered a helper class. In the original version, it is called from the `StateFound` activity. In the new version, it is called from the `StateFoundViewModel`. The view model still performs the parameter checking. This is all performed within a `RelayCommand`:

```
RelayCommand btnCalculateResults;
public RelayCommand BtnCalculateResults
{
      get
      {
           return btnCalculateResults ??
 (
               btnCalculateResults = new RelayCommand(
                  async () =>
                  {
                       var dialog = ServiceLocator.Current.GetInstance<IDialogService>();
                       if (TempBody < TempSurround)
                       {
                           await dialog.ShowError("Time Of Death Error", "Body temperature
                           must be greater than surrounds",
                               "OK", null);
                           return;
                       }
                       if (BodyWeight <= 0)
                       {
                           await dialog.ShowError("Time Of Death Error", "The body has to
                           have some weight", "OK", null);
                           return;
                       }
                       if (TempBody == -999)
                       {
                           await dialog.ShowError("Time Of Death Error", "Temp of surround =
                           null", "OK", null);
                           return;
                       }
                       if (TempSurround == -999)
                       {
                           await dialog.ShowError("Time Of Death Error", "Temp of surround =
                           null", "OK", null);
                           return;
                           }
                           // call the helper
                           new Calculation(calcData.GetCalcData()).calcTOD(TODDate);
                           // output the dialog box
```

```
                    await dialog.ShowMessage("Time of Death result",
                                string.Format("Death occured on or before {0}\nAt or
                                around {1}", DateOfDeath, TimeOfDeath),
                                "OK", null);
                }));
    }
}
```

Implementing the Tabs

Typically, when moving between views, the UI calls a `RelayCommand`, which in turn calls `navigate.NavigateTo(...)` to move to the new view. Tabs work differently, and so we have to allow the platform to perform the navigation.

This is not quite the truth though. It is entirely possible to have the view model control the navigation for a tab. The problem is that the view model is specific to that platform, so if you are only creating an app for a single platform, this is fine. For a multi-platform solution, using the view model within the PCL to control the navigation will not work.

We can still use view models and helpers to ensure we can create cross-platform solutions.

A Tab Gotcha (Android Specific)

When we create the activity views, we split the UI objects from the main activity. With a normal activity, this isn't an issue, as we create once and the resource IDs stay the same. With a fragment though, the fragments are inflated on the fly, so the resource IDs are different.

The problem is that the way we create the UI is based on the resources remaining the same.

Let's consider what I mean by this:

```
public TextView TxtView => textView ?? (textView = view.FindViewById<TextView>(Resource.
Id.textView1));
```

This line creates a public reference to a `textview`, but only if `textView` is `null`. Once it has been assigned, the object never needs to be refreshed. The upshot is that everything works for the first time, but never after that.

We can fix this issue by adding a small amount of code. First, within the source file where we define the UI objects, we need to set all of the objects to null. Thankfully, we can set all of the same type to null at the same time. For example:

```
TextView txtView1, txtView2, txtView3, txtView4;
ImageView imgView1, imgView2;
```

We next create a method:

```
void KillViews()
{
     txtView1 = txtView2 = txtView3 = txtView4 = txtView5 = null;
     imgView1 = imgView2 = null;
}
```

Note that we only destroy the objects, not the object references.

We then call this method within the `OnDestroyView` method:

```
public override void OnDestroyView()
{
    base.OnDestroyView();
    KillViews();
}
```

This issue does not occur with iOS or UWP.

Data Persistence

Given that each tab is considered its own entity, as soon as we move between tabs, any data from the first tab will be lost. We can always pass the data via the messaging system or even as the second parameter in the `NavigateTo` method. However, that is messy.

The best way to ensure the data is maintained between tabs is to create a `BaseViewModel` and for each VM to inherit it. The base has a static reference to the `CalcData` model, and from that all data is maintained between tabs.

```
    public class BaseViewModel : ViewModelBase
    {
        static CalcData calcData;
        public CalcData CalcData
        {
            get { return calcData; }
            set { Set(() => CalcData, ref calcData, value); }
        }
    }
```

Note, it is only the private reference that is `static`, not the public.

The `CalcData` model property is now available to any VM that inherits the `BaseViewModel`:

```
    public class MainViewModel : BaseViewModel
    {
```

This is one way to ensure data is persisted. Another way to do it is to use reflection. In this app, we have a service called `CalcDataService`, the meat of which is in the following methods:

```
public void SetData<T>(string propertyName, T value)
{
     try
     {
         var property = calcData.GetType().GetRuntimeProperty(propertyName);
         property.SetValue(calcData, value, null);
     }
     catch (Exception ex)
     {
         Debug.WriteLine("Exception thrown - {0}", ex.Message);
     }
}
```

```
public T GetData<T>(string propertyName)
{
     try
     {
         var property = calcData.GetType().GetRuntimeProperty(propertyName);
         return (T)property.GetValue(property, null);
     }
     catch (Exception ex)
     {
         Debug.WriteLine("Exception thrown - {0}", ex.Message);
         return default(T);
     }
}
```

This is a convenient method of getting and setting properties, but it does have some drawbacks, the largest being the performance hit reflection can produce. In this example, it's hardly noticeable, but it is worth remembering if you intend to use it in larger projects.

Platform User Interfaces

We have a basic UI on Android, and for what it is, it's a bit of a mess. The date and time pickers are very large, and other entries are inconsistent. Let's see if we can improve the look.

iOS

One obvious change is in the date-time spinners. On iOS, these can be implemented as a single spinner, but they don't look that appealing (Figure 5-3).

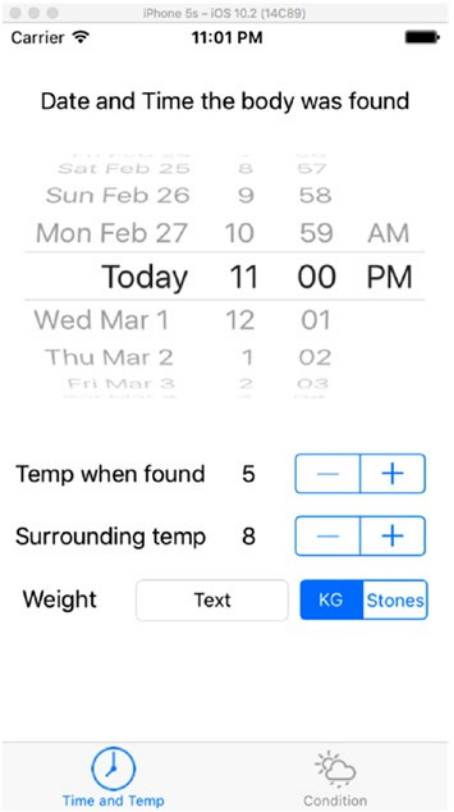


Figure 5-3. *Date and time spinners*

However, as a first run, it's not that bad. We have steppers to change the temperatures (very handy, as it means we can validate the temperatures quickly and ensure that the data being used is correct), and we have a simple method of selecting between the weight units.

The issue we have though is on the conditions view. Recall the Android UI (I've added a couple labels to Figure 5-4 for clarity).

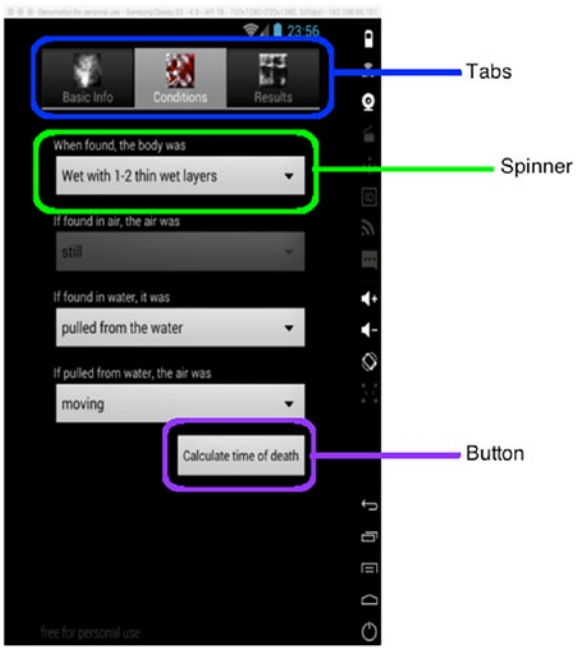


Figure 5-4. *Android UI*

We have four spinners. If we translate this to the standard iOS `UIPicker`, we get what is shown in Figure 5-5.

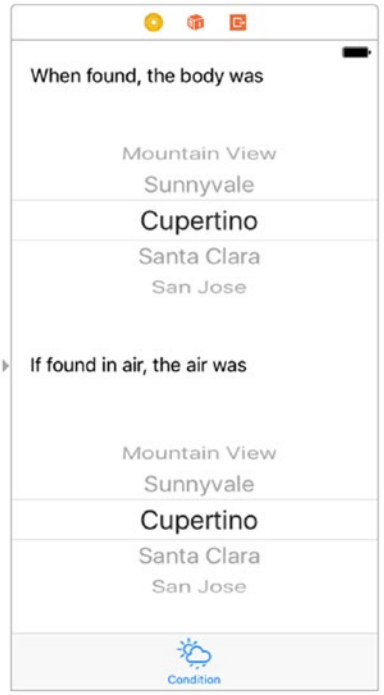


Figure 5-5. *iOS spinners*

We're Going to Need a Bigger Boat . . .

It's pretty obvious that if we have a `UILabel` followed by the `UISpinner` system, we're going to need a far larger amount of screen area, which is available via a `UIScrollView`. What we're trying to do, however, is make the UI nicer, and a scroll view won't cut it. We could always have another tab for two of the spinners, but that too looks messy.

There is an alternative. iOS is extremely flexible; it allows a picker to be added to a `textview`, so as soon as the `textview` gets the focus, the spinner appears. See Figure 5-6.

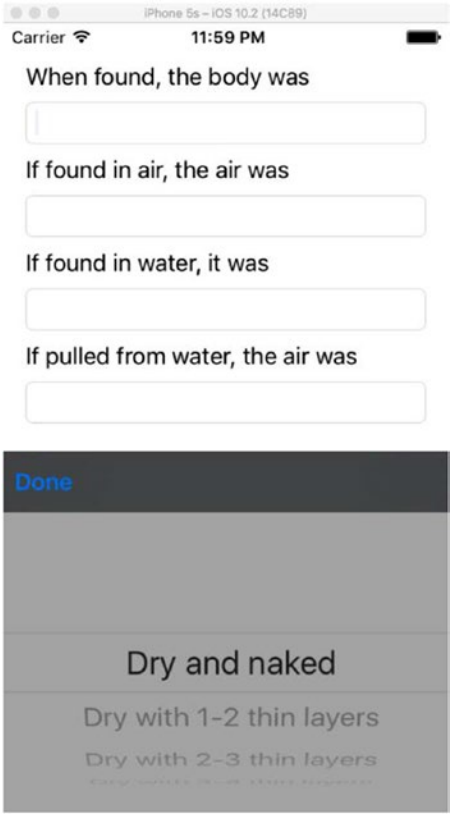


Figure 5-6. *iOS TextView with spinners*

It is outside of the scope of this chapter to go into how this is done, but we can examine how the framework deals with this.

Dealing with the View Model

In the original version, when certain options were picked, it would gray out other options (very similar to how a button can be disabled or enabled). We do have an issue, however. For buttons, we can use `RaiseCanExecuteChanged`, which is connected to a `RelayCommand`. Here, we can't, as there isn't a `RelayCommand` attached to the `UITextView`.

We can, however, use `WhenSourceChanges`:

```
txtBody.SetBinding(
                () => ViewModel.BodyCondition,
                BindingMode.TwoWay).WhenSourceChanges(()=> { txtFoundWater.Enabled =
                ViewModel.FoundWater; txtFoundAir.Enabled = ViewModel.Air;});
```

There is an additional "gotcha" when using the binding in this way; if you think about it, when the view model pointer is created, everything in that view model will be `null`, so what we're actually saying here is that we are setting `txtBody` to `null` (as `BodyCondition` is `null` when the VM pointer is created).

Can we do this another way? For example, can we bind directly to the UI element so that the UI doesn't need to use `WhenSourceChanges`?

This should work:

```
this.SetBinding(
                () => ViewModel.FoundWater,
                () => txtFoundWater.Enabled,
                BindingMode.TwoWay);
```

For Android and Windows Phone, this is fine. For iOS, though, it may not work (it should when running on the simulator though). With the exception of iOS, the devices use something called JiT rendering (JiT stands for Just in Time). In other words, the view is only generated moments before you get to see it. iOS uses a different model called AoT (Ahead of Time). The user interface is generated well in advance. This helps the device look reactive, as everything is there ahead of when it's needed. It looks great, but if `ViewModel.FoundWater` has not been set, it will be false, and the element will be grayed out. If `ViewModel.FoundWater` is changed within the view model (even with the event's being broadcast), the update has to be handled using `WhenSourceChanges`. However, it's not just a case of using `WhenSourceChanges`—we have to chain.

For Android, we would use this:

```
SpinBodyCondition.SetBinding(() => SpinBodyCondition.SelectedItemPosition)
        .ObserveSourceEvent<AdapterView.ItemSelectedEventArgs>("ItemSelected")
        .WhenSourceChanges(() => Console.WriteLine("SelectedItem = {0}",
                SpinBodyCondition.SelectedItemPosition));
```

What is this actually doing?

To start with, we're not binding to the view model, but rather to a property within the element—in this case, the `Spinner.SelectedItemPosition`. We're next going to attach `ObserveSourceEvent`. This gives a hook onto `AdapterView.ItemSelectedEventArgs`, with the event name's being hung on being `ItemSelected`. It is essentially the same as this:

```
SpinBodyCondition.ItemSelected += (s, AdapterView.ItemSelectedEventArgs e) =>
```

By itself this isn't going to do very much. We have to intercept when the source (the property bound to) changes and do something with it:

```
WhenSourceChanges(() => Console.WriteLine("SelectedItem = {0}",
        SpinBodyCondition.SelectedItemPosition));
```

To make this do something similar, we need to pass the text at the changed position back into the view model:

```
WhenSourceChanges(() => ViewModel.BodyCondition = ViewModel.GetBodyLayers[SpinBodyCondition.
SelectedItemPosition]);
```

Dealing with Nothing

It is not impossible for an object either from the view model or from the UI to be `null`. For example, if a table is bound to an asynchronous call and the call fails, the view model may return null, and setting an object to null destroys the object. You cannot bind null to anything and expect it to work.

To ensure the app doesn't crash, MVVM Light provides two fallback values: `fallbackValue` (for the model object) and `targetNullValue` (the value passed to the target):

```
this.SetBinding(
        ()=>ViewModel.SomeProperty,
        ()=>txtMyTextView.Text,
        fallbackValue: “Model is null”,
        targetNullValue: “value is null”);
```

Events on Windows Phone

Up to this point, we've concentrated on dealing with events on Android and iOS. Windows Phone creates the UI in XAML. Do we still need to do something special behind the scenes in there?

Windows Phone doesn't bind in the same way as Android and iOS do. While we are still able to bind to the likes of `ItemsSource` directly, and in some cases use `Command` and `CommandParameter`, detection of when the spinner is altered has to be handled via an event handler (which is in the code behind). Let's consider a single spinner:

```
<StackPanel Orientation="Vertical">
     <TextBlock Text="When found, the body was"/>
     <ListBox x:Name="BodyCond" ItemsSource="{Binding GetBodyLayers}" SelectionChanged="Spi
nnerChanged"/>
</StackPanel>
```

The `ListBox` has a `SelectionChanged` event that we hang off. The code behind is a more typical event than how we have had the bind set up in the Android and iOS projects:

```
void SpinnerChanged(object sender, SelectionChangedEventArgs e)
{
     var spinner = sender as ListBox;
     switch(spinner.Name)
     {
          case "BodyCond":
               ViewModel.BodyCondition = ViewModel.GetBodyLayers[spinner.SelectedIndex];
               break;
     }
}
```

(This is only a snippet of the code—the rest follows the same form.)

Putting Everything Together

We now have everything in place to write and compile the code to produce our new cross-platform app from the original code.

Admittedly, this isn't a groundbreaking app to have brought across; however, it does give the fundamentals of how to move code to the framework and shows how little is actually needed in the user interface code to produce the app.

For larger projects, the same methodology can still be employed. Things like LINQ are supported within the framework. Where you may have issues is regarding anything to do with using files, but even there solutions exist (`PCLStorage`, for example) on NuGet that allow easy access to storing data on the device.

Conclusion

Moving an app from a non-MVVM project to an MVVM project may take some time to code correctly, but it does make the creation of the app on other platforms many times simpler (and quicker). It does require some planning when creating the VMs and business logic, but in the end the advantages can be great.

CHAPTER 6

The Outside World

Many applications rely on webservices in today's modern, mobile world. If you take a look at many of the more popular apps, it's obvious that they don't work in isolation—they require data to go to and from a service. Even apps that you don't think use a service often do.

Dealing with online materials does lead to its own unique problems, which as an end user you may or may not have ever considered. Think about a text-message system. The user sends a message to the server, the server sends a notification to the other user(s), the other user(s) read(s) the message. It sounds simple, but as with anything simple, more than that happens. Consider just the message. Not only will it have an identifier for the sender (typically the phone number), but it will also have a date/time stamp, a message unique identifier, and the previous message identifier (an empty identifier means that the message is the "base" message).

The message is then requested from the server, which sends the message to the phone asynchronously (an asynchronous process lets the caller continue, and at some other time the process ends and the data can be used. This means that it is entirely possible for the user interface to crash the app if you're not careful (the app is trying to show something that doesn't yet exist).

While this chapter is not going to cover something as complex as this, we will be looking at how to interface with the outside world.

What Is Involved?

In order for an app to talk to the server, the server requires some form of code to take the request from the app and do something with it. For example, when an app requests data, it cannot talk directly to the database on the server. It would be a big security risk, to start with. Take the following example as for why this is.

An app communicates directly with the server and sends the following:

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

We have exposed the names of three tables and their relationships, as well as the field names from the databases (including ID names). If anyone is listening to network traffic maliciously, they are able to not only obtain a full dump of those tables, but as they know the table names they are also able to change or (worse) delete data.

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_6

However, if we send a simple POST request to the server, not only do we have a level of security, but we are also protecting the database and any internal structures used. Any would-be attacker has no access to the system and is only able to request the data the API allows. Add in a password and possibly a token, and the ability to compromise data is greatly reduced. It goes without saying that https should be used for any sort of security at a base level.

The purpose of the service is to only return data that you need, not everything else. The service may also take a lot of the heavy processing away from the app. While phones are powerful, they are like a cheap digital watch compared to a PC when put next to the processing power of a server.

The app must "talk" to the service in a particular way. It cannot send a message that says, "Hey, give me the data for the last day of April 2015." This would return way too much information—potentially information that you should not have access to. We can restrict the return by having the app send a message that says, "Hey, ID 12345 here. Give me the data for the last day of April 2015." The server first checks that 12345 exists and, if it does, what it can and cannot have access to. This may still be a lot of information, so the message being sent can be limited further to, "Hey, ID 12345 here. Give me the data for the last day of April 2015 for hard-drive purchases." Again, if the user is valid and is allowed that level of information, the data is returned.

Obviously, the app won't talk to the service in this way, but it helps in understanding to see it this way.

The app communicates using something known as an API (application programming interface). The API can be thought of in the same way as a class within an app. For example, I may call a method using the following:

```
string SomeMethod(string a, string b, int c = 1) {...}
var s = SomeMethod("hello", "world");
```

If the final int is not passed in, the method returns the strings concatenated in the order they're passed in. If the int is passed in (and is not equal to 1), the method returns the string concatenated in reverse. A Bool could also be used in this example.

An API works in the same way, except the app is calling a method sitting on a server. There are a couple of differences though. The biggest is that the calls are asynchronous. This is to ensure that the app remains responsive (there is no telling how long a call will take to return).

The other difference is that you have to tell the service what type of call you're making—are you trying to get information or sending information, and if you're sending information, are you just wanting to update information already there, or is it new, or are you even wanting to delete information?

In terms of development, these calls are known as GET, POST, PATCH, and DELETE, and they pretty much do what their names imply.

Introducing the Meeting Planner

Meetings can be an absolute pain to organize. People typically can't make the meeting, so a far better idea is to send out a list of possible dates and times along with some basic data on what it's about. Add on top of that the ability to see upcoming meetings, and you have something of value.

■ **Disclaimer** Meeting Planner is broadly based on the popular Doodle web-based meeting-planning application, but with a number of important changes. I do not have access to the source code for Doodle, nor do I have anything to do with the development team.

Planning the Planner

The planner's user interface is conveniently split into three sections:

1. The navigation window (lets you move around the application). This includes the menu.
2. Requesting the meeting
3. Viewing & responding to meeting requests

The Models

Part of the consideration when grabbing any large amount of data is how the data comes in and how it is deserialized. Let's have a look at the `Meeting` model:

```
public class BaseMeeting
{
    public List<Meeting> Meeting { get; set; }
}

public class BaseMeet
{
    public Meeting Meeting { get; set; }
}

public class Meeting : IInterface
{
    public int id { get; set; }

    public int TimeId { get; set; }

    public DateTime MeetingSchedule { get; set; }

    public int IsMyMeeting { get; set; }

    public int IveResponded { get; set; }
}
```

We need to have the `BaseMeet` and `BaseMeeting` classes, as we cannot directly deserialize into `Meeting`, but rather have to go via a base class. There is a reason for this—when data is sent back from the server, it is sent back as a JSON object. These are sometimes referred to as closed objects because there is an open and a close, and they are not sent through as, say, five strings.

We also have the classes that inherit `Iinterface`—what is that for?

For that we need to look at the database helper. While not required for anywhere that data is inserted (those types are strongly typed), when we take data from the database, we use a generic type `T`:

```
public List<T> GetListOfObjects<T>(string id) where T : class, IInterface, new()
        {
            lock (dbLock)
            {
                    sqlcon.Execute(DBClauseSyncOff);
```

```
                var sql = string.Format("SELECT * FROM {0} WHERE id=?",
                GetName(typeof(T).ToString()));
                var data = sqlcon.Query<T>(sql, id);
                return data;
            }
        }
```

As we don't have a type for T, we need to define what T is and where it's from and create a new instance before we can use it. IInterface only includes a single type, which will be in all classes that inherit that interface—int id—and we only have that as a get property.

The Webservices

For this app, we have POST and GET code to implement. As with all code within the framework, the service code is kept apart from anything not to do with webservices. This not only keeps the code cleaner, but makes debugging simpler.

Thankfully, the webservices are all fairly simple. We have essentially three methods: GetData, GetListData, and SendData. The Get methods are essentially the same, but take more arguments:

```
        public static async Task<U> GetListData<U>(string method)
        {
            var url = string.Format("{0}/{1}", Constants.WorkplaceUrl, method);
            U dta = default(U);
            try
            {
                using (var client = new HttpClient())
                {
                    var response = await client.GetAsync(url);
                    var departmentsJson = response.Content.ReadAsStringAsync().Result;
                    dta = JsonConvert.DeserializeObject<U>(departmentsJson);
                }
            }
            catch (Exception ex)
            {
#if DEBUG
                Debug.WriteLine("Exception in GetListData {0}-{1}", ex.Message,
                ex.InnerException);
#endif
            }

            return dta;
        }

        public static async Task<bool> SendData(string method, string json)
        {
            var success = false;
            var url = string.Format("{0}/{1}", Constants.WorkplaceUrl, method);
            HttpResponseMessage response = null;
            using (var client = new HttpClient())
            {
                var content = new StringContent(json, Encoding.UTF8, "application/json");
```

```
            response = await client.PostAsync(url, content);
            success = response.IsSuccessStatusCode;
        }
        return success;
    }
```

The code is very simple, but within the simplicity, we have a major problem, and it's not with the data sending.

Async Versus Sync

As has already been said, a responsive app makes use of asynchronous calls. This is both a blessing and a big pain.

Synchronous Calls

Let's consider the following piece of code:

```
void myMethodA()
{
    var myVar = myMethodB(10,20,30);
    Console.WriteLine($"myVar = {myVar}");
}
double myMethodB(int a, int b, int c)
{
    double total = a + b;
    for(var i = a; i < b; ++a)
        total += (c - b) / (a + i);
}
```

In terms of how this flows, `myVar` calls `myMethodB`, which then does some math that is returned to the variable and outputted. See Figure 6-1.

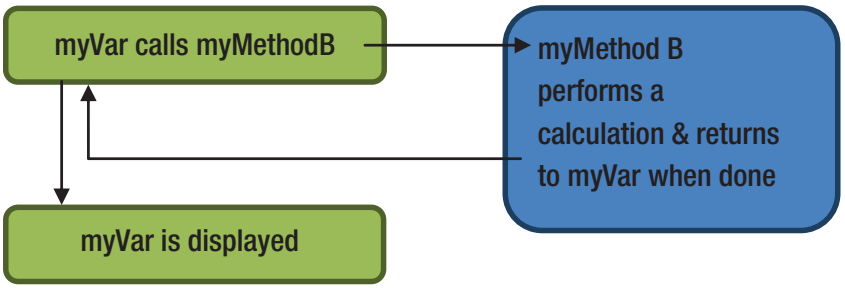


Figure 6-1. *Synchronous call flow*

The `Console.WriteLine` is guaranteed that the data in `myVar` is there. For something simple, this sort of code is perfectly acceptable. We can say that the *CONTROL* of `myMethodA` has been passed to `myMethodB` until `myMethodB` is complete, and then it is *RETURNED* to `myMethodA`. Typically, this will be performed on the same thread.

Asynchronous Calls

An asynchronous call works in the same way as a synchronous call, with the control from one method being passed to a second method. The difference though is that the asynchronous call doesn't wait for the second method to return. Consider the following:

```
double myVar {get;set;}
async Task myMethodA()
{
    await myMethodB(10,20,30);
    Console.WriteLine($"myVar = {myVar}");
}
async Task myMethodB(int a, int b, int c)
{
    double total = a + b;
    for(var i = a; i < b; ++a)
        total += (c - b) / (a + i);
    myVar = total;
}
```

We can see that `myMethodA` has been marked as asynchronous (`async`) and that it has a return type of `Task`. When `Task` is by itself, it can be considered as being the `async` equivalent of `void`. If there is something in a set of angle braces (< and >), then it can be considered the equivalent of a normal method returning that type (so a method of type `Task<string>` would be the `async` version of `string`). There is an `await` as well in the `myVar` line within the method.

It looks the same, but as soon as the `await` is executed, a new thread is created that runs the second method and, once complete, will return a value. The caller thread carries on to the next line within the method (in this case, a call to `Console.WriteLine`).

The problem is that there is no guarantee that `myVar` will contain anything. On one hand, we have a fast and responsive app (which is great), but on the other, the app may have no data (which is bad) when it tries to display a value.

This lack of data becomes more of a problem when calling an external service. On mobile devices, you're at the mercy of the quality and strength of your connection, which may result in a very slow return of data (if indeed any data is returned).

At this point, we need to consider not only how to code defensively, but also the underpinning operating system.

Let Me Introduce You to the Key Difference Between iOS and Everything Else

In the world of mobile technology, there are two differences in how the operating systems work—and it is essentially a "them" and "us" situation. iOS works with something called "Ahead of Time" while every other mobile OS works with "Just in Time." What does that mean?

iOS wants to know what is going to happen before it happens. For example, if the developer wants data on screen, the data is inserted into the `UILabel`, say, and then rendered. This creates a very smooth and professional look and feel. The interface is generated ahead of time. This is a very simple overview of how it works (in reality it is much more complex).

On the other operating systems, the user interface is generated moments before it is displayed. This is both a good and bad thing. It is good in that when we bring data in from a service, the user interface is up to date and can be updated without too much difficulty by changing data. The problem is that there is a delay as the user interface is created, which on older phones can cause a visible lag, and the user might think the app has crashed.

For most applications, this isn't a problem. When an app is dealing with data from a service though, the differences become apparent and more problematic. On iOS, the UI will be placed on screen, but with nothing there. The data comes in from the service after the UI has been rendered. This means the phone doesn't know what will be there, so it is possible that the rendering won't work as it should and, worse, that the user experience won't be as good.

This doesn't have to be such a problem, however, as MVVM Light allows the UI to be updated using the `WhenSourceChange` extension.

WhenSourceChange

When a property is altered, it can be set to generate a `RaisePropertyChanged` event either via the true value using `Set(()=>)` or directly. When `RaisePropertyChanged` is triggered, `WhenSourceChange` is triggered too. When this happens, the code can affect the UI element or call another method.

For example:

```
this.SetBinding(()=>ViewModel.MyProperty, BindingMode.TwoWay).WhenSourceChange(DoSomething);
```

will call a method called `DoSomething`. It is not bound to any UI object, so the `DoSomething` method will need alter the UI.

`WhenSourceChange` can also have a lambda expression:

```
this.SetBinding(()=>ViewModel.MyProperty, BindingMode.TwoWay).WhenSourceChange(()=>lblSay
Hello.Visibility = ViewModel.MyOtherProperty == "foo" ? ViewStates.Invisible : ViewStates.
Visible;);
```

Again, though, the binding is not set to a particular UI element. Is it possible to bind to a UI element and use `WhenSourceChange` to alter a different UI element based on it?

Would this work?

```
void BindUI()
{
    this.SetBinding(
        () => ViewModel.TxtEdit,
        () => EditData.Text,
        BindingMode.TwoWay).WhenSourceChanges(() => TxtResult.Text = ViewModel.TxtResult);
}
```

Sure, it will compile, but on execution it will throw an exception, as you cannot both set the `targetPropertyExpression` and call `WhenSourceChange`. It will also throw an exception if you alter the lambda to a method. As we can't directly bind from one UI element to another, the `WhenSourceChange` extension can be thought of as something similar to `PropertyChanged`, except that `PropertyName` is bound to a property.

Ensuring the User Interface Updates

Both of our user interfaces will have some form of list within them. To ensure the UI works, let's consider the flow (Figure 6-2).

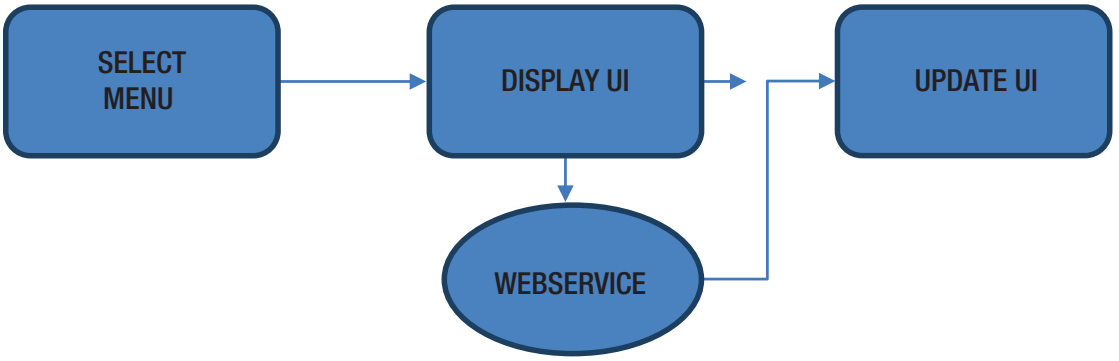


Figure 6-2. *Flow of the UI*

Why would the webservice be called when the UI is displayed rather than on the button click on the select menu? Moreover, why don't we just do the grab when we are on the select menu?

Grabbing the data while we're on the menu is wasteful. If we don't use a facility, then grabbing data for that particular facility is pointless; guessing what the user will do is never a good idea.

Performing the data grab when the button is clicked will work, but there is a chance of something known as a *race condition* occurring—the data returns before the UI has been created, causing the UI to become confused.

The speed difference between firing the data grab-off when the UI is displayed and doing so when the button is clicked is virtually nothing, so to avoid the possibility of the UI becoming confused, it is better to do it on the UI view itself.

The Connectivity Service

As we are considering the outside world, we must ensure that there is a connection. If there is not a connection, the data service will sit there, and the UI will never update. Worse is that it is possible the data service will cause an exception, crashing the app, if the service has not been trapped.

There is a NuGet package that makes life less painful: `xam.plugin.connectivity`. This provides a handy cross-platform library to check for different types of connections (such as mobile or Wi-Fi). This can be used with all Xamarin-supported platforms, but it should remain on the platform rather than in the framework code. To access the platform state, we create a service.

Connecting the Service: The Interface

Let's start by creating the interface with the framework to check on the service status:

```
namespace meetingplanner
{
    public interface IConnectivity
    {
        bool IsConnected { get;}
    }
}
```

Within each project, install from NuGet the `xam.plugin.connectivity` package.

Connecting the Service

Now that we've defined the interface, we need to connect, register, and write the service. Thankfully, the code is the same on all platforms, as is the method of registration:

```
public class Connectivity : IConnectivity
    {
        public bool IsConnected
        {
            get { return CrossConnectivity.Current.IsConnected; }
        }
    }
```

And to register:

```
SimpleIoc.Default.Register<IConnectivity, Connectivity>();
```

While the connectivity plugin also allows for testing the type of connectivity and various other connectivity options, the basic need is to know if a connection is there. We don't need to know if the connectivity has changed, as before any form of outside connection is attempted the connectivity property is first checked.

UI Considerations

Other than the synchronization between the webservice and the UI, the UI itself also needs to be considered, since part of the code is reliant on lists or tables. The only real difference between working with and without MVVM Light is that the source comes from the view model rather than anywhere else. Other than that, it's the same.

Defensive Programming

Defensive programming effectively means that you're planning for the worst; in this case, it is that there is no data in the property being accessed.

When it comes to an `async` call, we have a couple of tools in our arsenal. These are `ContinueWith`, `ConfigureAwait`, and `RunSynchronous` (there are others, but for this chapter, these are enough).

ConfigureAwait (true|false)

This is usually found within the UI, and if set to true it will cause the UI to wait until the `async` call has returned. This doesn't mean that the data required is there, but helps. Remember, if the code being called also contains an `await`, control will carry on to the next line, with the data being returned at some indeterminate point in the future.

RunSynchronous

Runs the called `async` method as if it were a synchronous method with all the advantages and disadvantages this provides.

ContinueWith

This is possibly the most useful of the arsenal, as it is called after the `await` has finished, with data being returned in `Result`. Therefore, if we had the following:

```
async Task myMethodA()
{
    await myMethodB(10,20,30).ContinueWith((t)=>
    {
        if (t.IsCompleted)
            Console.WriteLine($"myVar = {t.Result}");
        else
        if (t.IsFaulted || t.IsCancelled)
            Console.WriteLine("Failed to work");
    });
    Console.WriteLine("Hi folks!");
}
double myMethodB(int a, int b, int c)
{
    double total = a + b;
    for(var i = a; i < b; ++a)
        total += (c - b) / (a + i);
    return total;
}
```

the new thread would be created when the `await` was called and control would then be passed to the next line within `myMethodA`, but when `myMethodB` was completed, the code within the `ContinueWith` would be executed. The code is coincidentally more or less the same as the original synchronous version now.

In terms of a webservice, this is very useful—we have a guarantee that as long as the caller isn't cancelled or returned with a faulted state, data will be `Result`. We still need to be on guard, as there is nothing to say that what is returned in `Result` isn't `null`, so trying to perform any sort of manipulation on this will result in a crash if not surrounded by a `try/catch` structure.

Let's Have a Look at the App in Action

The images in Figure 6-3 are the app in action.

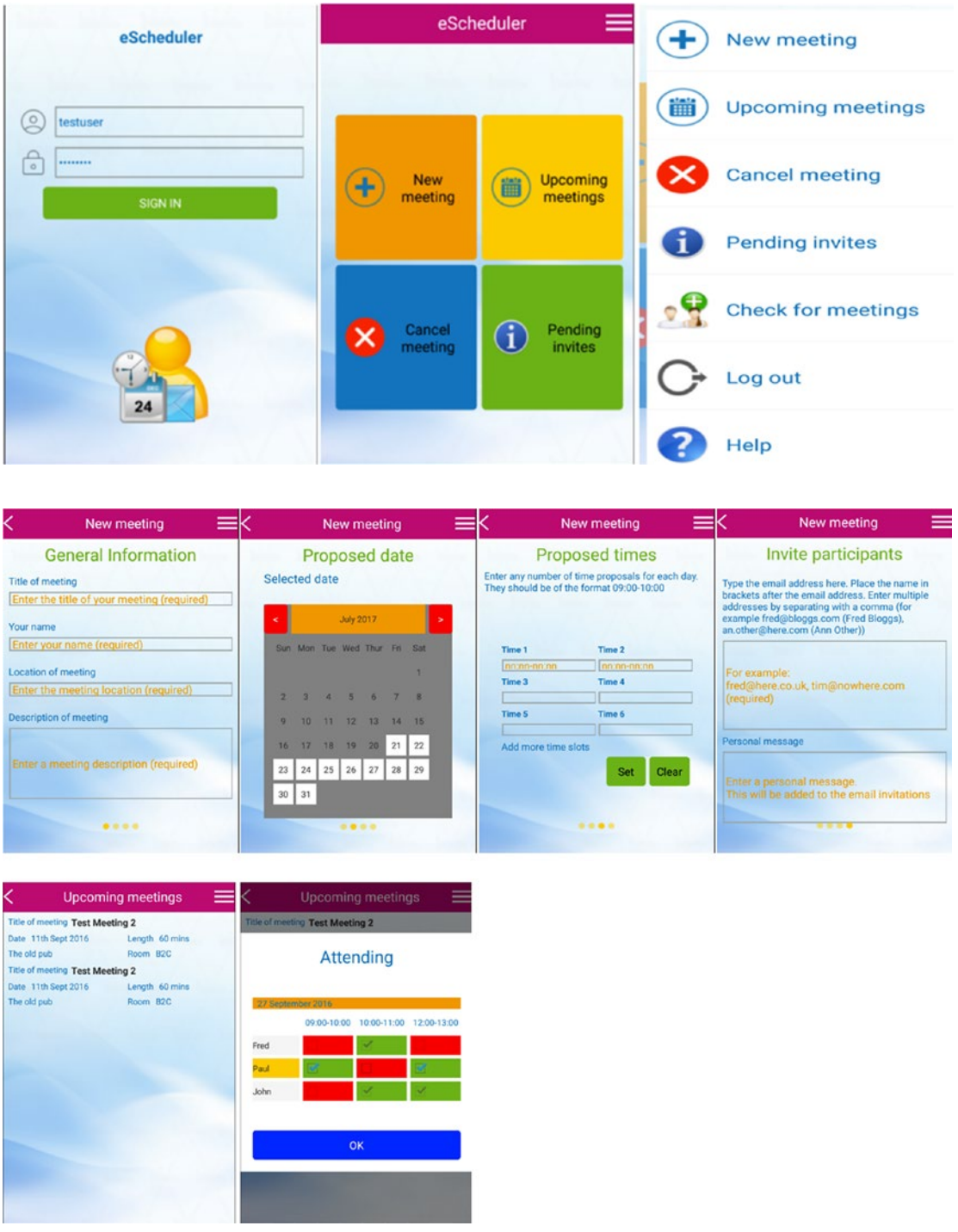


Figure 6-3. *The app in action*

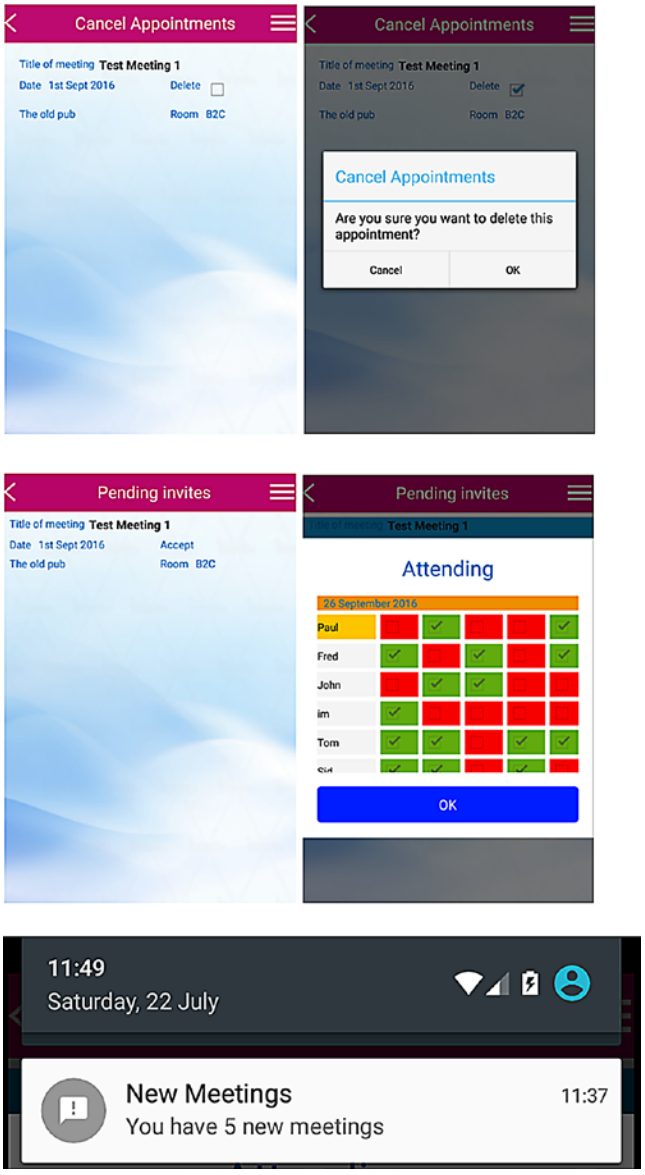


Figure 6-3. (*continued*)

We have the user interface in place, and the majority of the code within the MVVM framework doesn't have anything out of the ordinary. We create the webservice as we would any other service. For example:

```
public interface IWebApi
{
    Task<PollData> GetPollingData(int meetingId);
}
public class WebApi : IWebApi
{
     public async Task<PollData> GetPollingData(int meetingId)
     { ... }
}
```

And in the `ViewModelLocator.cs`, the service is registered like this:

```
SimpleIoc.Default.Register<IWebApi, WebApi>();
```

The SQLite code is the same as has been seen in the other chapters, so what is so important in this version that wasn't there in the other apps we have seen so far?

Push Notifications

Can we handle push from within the framework? We can, but push is not as simple as you may first think. Push works when an app is awake, but also when it is asleep. It is the asleep mode that requires the thought. Let's consider how push works. I'm not going to concern myself with the server code.

The app receives the notification and, depending on if the platform is handled by a service or a method, a notification alert is generated (depending on if the app is told to produce one or not). Here, we diverge.

The app has two states: awake and asleep. When the app is asleep, it is not supposed to take over anything other than for a very simple notification. The point of this is to ensure that apps in the foreground aren't slowed down or otherwise interfered with (consider getting a text message; you get the ping and a badge, but that's it). The app may do something in the background, but it has to be fast.

When the app is awake and the notification arrives, the same notification will show, but now there isn't a problem grabbing the data and doing what needs to be done.

As with anything, though, things aren't one way or the other; apps also have a waking state and a going-to-sleep state. Here, the app isn't asleep and isn't awake, so how are notifications dealt with? Simply put, they're not—anything that isn't awake is counted as asleep. However, during that time, the app can do something useful, like store when it fell asleep. Then, during waking, it can perform a data grab covering the sleep time to the awake time (`DateTime.Now`, essentially).

We can create an interface that we register just as any other service. If desired, we can grab the data or just store in a database table that metadata the notification gives; the choice of which is up to you. All you need to consider is the type of data you're wanting to get and what you want to do with it. For example, if you receive a message saying that there is new data on the database for a new meeting, then you may decide to grab that data only when requested. If, however, the message is of greater urgency (such as all meetings for a date have been cancelled or you need to travel to Australia), it will be needed as soon as the app wakes, so that data will be required instantly; grabbing it before the UI wakes is therefore necessary.

Do We Need to Access the Outside World?

We have a number of entries within our database and have had no push notifications to say there is new data. Do we actually need to access the service?

The simple answer is yes. There is no guarantee that a push notification will be received. However, if the service allows it, we can ask for the last update to that table for you. If it corresponds to what we have internally, nothing needs to be done. This will most likely be much quicker than grabbing all of the data for a date range, and we only need act if there is a change.

We also need to think about data integrity. A database that is populated with the same data will soon end up a mess and, moreover, it will take quite a while for the data to be saved to the database (depending on size).

Let's consider the following trivial code:

```
public DateTime LastUsed {get;set;}
async Task GetDataGrab
{
     var dataTables = repoService.GetListData<foo>();
     await webService.CheckForNewData(LastUsed).ContinueWith(async(t)=>
     {
         if (t.IsCompleted)
         {
             if (!t.IsFaulted && !t.IsCancelled)
             {
                  if (t.Result != 0) // there is data
                  {
                      await webService.GetNewData(LastUsed).ContinueWith((w)=>
                      {
                          if (w.IsCompleted)
                          {
                              if(!w.IsFaulted && !w.IsCancelled)
                              {
                                    // store data
                              }
                          }
                      });
                  }
             }
         }
    });
}
```

We have created a chain of async calls. As long as we receive that the task has not faulted, we can proceed to the next call.

■ **Note** It isn't a good idea to chain too many calls. While both iOS and UWP can handle them without an issue, Android will become more unstable with increasing numbers of `async` chained calls. However, as long as you don't exceed four layers, you should be fine.

We stared the method off by getting the data from the database. We already know that we only want to store the data we don't already have. While the service should only return from the last time checked, we're going to assume (for argument's sake) that the phone has lost the setting (this is not an unreasonable assumption), so we do a complete grab. We already had 20 entries, but the service returned 25, so we need to only store the ones we don't have.

This is thankfully quite simple using LINQ:

```
var items = t.Result;
var diff = items.Where(w => !dataTables.Any(p => p.Id == w.Id)).ToList();
repoService.SaveData(diff);
```

If we include the condition

```
if (!items.Count != dataTables.Count)
{
     ...
}
```

we can further speed up the process. While on the first run there are five new entries, on the next six runs there may be no changes, so there is no need to run the LINQ query.

How Can We Notify the UI?

All of this data processing has occurred within the UI, so how can we notify the UI? We've already seen that MVVM Light provides the very useful `WhenSourceChange` extension, but we need to do something before this is picked up.

The `WhenSourceChange` extension listens out for the `PropertyChanged` event behind the scenes; therefore, we have to either manually call `RaisePropertyChanged` within the view model or, if we use the `Set(()=>...)` form, set the broadcast variable to true.

This is fine for something simple (such as an `int`, `string`, or any other form of primitive)—if we change the value, the event is fired off. However, if we have a `List<T>` and change a property within `T` and save, the event isn't fired. This is because `T` itself hasn't changed, but rather a property within `T` has. If you think of `T` as a box with five smaller boxes inside it, and if we change the color of one of the smaller boxes, the outer box itself hasn't changed. If we add (or remove) more instances of `T`, the event will be fired.

To inform the UI that a property has changed, we have one of two choices: either we completely remove the instance of `T` that had the property in it, then insert the new instance of `T` with the modified property, or we create something known as a sacrificial property.

The Sacrificial Property

This is typically a Boolean. It has no function other than to notify the UI of a change in a property. We would have something like this:

(in the view model)

```
bool listHasChanged;
public bool ListHasChanged
{
     get => listHasChanged;
     set {Set(()=>ListHasChanged, ref listHasChanged, value, true);}
}

List<foo> myList;
public List<foo> MyList
{
     get => myList;
```

```
     set {Set(()=>MyList, ref myList, value, true);}
}

void SomeChangerMethod(int m)
{
     var t = MyList;
     t[m].SomeProperty = 42; // was 32
     MyList = t;
     ListHasChanged = !ListHasChanged;
}
```

(in the UI)

```
this.SetBinding(
()=>ViewModel.ListHasChanged,
BindingMode.TwoWay).WhenSourceChanges(RecreateTable);
```

When `ListHasChanged`, the `RecreateTable` method is called. Note that we cannot pass parameters to `RecreateTable`. This is because `WhenSourceChange` is used, so we can either alter something directly or call an action (which is what is happening here).

Another important aspect of the sacrificial property is that we don't care what the value is; we are just using it to inform the UI of a change.

Conclusion

With this in place, the app becomes quite a simple affair. Feel free to examine the code supplied to see how it works. There are plenty of comments surrounding the source explaining what is happening.

CHAPTER 7

Unit Testing

Unit testing is often omitted from the development cycle of a mobile application (unless you are working in a large team, typically). If you are not in a large team but are essentially the equivalent of the 1980s bedroom coder, why should you unit test when it's quicker to deploy to a device and see first-hand if something fails?

The simple answer is that it is usually very difficult to find what has gone wrong, and as the code base gets larger the ability to quickly find the issue decreases significantly. Take, for example, the following: I have a database-driven application. I add code to a user interface, which is inserted into my SQLite database. There is a button that pulls the data back and inserts it into a list. I enter 20 items, and none of them show, but no exceptions are thrown. Where is the error?

Say you fix the problem. There is nothing to say the fix doesn't break the code elsewhere, and possibly it's somewhere you don't expect it to break, as in the past it's always been rock solid and always worked.

Would it not be better to have a set of automated testing routines that could weed out that problem without the need to build, deploy, and then set myriad break points with random `Debug.WriteLines` scattered around the codebase? What if each automated test was essentially a small program that returned a pass or fail and only tested a single objective? Like the sound of it? Well, then, unit testing is for you!

Welcome! Let's make our apps, big and small, far more stable—and not at the cost of development time. Unit testing has all the advantages of stability without the time hit.

I must thank all at Primordial Radio (`www.primordialradio.com`) for allowing me to use the unit tests used within their apps currently in development.

A Quick Introduction

Unit tests will not debug your applications, nor will they fix bugs, but the more of them you write, the quicker finding problems and solutions becomes. They will also show when a fix in one part of the code breaks another part of the code.

There are plenty of unit-testing frameworks available (such as nunit, xunit, and moq). There are two flavors of unit test: platform and PCL. Platform does not limit itself to the user interface; it can also concern itself with anything purely related to the platform (such as user settings, the camera, or making a call). Unit tests can also be run automatically as part of a continuous integration pattern, but that is outside of the scope of this book.

Let's Make a Start and Add a Unit-Testing Project

In the source code folder for this chapter, you'll find a completely empty project (with the exception of having the framework set up) called `UnitTesting`. Using that, we will add unit tests for both the framework and the platform.

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_7

Adding the nunit Test Library

Figures 7-1 through 7-20 are from VisualStudio for Mac. Essentially, they are the same as if you were using VisualStudio (or VisualStudio for Mac). First, select the head project name in the Solution Explorer (Figure 7-1).

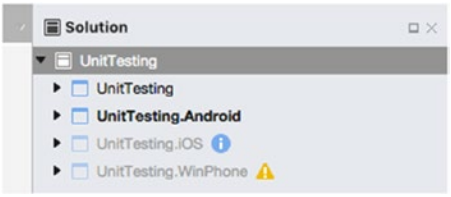


Figure 7-1. *Select head project name*

Right-click and navigate to Add New Project in the context menu. You will be presented with a new window similar to that shown in Figure 7-2.

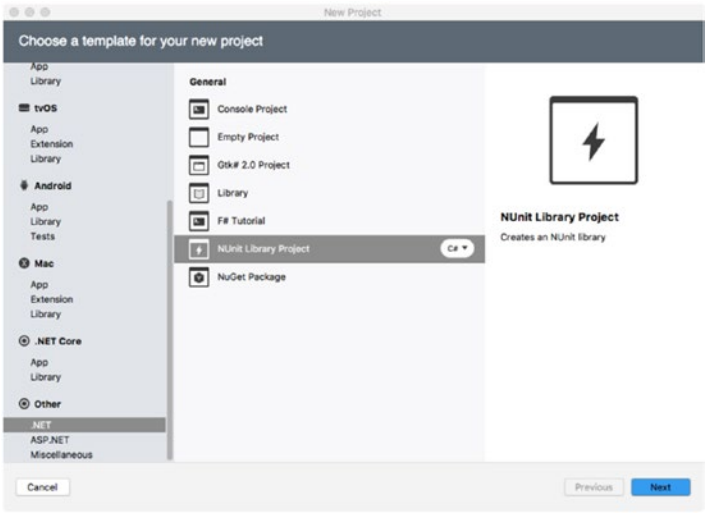


Figure 7-2. *Choose a template*

In the menu at the left-hand side, find the Other section. As this is not a user interface project, we need to select .NET as the project type. In the middle, select NUnit Library Project. You should change the language specifier from C# to F# if your code is in F#. You don't have to, but it makes sense to.

Click *Next*. The window will close and you will be asked to enter a project name (as in Figure 7-3).

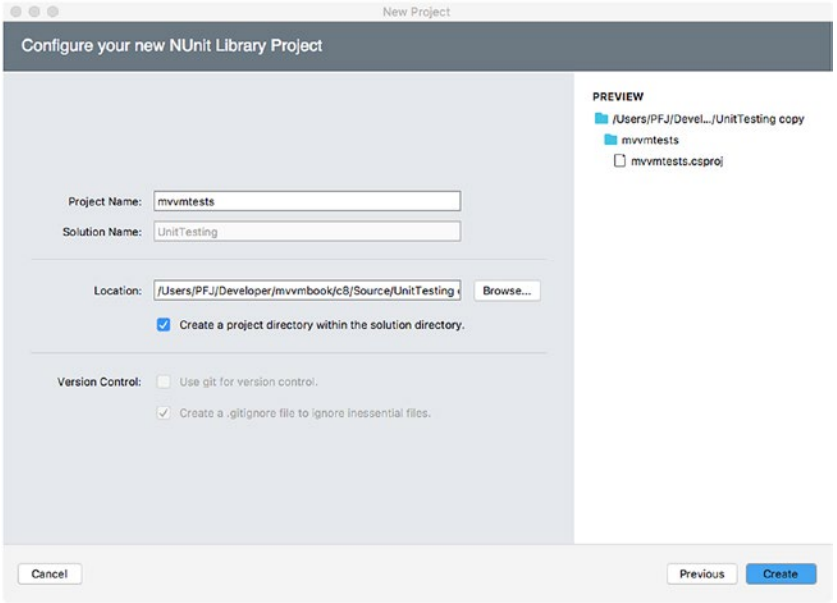


Figure 7-3. *Enter a project name*

When you have entered a name, click *Create*. Note that you cannot change the version control, as it is inherited from the master project settings.

Once created, the Solution Explorer will look like Figure 7-4.

Figure 7-4. *Solution Explorer*

The final step is to add a reference to the MVVM framework. In the case of this project, it's called `UnitTesting`. See Figure 7-5.

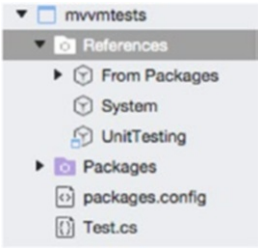


Figure 7-5. *Add a reference to the MVVVM framework*

The unit test library has now been added, and we can start adding tests. Let's take a moment to understand the makeup of a unit test.

The Anatomy of a Unit Test

If you open the `Test.cs` file in the root of the `mvvmtests` project, you will see the following auto-generated code:

```
using NUnit.Framework;
namespace mvvmtests
{
    [TestFixture()]
    public class Test
    {
        [Test()]
        public void TestCase()
        {
        }
    }
}
```

All unit tests start with `[TextFixture()]` before the class. It defines the code that follows is a unit-testing fixture. The `[Test()]` can therefore be considered a fitting within the fixture (if that sort of analogy works for you).

In actuality, the `[Test()]` is the beginning of that specific unit test. The code itself follows standard code rules (such as having a return type, method parameters, and so forth).

In its simplest form, this is a unit test, but it doesn't do anything. Let's create a slightly more meaningful unit test.

A Simple Addition Unit Test

Our unit test is going to check if two parameters passed in equal seven. To do this, we use the `Assert` function. If the condition is true, the assert passes. For example:

```
Assert.That(answer, Is.EqualTo(a+b));
```

would result in a `true` result. Note that the preceding line is a very "English" way of writing. We are saying that the answer should be equal to whatever is in the brace.

Our test is this:

```
[Test]
public void TestAddition()
{
     var answer = 7;
     Assert.That(answer, Is.EqualTo(3 + 4));
}
```

We can run the unit test by selecting Run Unit Tests from the Run menu. See Figure 7-6.

Figure 7-6. *Run the unit test*

Once the tests are run, the output is shown in the Test Results panel (Figure 7-7).

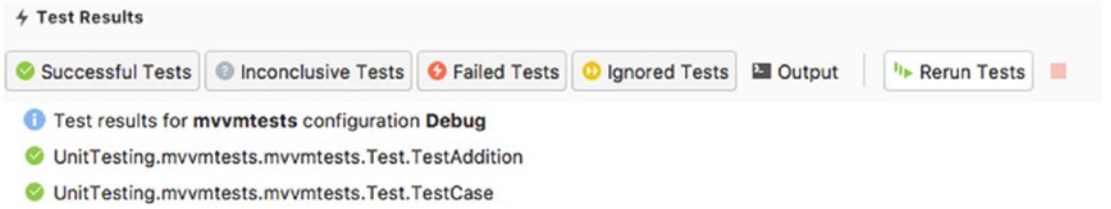


Figure 7-7. *Test Results panel*

It should come as no real surprise that the test passed. We have essentially told the unit test to run a test that proves 7 = 7.

Test Versus Test()

You will have no doubt noticed that the word *Test* in the square braces (known as the attribute) for the second unit test had the braces omitted. Is there a difference between `[Test]` and `[Test()]`?

The answer is, for examples, no. There is no difference. We are not passing parameters into the test, so for the sake of argument, `Test` and `Test()` here are the same.

Passing in Parameters to a Test

When they're *not* the same is if we add parameters into the test. Let's alter our test so that we are passing in two values that we will multiply and divide by 3. If the answer has no remainder, `Assert` will pass. We are using `TestCase` here rather than `Test`, as the `Test` constructor doesn't have an overload taking two values.

```
[TestCase(3, 7)]
public void TestMultiplyDivide(int a, int b)
{
     var answer = ((a * b) / 3) % 3;
     Assert.That(answer, Is.EqualTo(0));
}
```

A quick look at this and we see 3 * 7 = 21 / 3 = 7 % 3 = 1, so the `Assert` will fail (which indeed it does). The results will also give you the reason for the fail (Figure 7-8).

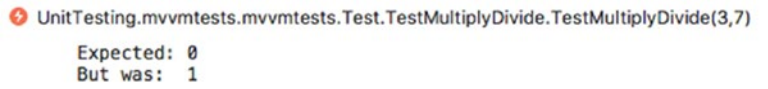


Figure 7-8. *Test results*

Running Multiple Tests at Once on the Same Test Case

One of the beauties of unit tests is that you don't need to keep writing the same code over again to test different parameters. You can just create a list of tests. The `TestMultiplyDivide` method will be slightly modified, and multiple tests will be run through it:

```
[TestCase(3, 7)]
[TestCase(2, 4)]
[TestCase(5, 4)]
[TestCase(8, 8)]
[TestCase(8, 7)]
public void TestMultiplyDivideT2(int a, int b)
{
     var answer = ((a * b) / 4) % 2;
     Assert.That(answer, Is.EqualTo(0));
}
```

When we run this code, we will see five sets of results (Figure 7-9).

Figure 7-9. *Five sets of results*

Expanding the failures shows that, as before, 0 was expected and 1 was found.

Testing the Result of a Second Method

In reality, it is rare that methods act in isolation. It is more usual that Method A passes something to Method B, which returns a value back. We can perform this form of test as well:

```
[Test]
public void TestMultipleMethods()
{
     var retVal = SecondMethod(3, 6, 8, "calculator");
     Assert.That(retVal, Is.Not.EqualTo(-1));
}
int SecondMethod(int a, int b, int c, string option)
{
     int rv = -1;
     if (option == "calculator")
          rv = a * b + c;
     return rv;
}
```

This will pass, as the fail condition only occurs if `option` is not `calculator`. We can prove this by adding another call to SecondMethod where `option` is something else:

```
[Test]
public void TestMultipleMethods()
{
     var retVal = SecondMethod(3, 6, 8, "calculator");
     Assert.That(retVal, Is.Not.EqualTo(-1));

     retVal = SecondMethod(3, 6, 8, "quackulator");
     Assert.That(retVal, Is.Not.EqualTo(-1));
}
```

When we run it, the test results show a fail, with the stack trace showing the `Assert` that failed (Figure 7-10).

Figure 7-10. *Test results show a fail*

Let's Get Real

While these simple tests are fine, we need to really be testing the code from the framework. As we've seen in other chapters, the framework does the vast majority of what is colloquially described as the "the grunt work" (in other words, the heavy lifting). Typically, we have databases, web services, helper methods, interfaces, services, models, view models, and most likely a lot of other code as well.

We can (and should) test them all. For this book though, it's not going to be possible to demonstrate how to test absolutely everything, so I will limit the discussion to databases, webservices, and view models.

Database Testing

There is a golden rule when it comes to unit testing a database system: never touch the "live" database (the database the app uses). Instead, the unit test should use a database purely stored in memory.

In the source directory is a project called `SQLiteUnitTest`. This has everything needed to test a database.

Setting Up the Test

When testing the database system, not only do the database connection and helper methods need to be tested, but also the data models the database works on.

To create the new database instance, we use the following lines:

```
ISQLConnection factory;
string testDatabase = ":memory:";
SQLiteConnection connection;
ISQLitePlatform platform { get { return new SQLitePlatformGeneric(); } }
```

We next need to add in a couple of testing attributes that are used for setting up, closing down, starting the transaction, and rewinding a transaction:

```
[OneTimeSetUp]
public void OneTimeSetUp()
{
     factory = Substitute.For<ISQLConnection>();
     connection = new SQLiteConnection(platform, testDatabase);
     factory.GetConnection().Returns(connection);
}
[SetUp]
public void Setup()
{
     connection.BeginTransaction();
}
[TearDown]
public void TearDown()
{
     connection.Rollback();
}
[OneTimeTearDown]
public void OneTimeTearDown()
{
     connection.Close();
}
```

`Substitute` requires the `NSubstitute` NuGet package to be installed. This can be considered (here) as a helper package. Here, it effectively creates an instance of an interface with the methods in tow.

These four tester methods start the ball rolling, and we can now construct our tests. We don't even need to run the methods, as they are automatically called.

Testing a Simple Store

Our framework contains a single model in the database: `DataEntry`. We have to first test if the database is able to store a card. This test is a two-part test. The first part is when we store some random data; the second is when we call it back and test that there was only one table stored. See the following code:

```
[Test]
public void TestSaveData()
{
     // create an instance of the repository
     var repo = new SqLiteRepository(factory);
     // save with random data
     repo.SaveData(new DataEntry{Id = 101, Password="hello", RandomNumber = new Random().
     Next(255), Today = DateTime.Today, Username="paul"});

     // recall the data
     var count = connection.Table<DataEntry>().Count(sc => sc.Id == 101);
     // perform test
     Assert.AreEqual(1, count);
}
```

All being well, the database will return that the test worked. However, see the results in Figure 7-11.

Figure 7-11. *Results*

The debug information is clear; the `DataEntry` table has not been created. There is a method in the `DBHelper.cs` file called `CreateTables`. In there should be the line

```
Connection.CreateTable<DataEntry>();
```

As the exception says, the table is missing, and it is very likely that the line is missing. Add it in and re-run the test.

This time, the test completes without any errors.

We can run this test as many times as we wish, and each time the test will pass.

Our next test is to ensure we can store multiple data sets:

```
[Test]
public void TestMultipleDataSets()
{
     var repo = new SqLiteRepository(factory);
     repo.SaveData(new DataEntry {Id = 101, Password = "hello", RandomNumber = new Random().
     Next(255), Today = DateTime.Today, Username = "paul"});
     repo.SaveData(new DataEntry { Id = 102, Password = "apple", RandomNumber = new
     Random().Next(255), Today = DateTime.Today, Username = "mac" });

     var data = repo.GetList<DataEntry>();

     Assert.AreEqual(2, data.Count);
     Assert.AreEqual(101, data.First().Id);
     Assert.AreEqual("paul", data.First().Username);
     Assert.AreEqual(102, data.Skip(1).First().Id);
     Assert.AreEqual("mac", data.Skip(1).First().Username);
}
```

This test runs to completion.

We continue testing the database helper. This time, we need to check the delete method:

```
[Test]
public void TestDeleteRecord()
{
     var repo = new SqLiteRepository(factory);
     var deleteData = new DataEntry { Id = 101, Password = "hello", RandomNumber = new
Random().Next(255), Today = DateTime.Today, Username = "paul" };
```

```
    repo.SaveData(deleteData);
    repo.SaveData(new DataEntry { Id = 102, Password = "apple", RandomNumber = new
    Random().Next(255), Today = DateTime.Today, Username = "mac" });

    // delete the first entry
    repo.Delete(deleteData);

    // tests
    var data = repo.GetList<DataEntry>();
    Assert.AreEqual(1, data.Count);
    Assert.AreEqual("mac", data.First().Username);
    Assert.AreEqual(DateTime.Today, data.First().Today);
}
```

This time, we get a fail (Figure 7-12).

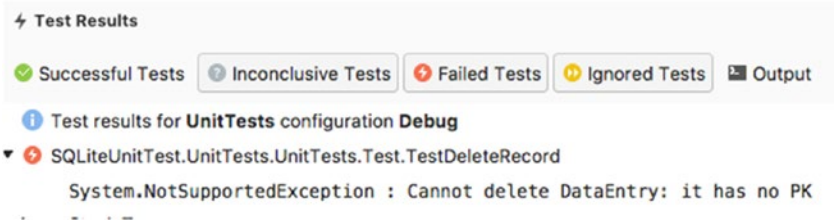


Figure 7-12. *The test fails*

What does this error mean though?

PK stands for PrimaryKey. This is an error in the model itself. To fix, add the `[PrimaryKey]` attribute above the `Id` property attribute and retest. It now passes.

The final test is to retrieve data. We already performed this when we stored data, but we also now explicitly test the retrieval of data:

```
[Test]
public void TestRetrieveData()
{
    var repo = new SqLiteRepository(factory);
    repo.SaveData(new DataEntry { Id = 101, Password = "hello", RandomNumber = new
    Random().Next(255), Today = DateTime.Today, Username = "paul" });
    repo.SaveData(new DataEntry { Id = 102, Password = "apple", RandomNumber = new
    Random().Next(255), Today = DateTime.Today, Username = "mac" });
    repo.SaveData(new DataEntry { Id = 103, Password = "elephant", RandomNumber = new
    Random().Next(255), Today = DateTime.Today, Username = "frog" });
    repo.SaveData(new DataEntry { Id = 104, Password = "giraffe", RandomNumber = new
    Random().Next(255), Today = DateTime.Today, Username = "becki" });

    var data = repo.GetList<DataEntry>();
    Assert.AreEqual(4, data.Count);
```

```
     var first = data.First();
     Assert.AreEqual(101, first.Id);
     Assert.AreEqual(5, first.Password.Length);
     Assert.AreEqual(DateTime.Today, first.Today);
     Assert.AreEqual("elephant", data.Skip(2).First().Password);
     Assert.AreNotEqual("giraffe", data.Skip(2).First().Password);
}
```

The final `Assert` test ensures that the data being tested is not giraffe rather (which it isn't).
At the bottom of the model, we have the following lines:

```
public override string ToString()
{
     return string.Format("[DataEntry: Username={0}, Password={1}, Today={2}, Id={3},
     RandomNumber={4}]", Username, Password, Today, Id, RandomNumber);
}
```

We are overriding `ToString` to return the contents of the model itself.
This can also be tested:

```
[TestFixture]
public class DataEntryUnitTest
{
    [Test]
    public void CheckToString()
    {
        var model = new DataEntry
        {
             Username = "paul",
             Password = "olliedog",
             Today = DateTime.Now.Date,
             Id = 1971,
             RandomNumber = 123123
        };

        Assert.AreEqual("[DataEntry: Username=paul, Password=olliedog, Today=12/03/2017
        00:00:00, Id=1971, RandomNumber=123123]", model.ToString());
    }
}
```

Note that for this to actually work correctly, the `Today` value would need to be changed to match the day it was being tested on. It may also need to be modified for your locale.

Testing Online Services

Our final foray into unit testing is testing online services. Many applications are now bound to a backend data service (for example, the Facebook and Twitter apps on your mobile phone are built around data services [held on the Facebook or Twitter servers respectively] that have a user interface built upon them) and so the data coming down (and being returned) also has to be tested.

There are a number of tests that can be performed:

1. Data connection
2. Handling HTTP exceptions
3. Getting data and deserializing
4. Sending data

Data Connection

Before any data connection is made, we have to test for the connection. The problem is that a unit test does not interface with the device, so how can we actually test the network connection?

The first thing to remember is that we are only mocking a call, not actually making the call. We can therefore use `Substitute` as we did for the databases.

Our network interface and client WebAPI interfaces are both in the MVVM framework itself.

Let's see how we would perform a test mocking that the device is connected:

```
[Test]
public void TestCase()
{
     var connection = Substitute.For<IConnection>();
     connection.IsConnected().Returns(true);
}
```

Handling Exceptions

An exception is thrown when the application tries to perform an action but the action fails. Simple examples would be trying to access an array or list from outside of the bounds of the array or list, running out of memory, or receiving a 404 error from a webservice—the list goes on. We can catch exceptions are part of the unit test. Let's start with a very simple example that is guaranteed to work:

```
[TestFixture]
public class Test
{
    [Test]
    [ExpectedException(typeof(ArgumentException))]
    public void ArgumentExceptionTest()
    {
        throw new ArgumentException();
    }
}
```

■ **Note** If you are using NUnit 3, the `ExpectedException` attribute is no longer available. Thankfully, nunit has released the code to implement the attribute. You can find it in the `exception_tests` source.

We know this test will pass, as the `ArgumentException` is explicitly thrown as the only line of code in the test (Figure 7-13).

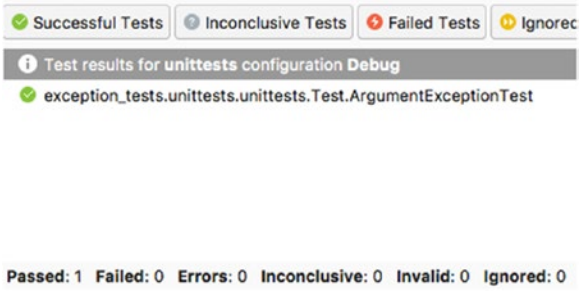


Figure 7-13. *The test passes*

In reality, we would not have this in our code, but we are likely to have the likes of a `NullReferenceException` or `ArgumentOutOfRangeException`.

The following is a very typical way of hitting a `NullReferenceException`. Within our MVVM framework project we have defined an interface, defined the class that inherits the interface, and created the method called.

```
public interface IExceptionTests
{
    void ShowText(string text);
}

public class ExceptionTests : IExceptionTests
{
    void IExceptionTests.ShowText(string text)
    {
        Debug.WriteLine(text);
    }
}
```

We next reference the interface within the unit test and create a test around the method the interface has a stub for:

```
IExceptionTests exceptionTests;

[Test]
[ExpectedException(typeof(NullReferenceException))]
public void NullReferenceExceptionTest()
{
    exceptionTests.ShowText("Hello world");
}
```

When the unit test is run, the `NullReferenceException` is thrown, since while we have defined the stub, we aren't initializing the interface, so `exceptionTests` is `null`. Within MVVM Light, this is a common error. Fixing this is simple:

```
public class MyFixedClass
{
      IExceptionTests exceptionTestService;
      public MyFixedClass(IExceptionTests exTests)
      {
            exceptionTestService = exTests;
      }
}
```

This has been seen many times in any view model that uses a service.

Another frequently generated exception is when the code tries to access something outside of the bounds of a list or array or some other collection type.

```
[TestFixture]
public class AORTests
{
    [Test]
    [ExpectedException(typeof(ArgumentOutOfRangeException))]
    public void AORExceptionTest()
    {
        var testList = CreateTestList().Where(t => t.ToLowerInvariant().Contains("dog")).
        ToList();
        var output = testList[2];

        Debug.WriteLine(output);
    }

    List<string> CreateTestList()
    {
        return new List<string> { "fred", "cats", "TARDIS", "Lord wetman",
        "boiled fish", "thread ogre" };
    }
}
```

Most likely, the `CreateTestList` would be generated by some form of data from a service or database query result.

Testing for online errors requires the `MockHttp` package from NuGet. This is a test layer for `HttpClient` that stubs out the library code, allowing for exceptions. Out test code looks like this:

```
[TestFixture]
public class OnlineExceptions
{
    [Test]
    [ExpectedException(typeof(HttpRequestException))]
    public async Task HRETest()
    {
```

```
        var connection = Substitute.For<IConnection>();
        connection.IsConnected().Returns(true);

        var content = new ContentWithException();
        var mockHttp = new MockHttpMessageHandler();
        mockHttp.Expect("http://www.primordialradio.com/scratchandsniff").Respond(content);

        var client = new HttpClient(mockHttp);

        var response = await client.GetAsync("http://www.primordialradio.com/
        scratchandsniff");

        await response.Content.ReadAsByteArrayAsync();
    }
}

class ContentWithException : HttpContent
{
    protected override Task SerializeToStreamAsync(Stream stream, TransportContext context)
    {
        throw new HttpRequestException();
    }

    protected override bool TryComputeLength(out long length)
    {
        length = 0;
        return false;
    }
}
```

This requires a bit of explanation.

We first create a class to inherit the `HttpContent` base class, which will throw the exception when the code tries to serialize to a stream.

Next, we create a mock HTTP message handler that expects the URL string to respond with some sort of content.

The code then creates the client and awaits the response, then tries to read the response content as a byte array.

Here, there is no data, as a 404 is thrown and the request fails.

Unit Testing the User Interface

It is entirely possible to unit test the user interface, and for this Xamarin provides a specialized unit test that unfortunately only works for Android and iOS. This comes as part of the `Xamarin.UITest` project (which is set when first creating a project) and has two specialized classes: `Xamarin.UITest.Android.AndroidApp` and `Xamarin.UITest.iOS.iOSApp`. Neither of these can be instantiated directly but rather are created via a helper class.

Setting Up Your Project

Setting up a project to include UI unit tests is the same as setting up a project normally, except on the final creation view you will see the item shown in Figure 7-14 at the bottom of the window.

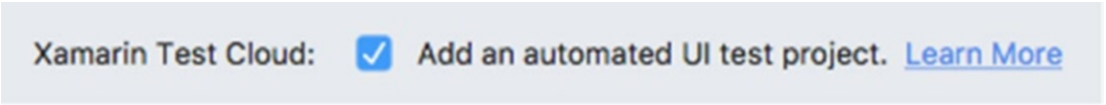


Figure 7-14. *Check the box*

Ensure the checkbox is selected to include the test project.

When the project has been created, you won't see too much difference in the Project Explorer from what you would see with a normal unit test structure (Figure 7-15).

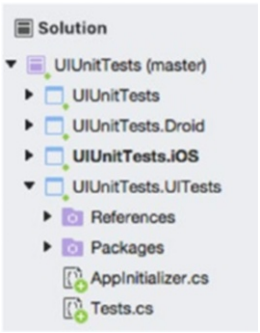


Figure 7-15. *Project Explorer*

Getting Things Working and Tested

Creating the UI unit tests still requires some code to be used by the UI (and then by the test). We can place this either in the platform projects or within the framework project. As we're concerned in this book with the framework, this is where the code will reside.

For our example, the validation is on if the passwords are the same, the password and username are six or more characters in length, and the email address is valid.

Our user interface looks like Figure 7-16.

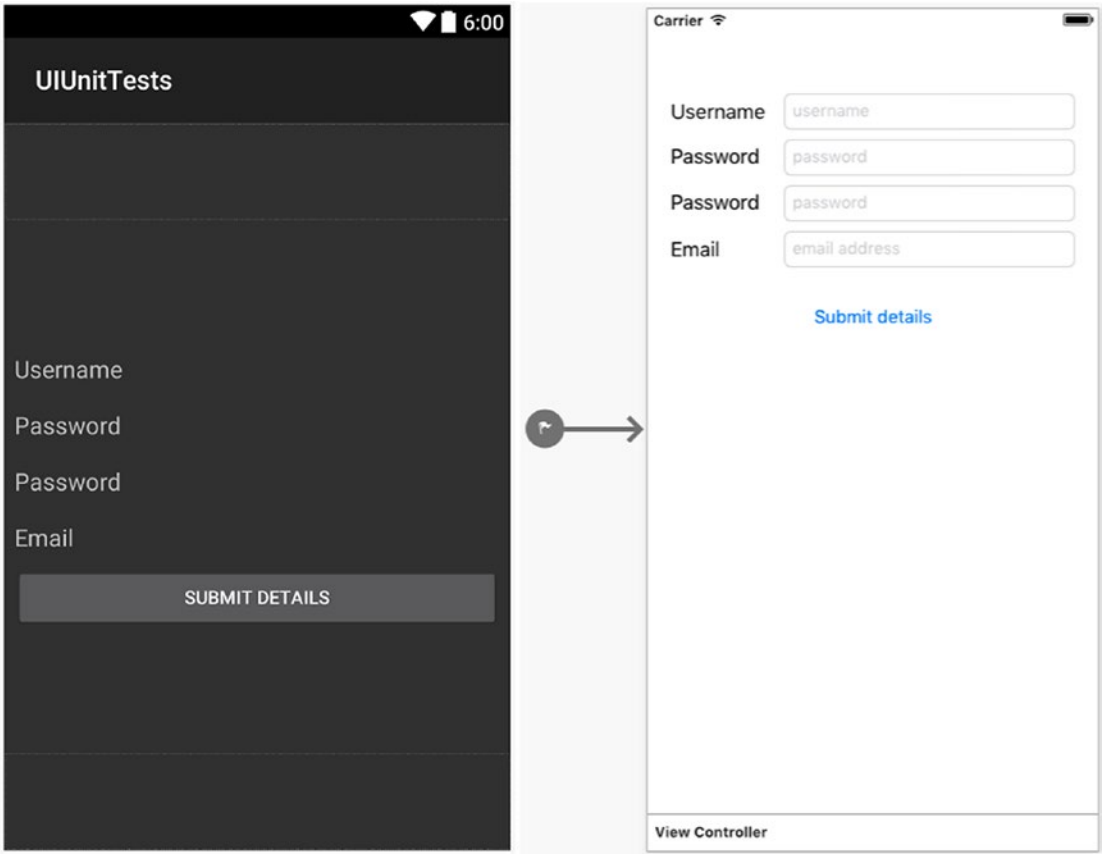


Figure 7-16. *The user interface*

As with any other MVVM application, we wire up the UI (see the source code on this) and create a `RelayCommand` in the view model to perform a partial validation. The `RelayCommand` will set a property called `ErrorMessage`, which generates an event that the UI can intercept and act on.

The difference though is that the UI elements are filled in and the button click is emulated.

The emulation is performed using a number of methods that take `Func<AppQuery, AppQuery>` to locate the view. `Func<AppQuery, AppQuery>` looks like this:

```
app.Tap(c=>c.Marked("SaveButton");
```

For the majority of the time, the methods shown in Table 7-1 for AppQuery will be of most use.

Table 7-1. *Methods for AppQuery*

Method	Description
Button	Locates one or more buttons
Class	Locates views of a specified class
Id	Finds a view with a specific ID
Index	Used within a collection of views of the same type. Returns a single view
Marked	Matches a view with a specified name. On iOS, it uses either the `AccessibilityIdentifier` or `AccessibilityLabel`. On Android, it uses the view `Id`, `ContentDescription`, or `Text` of the view.
Text	Matches views that contains the specified text
TextField	Matches either an `EditText` or `UITextField` object

There are also a number of events associated with testing, shown in Table 7-2 (this is not exhaustive).

Table 7-2. *Events Associated with Testing*

Event	Description
`PressEnter`	Presses the enter key
`Tap`	Simulates a tap on a button
`EnterText`	Enters text into the view. Works differently between Android and iOS. On Android, the text is directly entered. iOS uses the soft keyboard
`WaitForElement`	Pauses the test execution until views appear
`Screenshot(string)`	Takes a screenshot and saves. Returns `FileInfo` about the shot taken
`Flash`	Makes the view flash/flicker on the screen

A complete list can be found here: `https://developer.xamarin.com/api/type/Xamarin.UITest.IApp`.

■ **Note** Though we will be unit testing the UI, it's still worth running the compiler over the code to ensure everything is working and the code actually compiles!

To ensure data is getting to and from the view model, the code is compiled and run through the simulator. I know the data entered will fail, as the passwords need to be six or more characters long. The simulator showed things worked. See Figure 7-17.

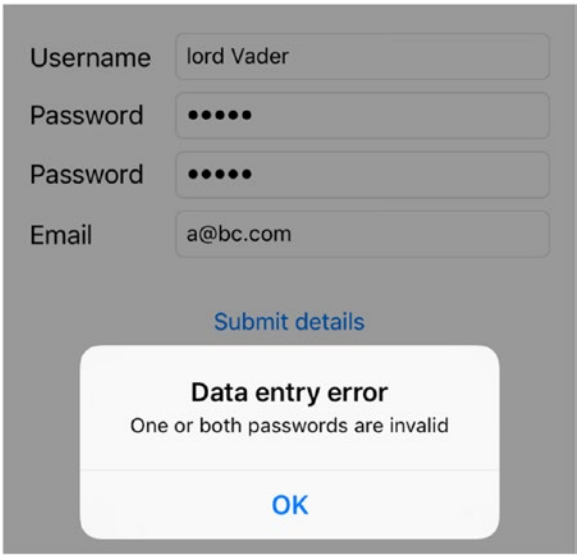


Figure 7-17. *The simulator shows things are working*

Knowing the code works, we can write the UI unit test. The beauty of writing the unit test is that every possible permutation can be tested without the tedium of typing into each editable text box and hitting the button every time.

Getting the Automated Test to Work

The UI unit test will be using the `Marked` method to find the correct UI element to enter text into. The key to ensuring this works on both platforms is that the names are consistent between the platforms (for example, on Android the ID may be `editEmailAddress`, but on iOS, the element name is `txtEmailAddress`; if `Marked` is set to look for `editEmailAddress`, the UI unit test will fail on iOS unless either the `AccessibilityIdentifier` or `AccessibilityLabel` are set to also be `editEmailAddress`. Similarly, if `Marked = "txtPasswordTwo"` is used, Android will not pick it up, as `txtPasswordTwo` needs to be identified in either the `Text`, `Id`, or `ContentDescription`). As long as `Marked` looks for something identified the same way on both platforms, it will work.

As we can run multiple tests using the same test method, the following can be used:

```
[TestCase("", "password", "password", "a@bc.com")] // empty username
[TestCase("username", "pass", "pass", "a@bc.com")] // invalid passwords
[TestCase("username", "password1", "password", "a@bc.com")] // password mismatch
[TestCase("username", "password", "password", "abc.com")] // invalid email address
[TestCase("username", "password", "password", "a@bc.com")] // everything is correct
public void InvalidEntryErrors(string username, string passone, string passtwo, string email)
{
    app.EnterText(c => c.Marked("txtUsername"), username);
    app.EnterText(c => c.Marked("txtPasswordOne"), passone);
    app.EnterText(c => c.Marked("txtPasswordTwo"), passtwo);
```

```
    app.EnterText(c => c.Marked("txtEmailAddress"), email);

    app.Tap(c => c.Marked("btnSubmit"));
}
```

When executed, iOS will auto-start the simulator, but on Android, you will need to start an emulator before executing.

The unit test shows something interesting (Figure 7-18).

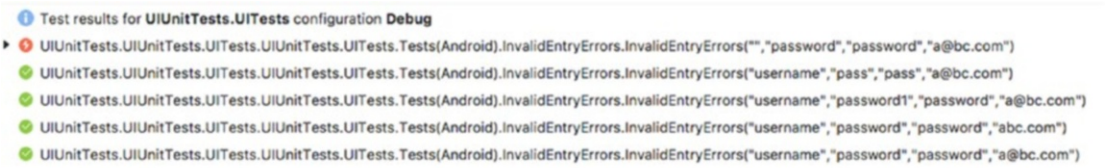


Figure 7-18. *Unit test results*

If `username` is empty, then the unit tests fail. Is this the result of our view model `RelayCommand` or the unit tests themselves? The simplest way to find out is to enter the details that failed and see what the debugger comes back with.

The exception comes in the view model when testing `username`. The `RelayCommand` passes the `Username` property to the validator using the following:

```
if (validationService.IsUsernameValid(UserName))
```

The validator test does not trap correctly, however:

```
public bool IsUsernameValid(string username)
{
    return username.Length >= 6;
}
```

This can be fixed by adding a `?` after `username`. This is a C# 6 change that checks an object for `null` and exits if it is. This only works if there is an extension method in use after the object (meaning you can't have `username?` to mean `string.IsNullOrEmpty(username)` or test for a returned list to be null without `[say].Count` after the list).

The other `ArePasswordsSame` and `IsEmailValid` validator methods will need a null test applying. In these cases, wrapping the code in an `if (!string.IsNullOrEmpty(...))` before testing values will sort the issue.

Once these changes have been made, the tests can be re-run (Figure 7-19).

Figure 7-19. *Re-run the tests*

All the unit tests have now been performed (for Android) successfully. For iOS, we still have a test fail, but in this case it's a false positive (the test failed, but it actually was because of the UI unit test system). See Figure 7-20.

Figure 7-20. *iOS results*

When an iOS application fires up, there is a 17-second "fight or flight" interval. If the app doesn't hit the main `ViewController` within that time (irrespective of whether it is running a `Xamarin.Forms` application, storyboard, or XIB-based application), iOS will consider the app to have stalled and quit.

With the first unit test, the UI has not hit, so the test cannot occur. Subsequent tests work fine, as the application has already fired up and the UI has become available. To solve this issue, we have it insert a wait into the unit test. The following will work for testing if a `UINavigationBar` has appeared:

```
app.WaitForElement(c => c.Class("UINavigationBar")
```

The problem is that `UINavigationBar` is for iOS only. If this is included, what happens with the Android UI test? Will it work?

The tests will run, but they will all fail, as the unit test will time out waiting for the `UINavigationBar`.

Conditional Compilation Within a UI Unit Test

Code can have conditional compilation rules applied. The names `__IOS__` and `__ANDROID__` are defined within the compiler and can be used to turn on and off certain aspects of the tests; therefore:

```
#if __IOS__
            app.WaitForElement(c => c.Class("UINavigationBar"));
#endif
```

will only be executed if the test is running on iOS.

With this in place and the unit tests re-run, everything will pass.

■ **Note** It is always worth checking the output of a unit-test failure. For example, it is possible on iOS for a test to fail because the `DeviceService` app (this is the unit test shell—it essentially acts as the sandbox that the unit test uses to insert data into the app under test) failed to shut down properly. This is another example of a "false-positive" and does not mean the unit test has failed per se, but more that the unit tester software failed.

Unit Testing a Xamarin Forms App

Xamarin Forms apps can also be unit tested, but require a few tweaks to ensure the tests run.

As with a normal UI project, ensure the Xamarin test-cloud checkbox is selected when creating the app. This will do some of the legwork for you.

With Xamarin Forms (version 2.3.4 or above), the platforms automatically have the UI elements map the `ContentDescription` or `AccessibilityIdentifier` to the `AutomationId` property (required for the `Marked` method).

The unit test itself really is no different from the platform test, except that if the `Class` method is used, the abstracted name has to be used (for example, `Class("UIButton")` or `Class("EditText")` would become `Class("Button")` and `Class("Entry")` respectively.

Conclusion

I hope this chapter has given you a useful introduction to the power of unit testing and the advantages using it can give on both the code and UI levels of testing. They're not that difficult to write and, as with anything, the more you do them, the quicker they become. Testing should be something you bolt on as a developer; it should be something you do as second nature.

CHAPTER 8

Using Xamarin Forms

An aspect of mobile development that is being used more over time is leveraging the UI abstraction layer known as Xamarin Forms. It is an oft-quoted statistic that using Xamarin allows roughly 90 percent of the same business logic between apps (which can further be made simple to use with one of the MVVM frameworks). This still means that a user interface has to be created, which means that, as a developer, you still need to be good with design on your platform. Not many developers can lay claim to this. Xamarin Forms provides an abstraction layer to the UI. In simple terms, through a common UI object type, the developer can quickly create a user interface. The `Forms` library deals with translating the UI object type into the platform type (therefore, something like an `Entry` is translated into a `UITextField` on iOS, `EditText` on Android, and `TextBox` on Windows Phone; as a developer, you only need to handle the Entry object).

In this chapter, we will be using MVVM Light with a Forms application, as well as technologies from previous chapters (such as SQLite, web services, and connectivity). We will also be adding in globalization of the text.

Before we go any further, I must thank Harrington Electronic Systems (and especially Will Harrington) for allowing me access to the uSee API. The example source contains only a subset of the full API due to commercial restrictions.

Setting Up Your project

There are two models that developers typically adopt when creating a Forms app using an MVVM framework, as follows:

1. Include the framework within the Forms PCL.
2. Keep the framework as its own PCL.

There is no right or wrong way to create a Forms MVVM app, but I find that keeping the constituent parts separate makes it easier to debug; either way will work, so that part is up to you.

As with any Forms application, first create your Forms app.

Creating Your Forms App with Visual Studio

Creating your Forms app is a simple multi-step process. Assuming that you don't have a project open, click on New Solution (Figure 8-1).

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_8

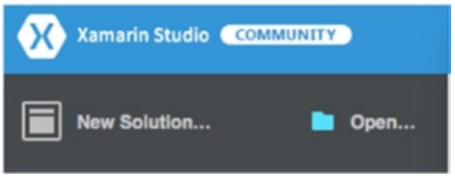


Figure 8-1. *Creating a new solution*

You will be presented with the modal window shown in Figure 8-2.

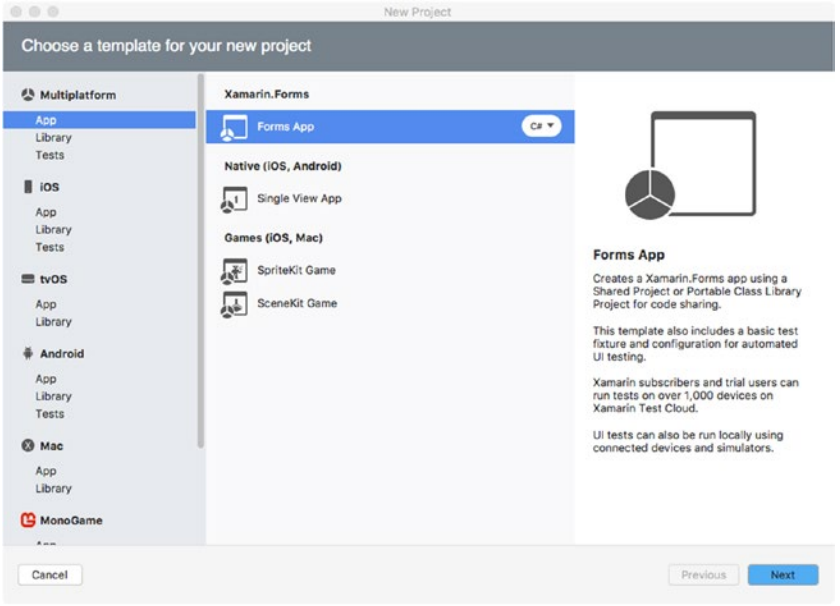


Figure 8-2. *Chosing the template*

Ensure that Multiplatform App ➤ Forms App is selected. You can select whether you are using F# or C# via the drop-down menu. When you're ready, click *Next*. The window will close and be replaced with the window shown in Figure 8-3.

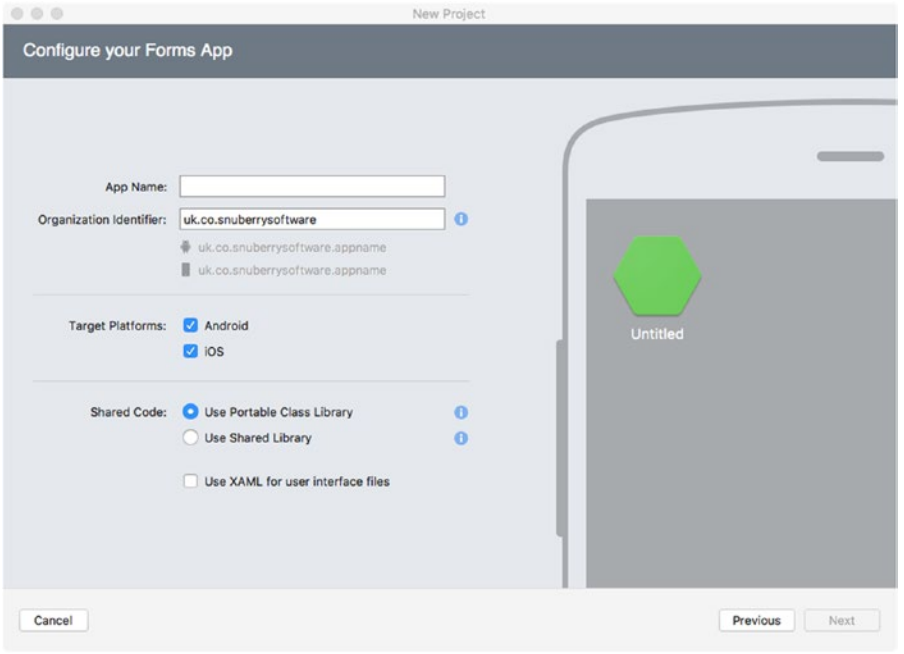


Figure 8-3. *Configure the app*

Type in a name for your app. If you wish to create the UI using XAML, click the checkbox next to "Use XAML for user interface files." For my example code, I'll be using both XAML and code. If you wish to mix and match, make sure the "Use XAML" box is unchecked.

After giving the project a name, click on *Next*. The window will be replaced again (Figure 8-4).

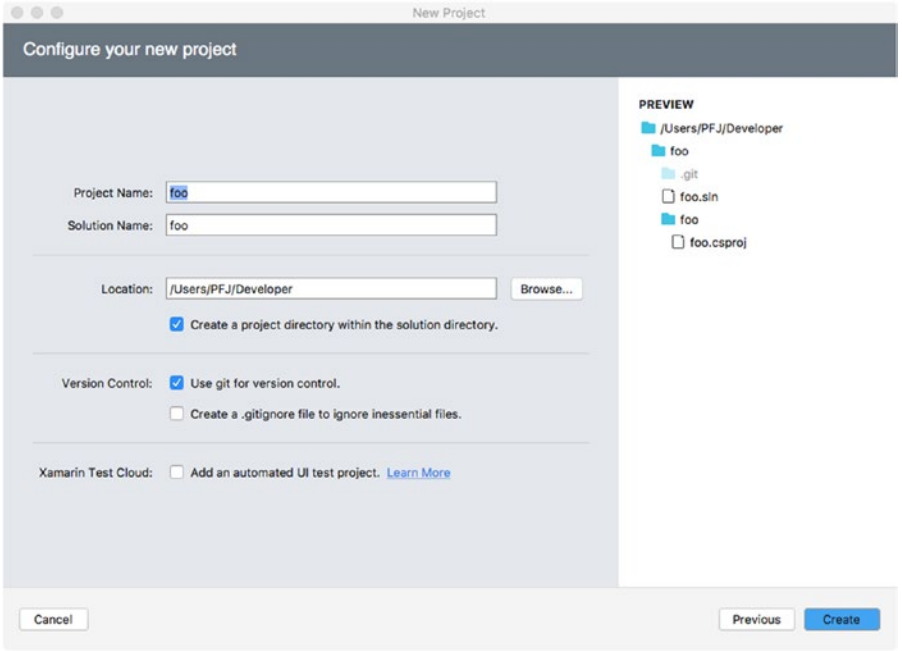


Figure 8-4. *Naming your app*

Finally, give the project a name and location to save to. If you don't wish to use source control, uncheck the "Use git for version control" box. When you're done, click on *Create*, and VisualStudio for Mac will create the project for you. When the project is ready, something similar to the image in Figure 8-5 will be in the Project Explorer pane. The test folder contains the Forms PCL with Droid and iOS and the initializers for Android and iOS. As of the time of writing, there are no plans to be able to create Windows Phone projects on a Mac.

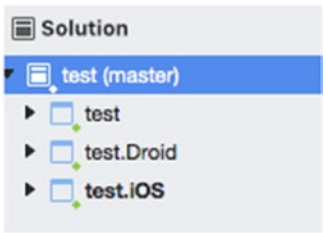


Figure 8-5. *The solution explorer*

Creating a Forms App on Visual Studio

Click on File ➤ New Project. You will be presented with a window such as that seen in Figure 8-6.

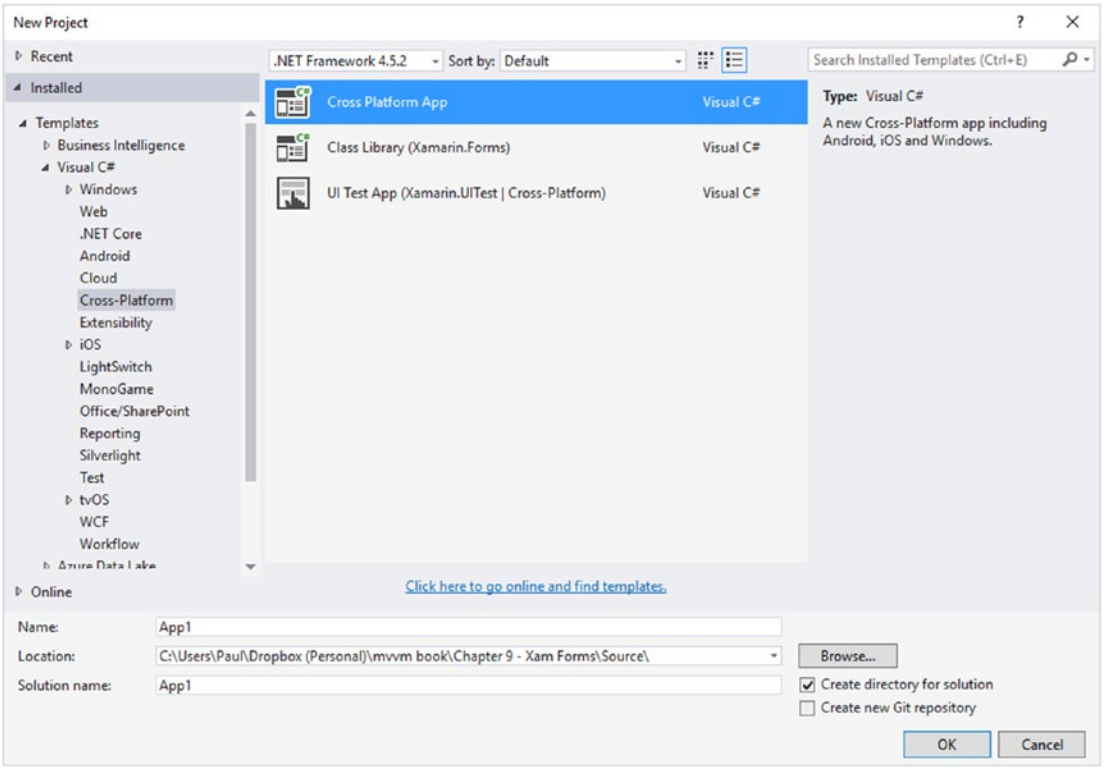


Figure 8-6. *Creating the app on VisualStudio*

You must select the Cross Platform App, not the Class Library, to create your Forms app. Name the solution and, when ready, click *OK*. The New Project window will close and be replaced with a new window, as shown in Figure 8-7.

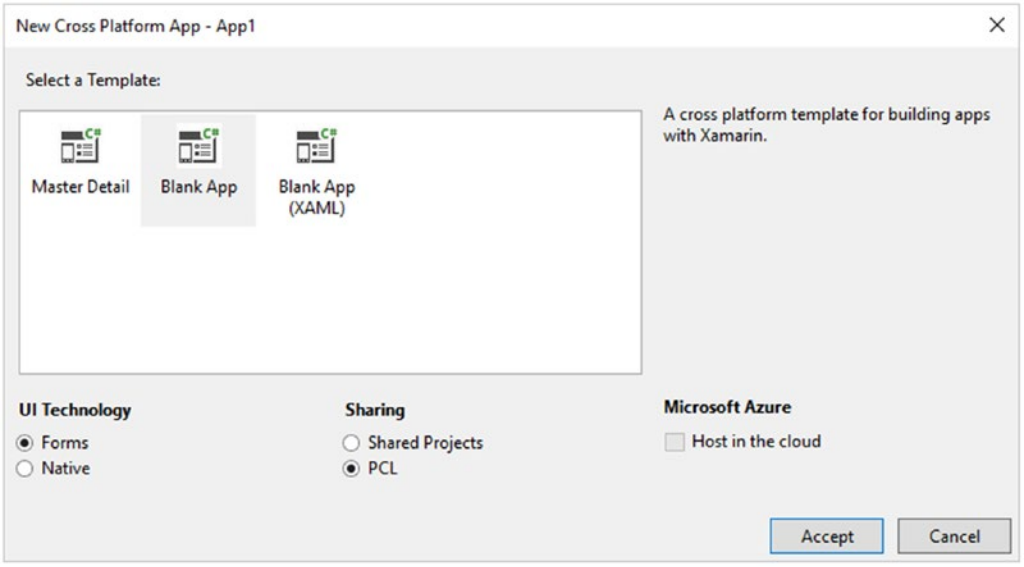


Figure 8-7. *Chose the project template*

Select Blank App and that it is a PCL rather than a shared project. Once set, click *Accept*. The window will close, and the app will be set up.

Removing the UWP Project

At the time of writing, UWP is not fully supported with Xamarin Forms. We therefore should remove the UWP project. The Solution Viewer will show something like what is shown in Figure 8-8.

Figure 8-8. *Removing the UWP project*

Right-click on the UWP project. On the menu select Remove (Figure 8-9).

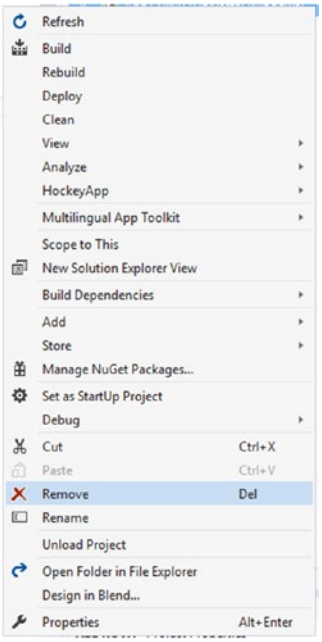


Figure 8-9. *The delete menu option*

When you select Remove, you will be asked to confirm you want the removal (Figure 8-10).

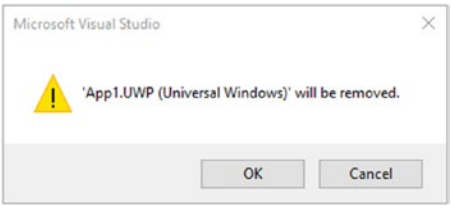


Figure 8-10. *Removal confirmation*

Select *OK*. The UWP project will be removed, and the Solution Explorer will reflect this (Figure 8-11).

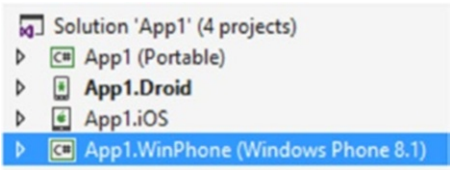


Figure 8-11. *UWP project removed*

Adding the MVVM Project to the Forms App on VisualStudio for Mac

In the Solution Explorer, highlight the top level of the Forms app (it has the word *master* next to the project name). See Figure 8-12.

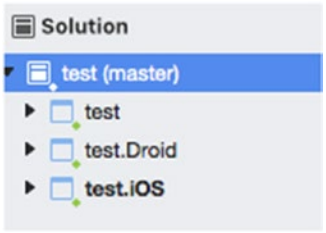


Figure 8-12. *Solution explorer on VS for Mac*

Click and select Add ➤ New Project from the context menu (Figure 8-13).

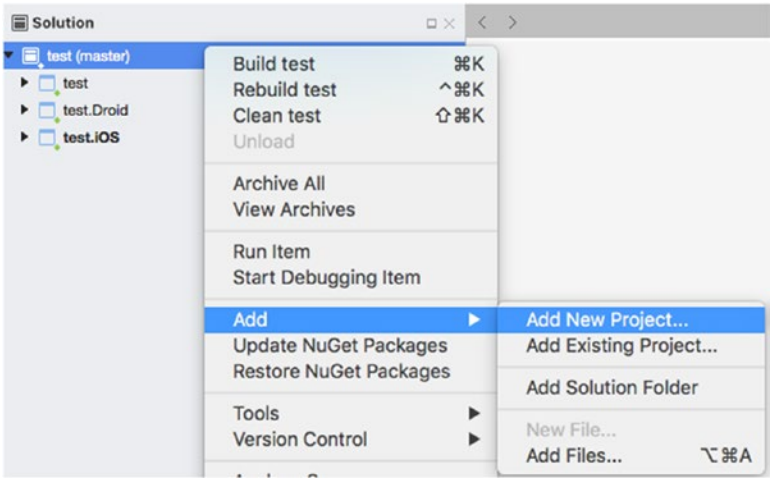


Figure 8-13. *Add new project*

You will then be presented with a new project window (Figure 8-14).

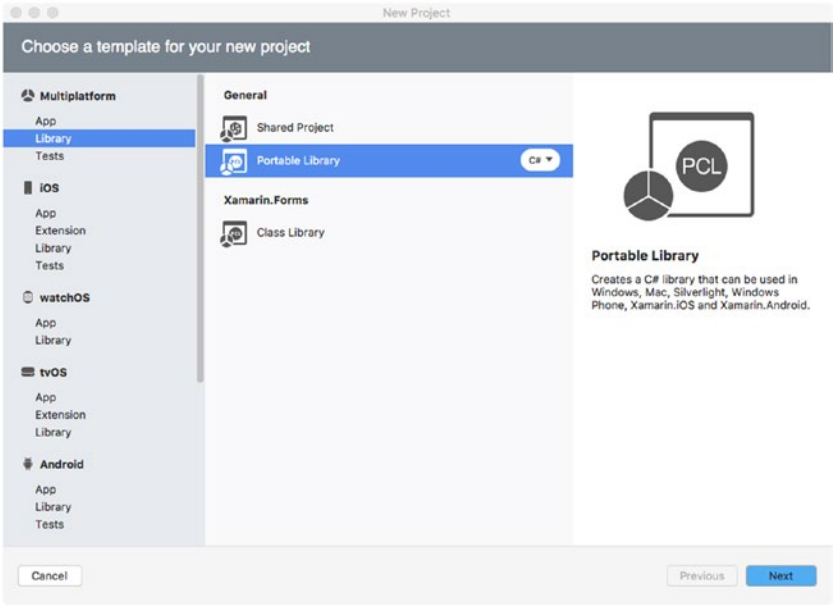


Figure 8-14. *New project choice on VS for Mac*

Select Library ➤ Portable Library. Click on *Next*. The window will then be replaced with that seen in Figure 8-15.

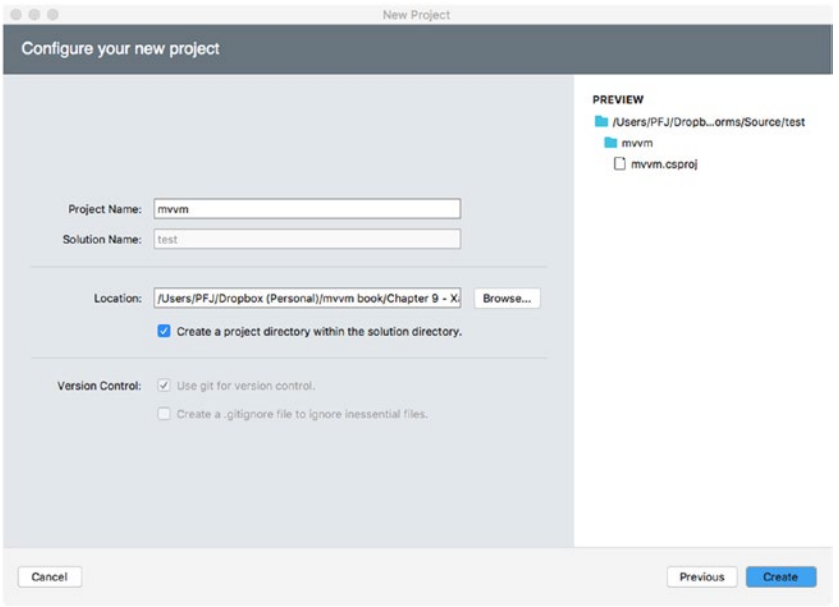


Figure 8-15. *Configuring the project*

As before, give the project a name. You shouldn't change the location path. Click on *Create* when you're happy with the names you've entered. Note that your selection for version control will follow through to the new PCL project.

Once created, you will see the new project in the Solution Explorer, as in Figure 8-16.

Figure 8-16. *Project successfully added*

Adding the MVVM Project to Visual Studio

In the Solution Explorer, right-click the root of the project (Figure 8-17).

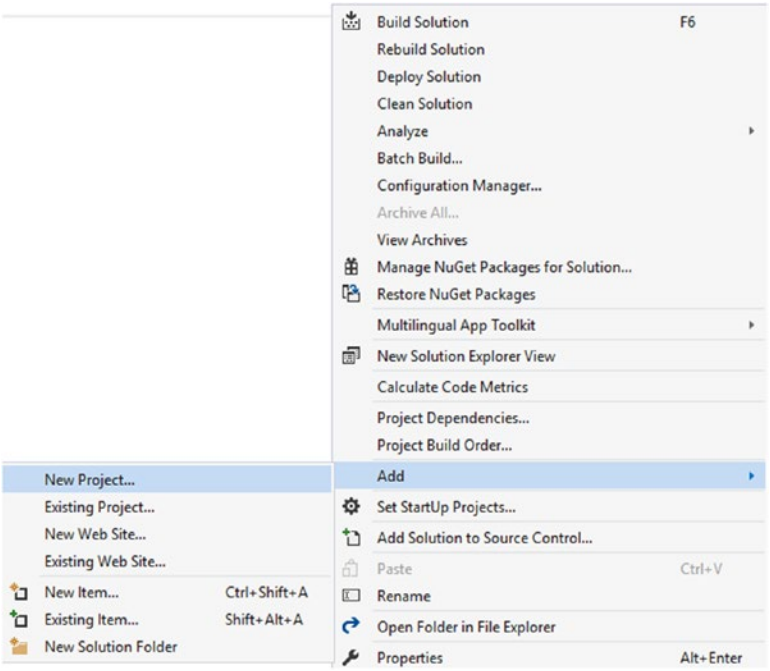


Figure 8-17. *Add new project on VS*

Add a new project from the context menu. You will be presented with a project-type window (Figure 8-18).

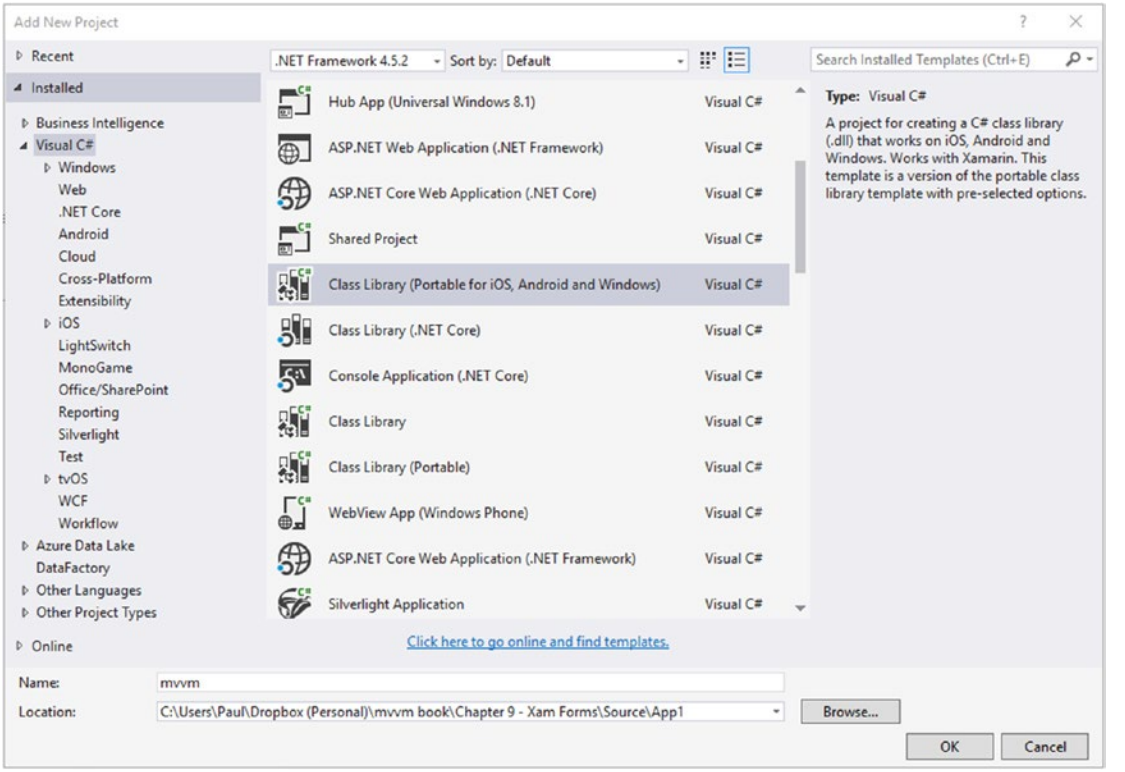


Figure 8-18. *Add new Class Library*

It is simpler to select Visual Studio from the project types and then select Class Library from the project templates. Give the project a name and click on *OK*. You shouldn't change the location. The window will disappear, and Visual Studio will create the new project, which will show up in the Solution Explorer (Figure 8-19).

Figure 8-19. *Project successfully added*

Adding the MVVM Framework

As we have done throughout this book, we need to add the framework to the newly created MVVM project. You will also need to add the library to all of the platform projects as well as to the Forms project.

`ViewModelLocator.cs` should remain within the MVVM project. All of the directories that we have used before in previous chapters (such as Models, ViewModel, Helpers, and so forth) should also be created if required.

Where Setting Up for Forms Differs

MVVM Light will require a bit of additional work for it to function. Out of the box, there is no support for the navigation service or the dialog service, which will need to be implemented within the Forms PCL. The source code for both can be found in the Source Code folder for this chapter.

We do not need to set up the navigation on each platform, as Forms itself deals with the navigation. This can be set up fairly easily from within the Forms singleton file:

```
public static ViewModelLocator locator;
public static ViewModelLocator Locator { get { return locator ?? (locator = new
ViewModelLocator()); } }
```

The navigation service is then created:

```
var nav = new NavigationService();
nav.Configure(ViewModelLocator.MainKey, typeof(Login));
nav.Configure(ViewModelLocator.LatestWeatherKey, typeof(WeatherView));
nav.Configure(ViewModelLocator.MainPanelKey, typeof(MainPanelView));
nav.Configure(ViewModelLocator.SiteDataKey, typeof(SiteDataKey));
nav.Configure(ViewModelLocator.SiteImagesKey, typeof(SiteImagesView));
nav.Configure(ViewModelLocator.SiteKey, typeof(SitesView));
SimpleIoc.Default.Register<INavigationService>(() => nav);
```

We next deal with registering for other services. Note: We are not yet initializing the navigation.

To register the SQL service, we will need to use dependency injection down to the platform, as we need to create the database file:

```
DependencyService.Get<MvvmFramework.ISqLiteConnectionFactory>().GetConnection();
```

Note that we set the interface not within the Forms project but, in this case, within the MVVM project. If you set it within the Forms project (and the MVVM code is not held within the Forms project), then the registration will fail.

The interface looks like this:

```
public interface ISqLiteConnectionFactory
{
    SQLiteConnection GetConnection();
}
```

On the platform, we create the connection (and the database) and register the SQL service from there. The following example is from the Android project (it is very similar for the iOS and Windows Phone projects):

```
public class SQLConnection : MvvmFramework.ISqLiteConnectionFactory
{
    readonly string Filename = "usee.db";
```

```
    public SQLiteConnection GetConnection()
    {
       var path = System.Environment.GetFolderPath(Environment.SpecialFolder.Personal);
       path = Path.Combine(path, Filename);

       SimpleIoc.Default.Register<ISqLiteConnectionFactory, SQLConnection>();

       return new SQLiteConnection(SQLitePlatform, path);
    }

    public ISQLitePlatform SQLitePlatform
    {
        get { return new SQLitePlatformAndroid(); }
    }
}
```

This creation of the SQLite registration on the platform is the same for Forms as it is for a native project.

■ **Caution** If you do not set up the SQL handler correctly, the code will fail anywhere that the `SQLConnection` is called. This may lead to a lot of confusion when it comes to trying to debug, as you will get the infamous "The service has not been registered" error. However, that may not be obvious from the throwback generated during the crash.

Once we have registered using SQLite, we can initialize the navigation service. This is again different than a native approach, as the implementation of the navigation service requires the `NavigationPage` be passed to it. We can also initialize the dialog service at the same time.

```
var firstPage = new NavigationPage(new Login());
nav.Initialize(firstPage);
var dialogService = new DialogService();
SimpleIoc.Default.Register<IDialogService>(() => dialogService);
dialogService.Initialize(firstPage);
MainPage = firstPage;
```

Passing Parameters Using the Navigation Service

We have seen in past examples that we can pass parameters within the navigation service when we call `NavigateTo` with a parameter after the page key. This is also enabled with the code provided in this chapter. However, rather than requiring the platform to use `GetAndRemoveParameter<object>` (for Android), the parameters are passed into the new `ContentPage` using the constructor.

For example, in our app, we pass an ID from the navigation service:
(from `MvvmFramework/ViewModels/MainPanelViewModel.cs`)

```
RelayCommand btnWeather;
public RelayCommand BtnWeather
{
    get
    {
```

```
        return btnWeather ??
        (
            btnWeather = new RelayCommand(() => navService.NavigateTo(ViewModelLocator.
            LatestWeatherKey, SelectedSite.id))
        );
    }
}
```

Our second parameter is a string. In our Forms content page, we will need to intercept this parameter within the page constructor.

■ **Note** If you omit the parameter, the app will crash, as there is not a constructor available that takes the additional parameter.

In `Forms/Views/WeatherView.cs`, we therefore need the following:

```
public class WeatherView : ContentPage
{
    string SiteId;

    public WeatherView(string siteId)
    {
        SiteId = siteId;
    }
}
```

Data Binding

One of the strengths of any MVVM framework is the ability to bind the input or output of a UI object in a view to the processed data in the view model. Xamarin Forms utilizes binding to most UI objects. This is normally done via the `BindingContext`, though it is not always the case (or at least is obvious).

Take, for example, a simple list. The `ItemsSource` takes an observable collection that contains many data objects held in a class (think of it as `List<T>` where `T = class foo`). When passed into the `ViewCell` (the list's `ItemTemplate`), the instance of the list becomes the binding context. We then bind to the UI object's property (for example, `Label.TextProperty`) via the following:

```
var myLabel = new Label
{
    TextColor = Color.Blue,
    FontSize = 16
};
myLabel.SetBinding(Label.TextProperty, new Binding("property_name"));
```

When the cell is produced, the output looks as if we have placed the contents of `property_name` directly into the `Text` property of the label. This is because we have, except it is through a binding. The beauty of binding is if the property issues the `PropertyChanged` event, the UI object bound to the property is notified and altered.

Is this of much use?

Of course it is! Think of a simple example. We have an app that is displaying soccer results for the Premier League, and it's the last day of the season (so all teams are playing). We will have 22 teams on the UI. Teams score, and the UI changes to reflect this.

We could just regenerate the entire user interface, but this is very messy, and depending on the level of complexity it is possibly going to be memory or resource hungry. While we're refreshing, there is nothing to say that someone else won't score, so the refresh must finish the current loop and start again.

If we had a class constructed like this, however:

```
public class Teams
{
    public string HomeTeam {get;set;}
    public string AwayTeam {get;set;}
    public int HomeScore {get;set;}
    public int AwayScore {get;set;}
}
```

and placed that class into our collection list, which was then passed into the `List` itself, we could create the `ViewCell` to hold these details and bind the `Label.TextProperty` to the property from the class. We would then have an updating view. Well, nearly. We would need to implement `OnPropertyChanged` within the `Teams` class and handle the `BindingContextChanged` event within the `ViewCell`.

OnPropertyChanged

We implement this on the class we store information in. To start, we need the class to inherit `INotifyPropertyChanged`:

```
public class Teams : INotifyPropertyChanged
{
    public string HomeTeam {get;set;}
    public string AwayTeam {get;set;}
    public int HomeScore {get;set;}
    public int AwayScore {get;set;
}
```

We then need to implement the code to satisfy the interface:

```
public class Teams
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged == null)
          return;

        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
    public string HomeTeam {get;set;}
    public string AwayTeam {get;set;}
    public int HomeScore {get;set;}
    public int AwayScore {get;set;}
}
```

Finally, we modify the `HomeScore` and `AwayScore` properties to issue the change of score:

```
public class Teams
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged == null)
          return;

        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
    public string HomeTeam {get;set;}
    public string AwayTeam {get;set;}

    int homeScore;
    public int HomeScore
    {
        get {return homeScore;}
       set { homeScore = value;
             OnPropertyChanged("HomeScore”);
           }
    }
    int awayScore;
    public int AwayScore
    {
        get {return awayScore;}
       set { awayScore = value;
             OnPropertyChanged("AwayScore”);
           }
    }
}
```

When the home score or away score changes, the event is broadcast.

BindingContextChanged

In the `ViewCell`, we create five labels: `lblHomeTeam`, `lblAwayTeam`, `lblHomeScore`, `lblAwayScore`, and one to hold the letter “v.”

We bind each label’s `TextProperty` to the corresponding property from the class. This sets the UI up for the initial scores and team names. We next need to intercept the `BindingContextChanged` event. This is placed directly on the label:

```
lblHomeScore.BindingContextChanged +=
{
    var m = (Teams)BindingContext; // cast the binding context to the class
    lblHomeScore.Text  = m.HomeScore.ToString();
};
```

We don't always need to intercept `BindingContextChanged`. For example, if we have a property called `IsBusy` and we bind that to an `ActivitySpinner` UI element, we can start or stop the spinner when we alter `IsBusy` so long as the event is broadcast (in this case, the `IsVisible` property of the spinner should also be bound to the same property; this makes the spinner appear or disappear accordingly).

Accessing the View Model from Within the ContentPage

To bind our view model to the view within a content page, we need to set the `BindingContext` of the content page to that of the view model. The only problem is that we can't do this directly (the types are incompatible and can't be cast).

We can, however, perform the same "trick" that we earlier used in a non-Forms app to access the view model:

```
MainPanelViewModel ViewModel => App.Locator.MainPanel;
```

`App.Locator` is set in the Forms singleton class:

```
public static ViewModelLocator locator;
public static ViewModelLocator Locator { get { return locator ?? (locator = new
ViewModelLocator()); } }
```

Once we have this alias set, we can directly allocate it to the `BindingContext` for the page.

■ **Note** It is always good practice to bind the context within a `try/catch` construct during development. Any mistakes in either the bindings on the page or the view model will cause the page to not load. Using `try/catch` will at least tell you where the problem is!

```
try
{
     BindingContext = ViewModel;
}
catch (Exception ex)
{
     Debug.WriteLine("Exception - {0}--{1}", ex.Message, ex.InnerException);
}
```

■ **Note** If you are using XAML for your UI, perform the binding in the code behind `.cs` file in the same way as we've done here. You can bind the view model to the `BindingContext` in XAML if the view model is native to Forms, but as the view model is *not* from Forms, you will run into the same issue of there being an incompatible type.

Binding Within XAML

Binding within XAML looks different than binding to an object in C#. Say we have an Android `TextView` and wish to bind to it. We have code that looks like this to do it:

```
this.SetBinding(
                () => ViewModel.Text,
                () => TxtView.Text,
                BindingMode.Default);
```

We set the `TxtView.Text` property to the contents of the `ViewModel.Text` property and use the default `BindingMode` (how the UI object reacts to changes in the `ViewModel` property).

In XAML, that becomes the following:

```
<Label Text="{Binding Text, BindingMode=Default}" />
```

■ **Note** As long as the `BindingContext` has been set up as previously shown, this binding will work. If the `BindingContext` has not been set, the label will still be created, but nothing will show inside of it.

When binding, all of the rules for standard forms binding apply. For example:

```
Text="{Binding MyIntValueAsString, StringFormat=IntValue is \{0\}}"/>
```

will produce a label with the text "IntValue is 10" inside of it. If you need to create a converter, again, this is supported:

```
label.SetBinding(Label.TextColorProperty, new Binding("is_reply", converter: new
BoolToColor()));
```

or

```
<Label TextColor="{Binding is_reply, Converter = BoolToColor()} />"
```

(if `BoolToColor()` is not in the same namespace as your user interface, you will need to append this before your method name (for example, `Converters.BoolToColor()`).

Code Localization

Localizing the text used within any app is becoming a bigger and bigger business. While it is entirely possible to do this within your app on a platform-by-platform basis, Forms uses a system very similar to that used within Windows, namely a Resource XML file with a `Resource.designer.cs` file behind it.

This requires a very simple two-step process.

Step 1: Setting up the localization text

To do this:

1. Create a directory in the root of the Forms application and call it something like Languages.
2. Right-click on the directory and choose Add New File.
3. Of the new file choices, choose New Resource File. Call it Langs. When created, it will show as `Langs.resx` in the Solution Explorer.
4. Select the properties of the file and set the Custom Tool to `PublicResXFileCodeGenerator` and the Resource ID to `MyNamespace.Languages.Langs.resources`.

And that's it for Step 1! You can then add your text into the `Langs.resx` file. Do not alter the `resources.cs` file; it is regenerated each time you add something to the `.resx` file.

The format of resx content is an XML representation similar to that of a dictionary:

```
<data name="General_Close" xml:space="preserve">
    <value>Close</value>
</data>
```

To access this within your code:

```
using Languages;
...
myLabel.Text = Langs.General_Close;
```

It is slightly different within XAML. First, we need to include a converter for the title and a helper within the `ContentPage` header. Then, we need to create a key in the `ContentPage.Resources` file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.
microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:MyApp;assembly=MyApp"
xmlns:helpers="assembly=MyApp" x:Class="MyyApp.Views.LoginView" Title="{Binding
Converter={StaticResource Language}, ConverterParameter=Login_Title}"
BackgroundColor="#022330">
<ContentPage.Resources>
    <ResourceDictionary>
         <helpers:LanguageConverter x:Key="Language" />
    </ResourceDictionary>
</ContentPage.Resources>
```

Note that we have a converter here. This converts the token to the string and returns the string back to the caller:

```
<Label Text="{Binding Converter={StaticResource Language}, ConverterParameter=Login_
Password}" />
```

The converter looks like this:

```
public class LanguageConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        var prop = (string)parameter;
         if (string.IsNullOrEmpty(prop))
             return string.Empty;
         return Langs.ResourceManager.GetString(prop);
    }

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

Step 2: Getting the platforms to cooperate

For the platforms to output the correct language, we have to first set the `Langs.Locale` to that of the platform. There is no point in outputting text in English when your phone is set to Slovakian!

This is performed using dependency injection within the Forms singleton. First, though, we need to create an interface for the injection to work on:

```
public interface ILocalize
{
    string GetCurrentLocal();
    void SetLocale();
}
```

We need to be able to get the current locale code as well as set the locale (for example, we decided to change our language within the app to go from English to German).

The code to set the locale is as simple as this:

```
var netLanguage = DependencyService.Get<ILocalize>().GetCurrentLocal();
Langs.Culture = new CultureInfo(netLanguage);
DependencyService.Get<ILocalize>().SetLocale();
```

It's then up to the platform to get and set the languages (you can find the code for this in the example).

Can You See a Problem with This?

There is a very subtle problem with this, and it only occurs when the MVVM project is kept apart from the Forms PCL. We have something known as a use-case problem. Say we call the webservice (this is in the MVVM project) to get some information, and the service fails. We need to be able to produce an error. The problem is that you can't include the Forms MVVM binary from the Forms PCL back into the MVVM project (This gives rise to a circular dependency. The Forms PCL relies on the MVVM PCL; in order to build, the

MVVM PCL relies on the Forms PCL. As soon as one of the two is built, the other is out of date, so it needs building, which then puts the other library out of date, which will then need recompiling which . . . well, you get the idea).

We can't use a Forms `DialogAlert` in the MVVM project, but we do have the `DialogService`, which does not have access to the localized strings.

Solving This Conundrum

Let's think about this and see if there is anything inherently usable from how MVVM works.

The view gets and sets data to the associated view model. The view model calls the webservice, which returns to the view model, which returns control to the view. We know this.

The errors aren't typically going to be that exhaustive (you typically have a specific title and message for authentication failures or data corruption for the webservice), so we can therefore pass them into the view model, and that can handle the new messages.

This approach is fine if you have only a single error message and title going to the view model, but what happens if you have several messages and titles going up? We could use a `List<string>`, but then how could we determine which message to use for a given scenario? We need to be able to associate a message handle with a message.

We can therefore use a `List<Dictionary<string, string>>` and a simple helper function to return the correct data. Something similar to this can be used as a receiver for the messages from the view:

```
List<Dictionary<string,string>> errorMessages;
public List<Dictionary<string,string>> ErrorMessages
{
    get {return errorMessages;}
    set {Set(()=>ErrorMessages, ref errorMessages);}
}
```

For the helper function, we use a small piece of code (it will require using `System.Linq`; add it to the top of the source file):

```
public string GetErrorMessage(string tag)
{
    if (ErrorMessages == null)
        return string.Empty;
    var errorMessage = string.Empty;
    var errorDict = ErrorMessages.FirstOrDefault(w=>w.Keys.Contains(tag));
    errorDict.TryGetValue(errorDict, out errorMessage);
    return errorMessage;
}
```

We now have another small problem: this code will need to be replicated everywhere that we need to produce a `dialogService.ShowError` (or `ShowMessage`), which means a lot of replication.

Let's look again at our view model:

```
public class MainPanelViewModel : BaseViewModel
```

We're not inheriting the `ViewModelBase` from MVVM Light, but rather our own base class (which does inherit `ViewModelBase`).

As all our view models use this `BaseViewModel`, we can place our localization handler code inside of the `BaseViewModel`. We then set it as we would any other property in the view model.

The question now is how does this all work? Surely, the MVVM project is called into existence before we do start anything?

The answer to this is in how Xamarin Forms starts. Let's consider the Android code:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    global::Xamarin.Forms.Forms.Init(this, bundle);
    LoadApplication(new App());
}
```

You cannot start a library by itself; it needs an entry point. The second line in the preceding code initializes the Forms PCL with a pointer to the activity and the bundle. We then create an instance of `App()`, which in turn creates whatever is needed for the Forms PCL to work.

While we have a couple of properties with which to access the MVVM PCL `ViewModelLocator`, we don't really start firing things through until we create the services, so really the Forms PCL has already created the localization and fired its own routines up before we even look at the MVVM PCL.

File Handling

Let's consider the following scenario. My webservice grabs a file from the server. The file can be anything (let's suppose it's a picture of my dog, Ollie). The webservice sits in the MVVM PCL project, so it has no idea where to save the file or, indeed, how to save.

There are several NuGet packages that provide file handling, the most popular being `PCLStorage`. This provides us with something akin to the standard IO.File classes.

Our complete save routine would be this:

```
public class FileIO
{
    IFileSystem Current { get; }
    IFile File { get; }

    byte[] ReadFully(Stream input)
    {
        var buffer = new byte[16 * 1024];
        using (var ms = new MemoryStream())
        {
            int read;
            while ((read = input.Read(buffer, 0, buffer.Length)) > 0)
            {
                ms.Write(buffer, 0, read);
            }
            return ms.ToArray();
        }
    }

    public async Task SaveFile(string filename, Stream data)
    {
        await Current.LocalStorage.CreateFileAsync(filename, CreationCollisionOption.
        ReplaceExisting).ContinueWith(async (t) =>
        {
```

```
            if (t.IsCompleted)
            {
                var bytes = ReadFully(data);
                using (var stream = await File.OpenAsync(FileAccess.ReadAndWrite))
                {
                    stream.Write(bytes, 0, bytes.Length);
                }
            }
        });
    }
}
```

We can also load back to a stream and pass that back to the view. In the view model, we have:

```
public Stream GetFile(string filename)
{
     return new FileIO().LoadFile(filename).Result;
}
```

And we then add the following to our `FileIO` class:

```
public async Task<Stream> LoadFile(string filename)
{
     var folder = FileSystem.Current.LocalStorage;
     var file = await folder.GetFileAsync(filename);
     //load stream to buffer
     using (var stream = await file.OpenAsync(FileAccess.ReadAndWrite))
     {
         var length = stream.Length;
         var streamBuffer = new byte[length];
         stream.Read(streamBuffer, 0, (int)length);
         return new MemoryStream(streamBuffer);
     }
}
```

We then can convert this back to a Forms `Image`:

```
var img = new Image
{
    Source = ImageSource.FromStream(ViewModel.GetFile("filename.jpg"))
};
```

Settings

There is some argument over whether the platform settings should be abstracted to the MVVM PCL; after all, with a common library we should be able to do this. While this is entirely true, we must remember what the framework is for. MVVM is there essentially to provide a separation between UI, the platform, and what is described as the business logic. We need to ask ourselves, are the settings UI, platform, or business logic? The simple answer is that they are for the platform but are set by the UI.

Creating a Responsive Application

In our example application, there are two parts that cause a slowdown: all web services (nothing can be done about these, as we rely on external factors) and the generation of the image grid.

If you run the Forms MVVM app, you will see that the grid is extremely slow during an update; even with the images being downloaded asynchronously, it's slow. How can we improve the speed?

We have three ways to address this issue. We can first just use the standard Forms `Image`, which will download the images asynchronously using the `ImageSource.FromURL` static method. We could use the `FFImageLoading` package from NuGet, or we could just write our own asynchronous file loader.

While all three download the files asynchronously, only the Forms `Image` will not allow us to use a placeholder image, so all you will see on screen is the date under the cell.

The three options are given in the example source code. Our test app is using an async lazy loader; feel free to experiment with the other examples in the test app.

Creating our own file loader does have an advantage—we can place the code in the framework project. Let's consider for a moment why that is possibly a better solution. If you consider the standard Forms `Image`, we're assigning the image source to a URL with a result that comes from the view model. In other words, we're binding the result of a data grab to the image's `SourceProperty`. With `FFImageLoading`, while we may be doing some extra things with the `CachedImage`, we're still binding to the `SourceProperty` in the same way as we do for an `Image`.

Would we therefore not be better off with the image code in the framework project? After all, we are doing all of our web data–grabbing from the framework. We essentially set a property and broadcast when complete, which updates the view.

Can We Do Anything Else to Increase the Performance?

As it stands, the app retrieves all information as and when it is required. While this is more efficient in terms of network activity (you only get what you request), it does cause bottlenecks when it comes to performance, as the UI has to wait until all of the data is down for that particular activity.

If you consider the user experience, will the user be bothered by a couple of extra calls if all of the data, which allows the app to run smoothly, is being retrieved in the background? Chances are, they won't care!

QUICK EXERCISE FOR YOU—TIME THE APP AS IT STANDS AND THEN ALTER IT SO ALL OF THE DATA IS GRABBED AT THE OUTSET. SEE WHICH RUNS THE BEST

We can also speed things up by storing all of the data in the sqlite database and only running the webservice downloads for the images from the monitoring stations infrequently (roughly every four hours, so you grab at 9 a.m. and until it is 1 p.m. use only that data from the database). Again, this does mean that we will have stale data, but if an image is only taken every four hours, what is the point of downloading all of the image data again?

Conclusion

As you can see, it is not only possible to use MVVM Light with Xamarin Forms, but in some respects it makes more sense than trying to use your own framework. I would, however, say that given how Forms works, I would advise you not use any MVVM framework until you're comfortable with how Forms operates. It will needlessly complicate the learning process.

CHAPTER 9

Rounding Things Off

There are a couple of things we need to look at before signing off. In this chapter, we will be looking at how to deal with the application lifecycle, including handling data when the app is asleep. Later in the chapter, we'll examine other tasks, such as playing audio. Before we do any of that though, it's worth looking at a new development (at the time of writing this book)—the use of Shared Class Libraries over Portable Class Libraries and how you can move between them.

PCL Versus SCL—What's the Difference?

When we use a PCL, we have limitations, those being that the methods used must be available on all platforms that the PCL supports. Therefore, we don't have the likes of the `File` class; these have to be implemented via an interface and DI in order for the PCL to support something as simple as saving a file that we've downloaded from the internet.

Note The SCL is important and is the way Microsoft is taking shared libraries. You will also see something referred to as the .NET Shared Class Library; the SCL is that library. SCL is the more common name.

An SCL says that there will be methods available such that if you want to do something platform specific you can, you just need to wrap them in an `ifdef` condition.

The following would be fine within an SCL, whereas it wouldn't work in a PCL, as there is no guarantee that the `#if` directives are defined for a particular target:

```
public string Platform
{
    get
    {
        var platform = "ZX Spectrum";
        #if WINDOWS_PHONE
            platform = "Windows Phone";
        #else
        #if __ANDROID__
            platform = "Android";
        #else
        #if __IOS__
```


P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5_9

```
            platform = "iOS";
        #endif

        return platform;
    }
}
```

There is a small drawback with this in that the more platform-specific code you have in the SCL, the messier it will become. There is also an issue when defining an assembly name. Say you have a XAML UI design in the project and want to reference a control and an assembly name. Each native project will have a different assembly name (say, `MyApp.Android`, `MyApp.iOS`, and `MyApp.UWP`), which means there will need to be more `#ifdefs` or `OnPlatform` conditions. The other option is to have everything use the same assembly name. This can be a major headache and really should be avoided.

Removing the Need for #if Statements

As with a PCL, you can remove the need for compiler directives if you choose certain versions of the SCL. This table can be used as a quick reference.

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
UWP	10.0	10.0	10.0	10.0	10.0			

If you wanted to support all of the mobile platforms, you could only choose up to the .NET SCL version 1.4. All of the APIs from the version up to and including the version you choose will be available to you. This is the same as if you were targeting the .NET 4.6.2 framework; all APIs up to and including 4.6.2 are available to you.

If a box has nothing in it (UWP past 1.4 of the SCL), it is unsupported by that platform currently.

SCL and NuGet Packages

This is where things change greatly. With a PCL, the NuGet package typically needs to be installed both on the platform and in the PCL. As the SCL is shared, NuGet packages are installed onto the platform only.

At first sight, this will make your project look different (there isn't a Packages or References directory to start with). This is fine, as the packages are held on the platform, and the shared library just links back to the platform it is being called from.

■ **Note** If you're from a C background, this should be very familiar to you!

SCL and MVVM Light

MVVM Light is available for SCL 1.0 (it therefore covers UWP as well as Android and iOS).

Let's create a new package. There are two ways to do this. The first is to convert an existing PCL to an SCL. In my example, I'm creating a PCL to start with. The first step is to create a new project. When the project has been created, we need to convert this to use SCL.

To do this, highlight the PCL library within the Solution Explorer and select Options from the drop-down menu. You will be presented with the screen seen in Figure 9-1.

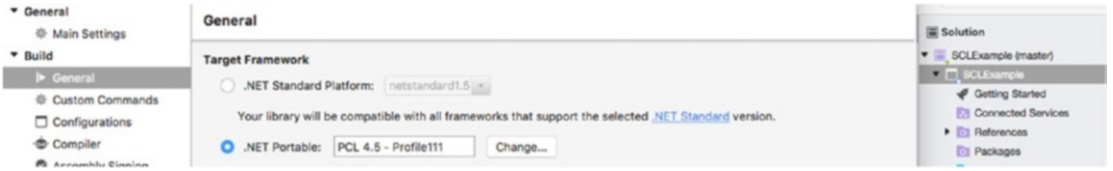


Figure 9-1. *Changing the target framework on VS for Mac*

At the moment, it's using the .NET portable PCL 4.5 Profile 111. We need to change this to use .NET SCL. To do this, select the radio icon next to .NET Standard Platform. As we want to target 1.4, click on the drop-down and select 1.4, as shown in Figure 9-2.

Figure 9-2. *Chosing the .NET standard platform*

When you're happy, click *OK* button.

■ **Note** Once you have moved to .NET SCL, it is still possible to go back to a .NET portable library, but it's not simple (you must recreate the project and copy over all of the directories from the SCL).

We next need to install MVVM Light. Select Packages and launch NuGet Manager. In the search box, type *mvvmlightlibsstd10*. As it's in pre-release, you'll need to select the "Show pre-release packages" checkbox, as seen in Figure 9-3.

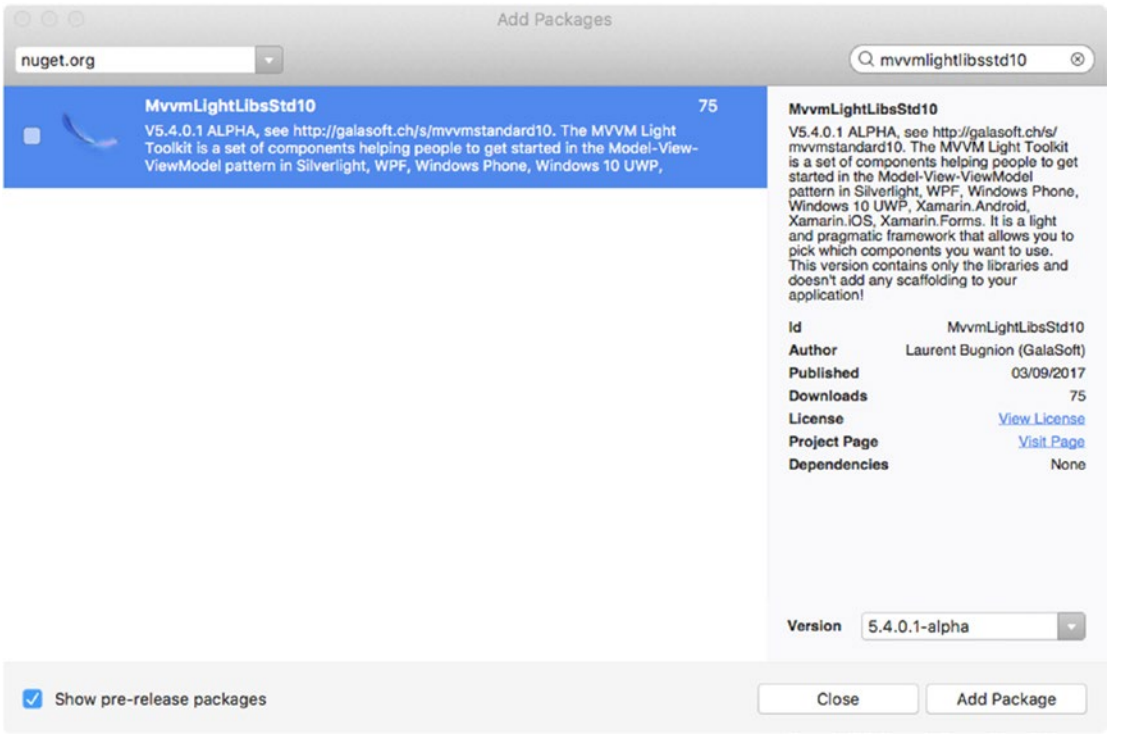


Figure 9-3. *Adding the SCL version of Mvvm Light*

Select the package and click on the *Add Package* button.

The SCL version of MVVM Light is now installed in what is now our SCL. As with the PCL version, this library also needs to be installed anywhere else it may be needed (in our case, on the platforms).

Where Has the CommonServiceLocator Package Gone?

A slight incompatibility between the PCL and SCL version results in `IServiceProvider` being unavailable in the SCL (`IServiceProvider` is found in `CommonServiceLocator`). The decision was taken to remove the interface, and so `CommonServiceLocator` is not required.

Changing Your Code to Run with the SCL

Very little is required for the project to run with the SCL version. The main changes are made in the `ViewModelLocator` class, as follows:

1. Remove `using Microsoft.Practices.ServiceLocation` and replace with `using MvvmLightWithPcl.Model`.
2. Remove `ServiceLocator.SetLocatorProvider(()=>SimpleIoc.Default);`.
3. Remove all instances of `ServiceLocator.Current.GetInstance` and replace with `SimpleIoc.Default.GetInstance`.

Converting an Existing Package to Use the SCL Version

This is essentially the same as just described, the only difference being that you need to uninstall all instances of the packages `CommonServiceLocator` and `mvvmlightlibs` and install `mvvmlightlibsstd10`.

Creating the SCL Without Converting a PCL

The process is similar except that when given the choice of PCL or SCL, you choose SCL. See Figure 9-4.

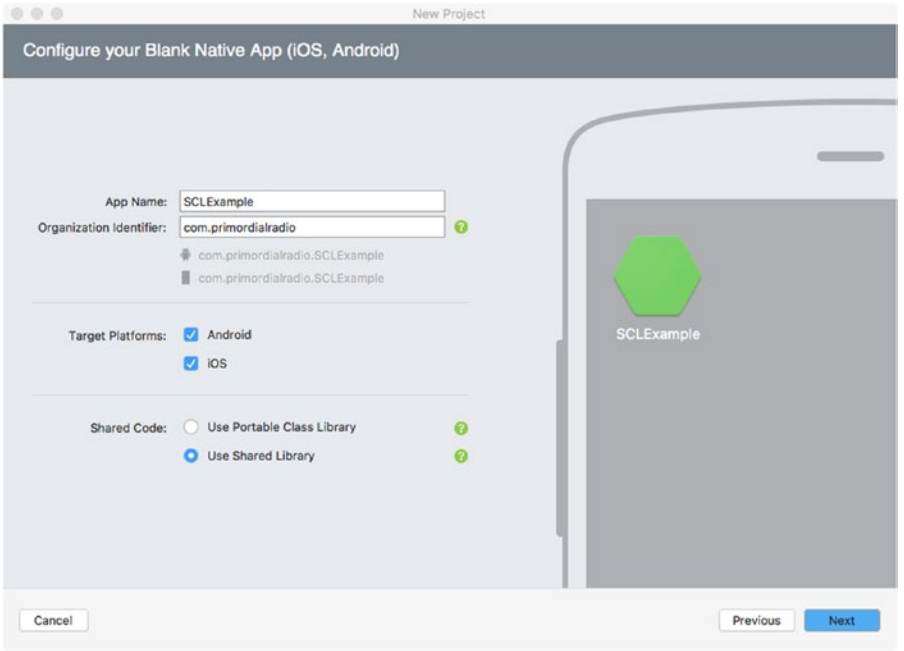


Figure 9-4. *Configuring your new native app*

The `mvvmlightlibsstd10` package will then need to be added to just the platforms (there isn't a Packages or References directory within the SCL).

A Practical Example: Playing Audio

The SCL_Audio source code is very simple. It has a couple of buttons on the UI. Click on a button, and audio plays!

The difference this time is that we're using an SCL, and in the SCL view model we have platform-specific code for handling a change in where to find the file:

```
void SetFilename()
{
    var filename = CurrentButton == 3 ? Filenames[RandomNumber] : Filenames[CurrentButton];
#if WINDOWS_PHONE
```

```
    filename = $"Assets/Audio/{filename}";
#endif
    FilenameToUse = filename;
}
```

When the SCL is built for UWP, the compiler directive will change the filename from being, say, `audio_1.mp3` to `Assets/Audio/audio_1.mp3`.

We are also handling the button clicks differently than we've seen previously. Normally, you may expect the button to be handled like this:

```
btnRandom.SetCommand("TouchUpInside", ViewModel.BtnPlayRandom);
```

However, we're not using a `RelayCommand` within our code, and there is no simple way to bind a `Click` event to a function that checks on when a property changes to call a new method.

Let's look at the event-handling routine:

```
void BindButtons()
{
     btnAudioOne.TouchUpInside += SetClicked;
     btnAudioTwo.TouchUpInside += SetClicked;
     btnAudioThree.TouchUpInside += SetClicked;
     btnRandom.TouchUpInside += SetClicked;
}
```

It is simple and compact to handle the event, with the event handler being equally short:

```
void SetClicked(object s, EventArgs e)
{
    var text = ((UIButton)s).TitleLabel.Text;
    ViewModel.ProcessButtonText(text);
    if (!string.IsNullOrEmpty(ViewModel.FilenameToUse))
    {
         PlayAudio(ViewModel.FilenameToUse);
    }
}
```

Can we alter this to use the standard binding model? Typically, to bind an event for a button we need the following:

```
btnRandom.SetCommand("TouchUpInside", ViewModel.RelayCommandHandler);
```

We're not using a relay command, so let's create one that calls the method we want in the view model:

```
btnRandom.SetCommand("TouchUpInside", new RelayCommand(()=>ViewModel.
ProcessTextButton(btnRandom.TitleLabel.Text)));
```

(Remember, a `RelayCommand` is essentially the same thing as a standard .NET `Command`.)

This passes the code into the view model. What we need to do is get the data back out. Here, we can use `WhenSourceChanges` to fire off the play routine:

```
this.SetBinding(()=>ViewModel.FilenameToUse, BindingMode.TwoWay).
WhenSourceChanges(PlayAudio);
```

It's debatable which would be more efficient (since we have four buttons, we have to bind each to `SetCommand`). However, as we're hanging on `FilenameToUse`'s changing its value before progressing to the `PlayAudio` method, the entire `SetClicked` method can be safely removed. We aren't binding each click to an event either—rather, to a `RelayCommand`—so when the class goes out of scope, there isn't a need to unbind from the `Click` event either.

On a purely code basis, using the MVVM Light binding model should make the code faster and more efficient; however, the compiler (either Mono or Roslyn) is far more efficient in terms of optimization, so the chances are that there will be an improvement, but it'll be so small you'll not notice it!

Application Lifecycle Handling

Unless you're doing something that is going to take a lot of time or are streaming music, your app will at some point "go to sleep" (screen blanks to conserve battery life). When you next click on the app after waking, you expect the app to go back to the view it was already on.

This poses a number of problems for the developer, the biggest being the "how on Earth can I get MVVM Light to work with it?" In order to appreciate how we need to approach this, we first need to understand the app lifecycles for the mobile platforms. See Figure 9-5.

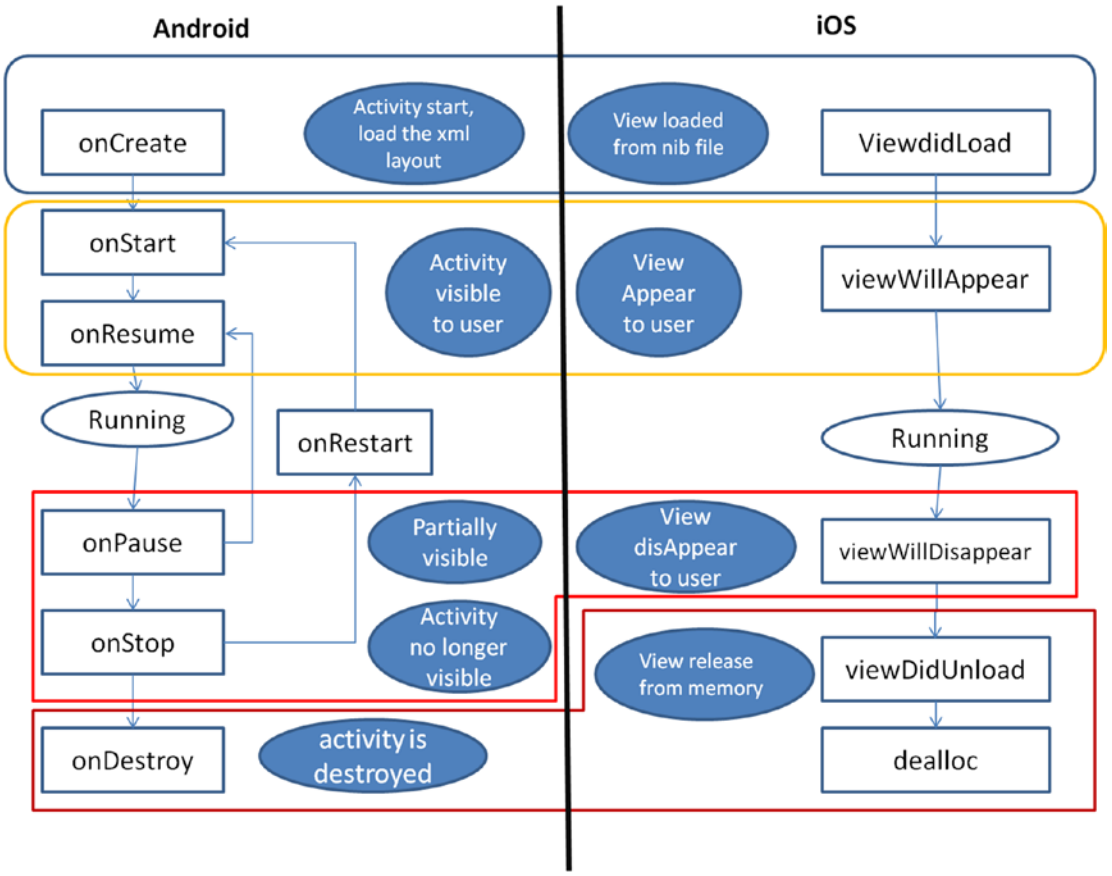


Figure 9-5. *App lifecycle (iOS and Android)*

Android and iOS are fairly simple, with the exception that as Android is Java based, the activity itself is destroyed and the GC does a cleanup. This continual destruction of activities is a bane for Xamarin.Android developers, as they are used to a .NET model rather than a Java model (which also doesn't promote the passing in of values to the constructor).

iOS deallocates the memory held for the UI when the class goes out of scope.

A secondary (though less important) aspect for iOS is that iOS uses Ahead of Time (AoT) compilation rather than Just In Time (JiT) . For AoT to work, all of the data for the view must be available.

UWP has a different model again. See Figure 9-6.

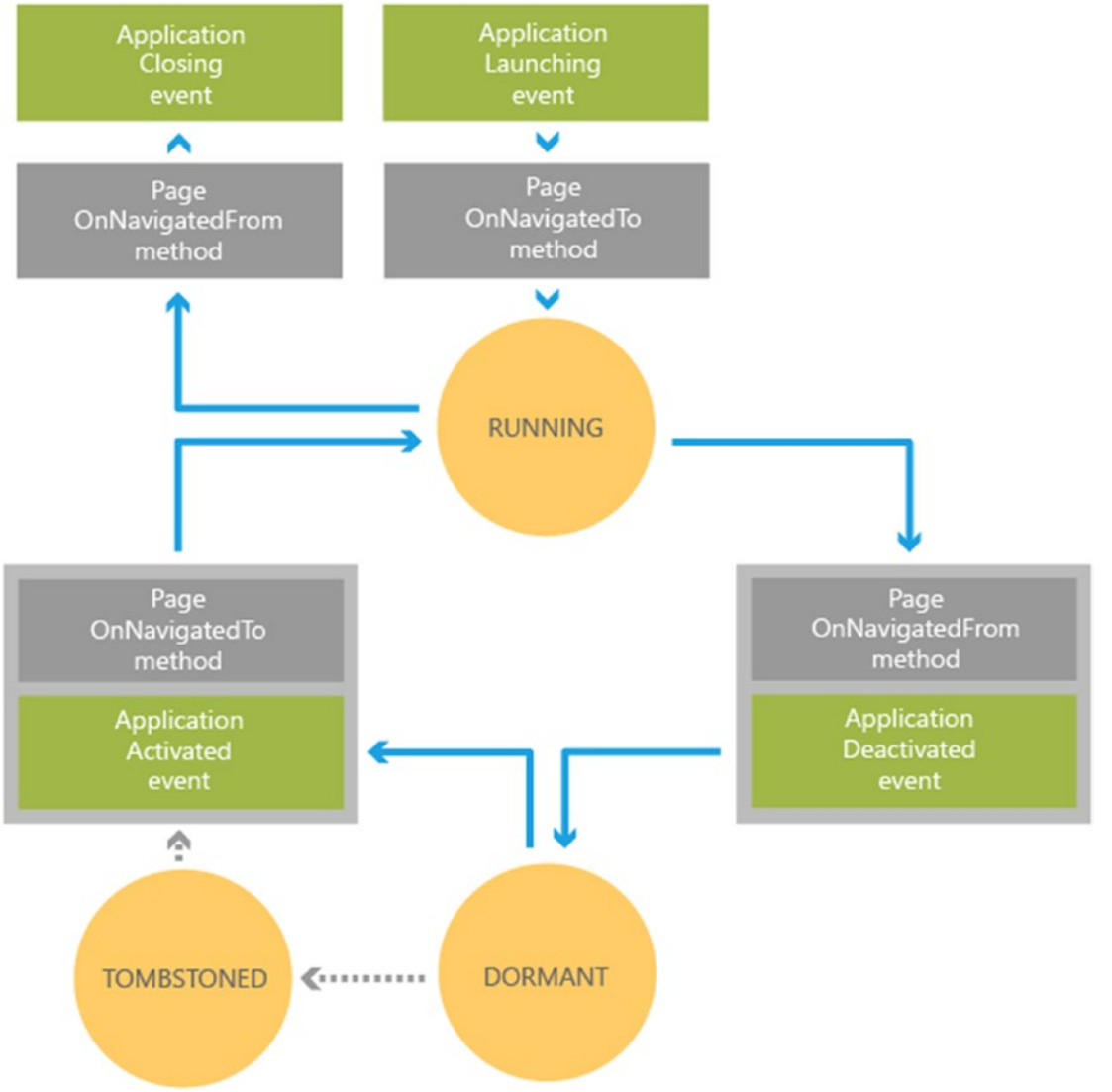


Figure 9-6. *App lifecycle for UWP*

UWP has two modes for an app in hibernation—Dormant and Tombstoned (don't worry about the name Tombstoned; it doesn't mean that the app is dead, but rather refers to how the app needs to deal with the recovery from Dormant and get back into action).

They all follow (roughly) the same route: app starts, UI launches, waits for a period of time, goes to sleep. If the app is not restarted, it remains there (iOS will now kill the app after a period of time unless it is performing background tasks or being kept alive artificially).

The different platforms deal with things in different ways.

iOS

iOS is nice and simple when handling the sleep and wake-up systems. There are four overridden methods that deal with it and one for the change of state.

Method	Action
`OnResignActivation`	Called when the app is about to move from active to inactive state
`DidEnterBackground`	Called when the app enters the background. It releases shared resources, saves user data, invalidates timers, and stores the state
`WillEnterForeground`	Called when the app moves from background to foreground
`OnActivated`	Restarts any paused tasks
`WillTerminate`	Called when app is about to terminate

Happily, on iOS there is only a single point where the app moves in and out of the active state. We really only need to store which view controller the app was displaying before moving to the inactive state.

UWP

UWP has three application states: running, not running, and suspended. The app is suspended when it is minimized or you have swapped to another app. The user thinks the app is running in the background when in reality the OS has freed up resources by moving the app into a hibernation mode. The app's threads are stopped, and its state is held in memory. The beauty of this is that when returning to the app the user is none the wiser, and the app is restored from not running to a running state.

There are two events that handle the switching: `Application.Suspending` and `Application.OnLaunched`. There isn't a terminated event. These events are handled within `App.xaml.cs`.

UWP adds in a couple of other helpers for the developer. When an app is killed it moves from running to not running. At this point, the app may have crashed or been forced to close. Alternately, the app has been closed by the user. In this second case, the app has been suspended, and the `Suspending` event called.

In order to keep the user experience, UWP places a five-second suspend rule on the suspending event. If the deadline is not hit, the app dies. This is a problem if an async call is made within the `Suspending` event, as the chances are that the call won't be finished within the five seconds. To solve this problem, the `GetDeferral` method is called. This essentially tells the `Suspending` event to not terminate when the async call returns but rather to wait until the `Completed` method is called or the five-second deadline is hit. The important thing to remember is that `GetDeferral` works on whichever (`Completed` or five-second deadline) comes first.

Relaunching an app is handled with the `OnLaunched` method. If the app was suspended, then the `Resuming` event is triggered; otherwise, `OnLaunched` is called. `OnLaunched` receives the `LaunchActivatedEventArgs` object, which contains the last state of the app prior to launching. The property name is `PreviousExecutionState` and has the values seen in the table.

PreviousExecutionState	What it does
NotRunning	App launched for the first time or was killed in an unexpected way, so essentially this is a first launch
Terminated	App previously suspended and then shut down. It is not the same as a suspended app, but was killed by the OS to free memory.
ClosedByUser	Intentionally closed by the user
Running & Suspended	App launched from another tile when it is already running

As with iOS, the app handles these events in one place, so we need to store the view in use when it was suspended.

Android

Unlike UWP and iOS, all Android activities have `OnPause` and `OnResume` overrides, which is both good and bad. It's good in that the app restarts on the activity it was suspended on. It's bad in that *every* activity needs `OnPause` or `OnResume` coded for.

Thankfully, we can use code similar to that used within some of our view model classes elsewhere; we have a `BaseActivity` that deals with the `OnPause` and `OnResume` classes.

How Can We Use the MVVM Model for the Lifecycle?

When you consider how lifecycles work, we will always need to know first, which view we are in and second, any relevant states.

We are already using the built-in navigation service to deal with movement (unless you're in forms where it can be more a hinderance than a help) and to store code within the view model properties. Typically, the `INavigation` interface is passed into a view model, which will help.

The simplest method of storing the view model is going to be to serialize on sleep and deserialize on waking. Using JSON.NET (available via NuGet), the entire class can be serialized like this:

```
var serializer = new JsonSerializer();
serializer.Converters.Add(new JavaScriptDateTimeConverter());
serializer.NullValueHandling = NullValueHandling.Ignore;
using (var sw = new StreamWriter(my_file_and_path))
{
    using (var writer = new JsonTextWriter(sw))
    {
        serializer.Serialize(writer, product);
    }
}
```

with the deserializer step being as follows:

```
T deserializedProduct = JsonConvert.DeserializeObject<T>(output);
```

(T is the class to deserialize to—in this case, it's whichever view model was serialized originally).

It remains on the wake step to interrogate the navigation stack and navigate to the last used point.

Threading: Let Buyer Beware

I've left this topic until last as it's not something many people really worry about—threading.

If you think about something as simple as binding to a `Text` property in the UI, you can see that the MVVM library is a multi-threading system. In 99.999 percent of cases, this isn't an issue; when a `Text` property gets data from a property, the chances are that the property will be non-null, and when one of the entry widgets is used, it will be a one-to-one relationship between the widget and view model property with no form of thread-locking required.

Consider the following though. The UI sends a request to the view model to go grab some data from somewhere. This will be on, say, thread 1. When the view model starts working, it will most likely not be on that thread, but on a number of threads (especially if the data grab is asynchronous). The nature of threads is that there is never a guarantee that at any one point the thread is complete and an answer given. Let me explain a bit better.

Threads are easiest when thought of graphically (at least I think so). We start with our main thread, seen in Figure 9-7.

Figure 9-7. *A single thread*

The main thread goes from the start of the application to the end of the application. At any point on our main thread, we can create a new thread (or new threads if required). See Figure 9-8.

Figure 9-8. *Multi-threads*

These two new threads can do anything the application needs them to do. There is a simple rule though—the threads can only last as long the application does. As Figure 9-8 shows, the threads start and carry on their merry way; there is nothing to say the thread has to rejoin the main thread, nor is there any rule to say at what point the thread returns (which can cause some very large thread-safety issues that can lead to panics).

It goes without saying that each thread can also spawn their own threads to perform sub-processes. See Figure 9-9.

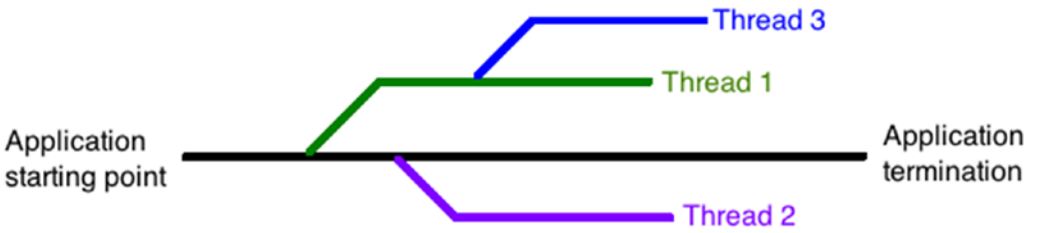


Figure 9-9. *A thread coming off a thread*

■ **Note** A thread can return to the main thread at any time, though this "at any time" can be disastrous to the safe running of the application (race conditions, properties being accessed or manipulated when they are null, etc.)

This is especially important when considering an asynchronous call. Unless a `ContinueWith` is added to the end of an `await` call, as soon as the `await` is hit, in general the flow returns to the next line, with the process being essentially thrown to its own new thread path. This new thread path will rejoin when it's done (if the operation is network or I/O bound, they don't use local threads). The time taken to return from the `await` may be a few seconds or less; irrespective of that though, if the thread that spawned the `async` needs to perform an action on the data and the data isn't there, an exception is thrown. If you further consider the speed of operation for a given device, a few seconds is a lifetime (or three). This is sometimes referred to as a race condition (which process will reach the desired goal first?).

How to Help with This Race Condition

Let's consider the following code (it's purely fictional, so don't worry if you'd never use it!):

```
var foo = await myService.GetListOfObjects();
foo = foo.SortBy(t=>t.property1).ThenBy(w=>w.property2).ThenBy(z=>z.property3).ToList();
return foo.First();
```

On the face of it, this code looks fair enough. We make the call, get some data, and sort and return the sorted list. The problem is, what happens if the request takes too long or returns nothing? Simple answer, crash.

Restructuring the code will help here. Let's first create a property for the list of objects:

```
MyList myListOfObjects;
Public MyList MyListOfObjects
{
    get => myListOfObjects;
    set
    {
        if (value != null)
        {
            var v = value.SortBy(t=>t.property1).ThenBy(w=>w.property2).ThenBy(z=>z.
            property3).ToList();
            Set(()=>MyListOfObjects, ref myListOfObjects, v, true);
        }
    }
}
```

We now have a null check and emitting of the PropertyChanged event when MyListOfObjects has been called.

Next, we modify the await:

```
await myService.GetListOfObjects().ContinueWith((t)=>
{
    if (t.IsCompleted)
        MyListOfObjects = t.Result;
});
```

We have now gotten a call on two threads, but without a race condition (as long as the PropertyChanged event is listed for then the property accessed).

As I said, for the vast majority of the time, this isn't required for consideration. However, any sort of code that runs on multiple threads must be at least aware of the pitfalls speed produces.

Conclusion

We've seen how we can use the new SCL within our code and the importance of understanding at least the basics of how threading works and how to interact with the state changes when moving in and out of the sleep/suspended state of an app. It's not that painful an affair when you look at it.

Index

P. Johnson, *Using MVVM Light with your Xamarin Apps*, https://doi.org/10.1007/978-1-4842-2475-5

I, J, K

L

M

N

O

P, Q

R

S

T

U

V

W

X, Y, Z