Advanced Cross-Platform Patterns

Xamarin Evolve, Chapter 12

Xamarin Inc.

Overview

Building cross-platform applications can be made even easier with a solution and project structure that supports and encourages code sharing.

This chapter will teach you:

- Overview of design patterns that are useful for cross-platform code sharing.
- How to structure a cross-platform solution for easy code sharing.
- What Portable Class Libraries are and how they work.
- About some 3rd party frameworks that help build cross-platform mobile applications with Xamarin.

The code samples for this chapter are mostly stand-alone examples that illustrate the patterns and architecture concepts being discussed. Three separate solutions are included.

- **TaskyProLinked** – a simple example of a cross-platform solution using linked files to share code.
- **TaskyProPortable** – a simple example of a cross platform solution utilizing Portable Library Project to share code.
- **FieldService** – a complete field service cross-platform application demonstrating the MVVM pattern.

We also discuss the MvvmCross project that is available from https://github.com/slodge/MvvmCross.

You should review the Building Cross Platform Applications and Sharing Code Options documentation for additional information about building cross-platform applications with Xamarin.

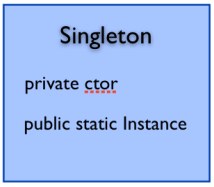
Patterns Overview

Here is a quick overview of the patterns discussed in this chapter.

Singleton

The Singleton pattern provides for a way in which only a single instance of a particular object can ever exist. Singletons are usually implemented in C# with a private constructor and a public static getter. Microsoft has guidance on implementing Singleton in C#.

Example: FieldService application – `ServiceContainer` class.

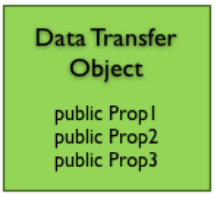


Data Transfer Object

Simple classes that generally only consist of public properties, used to model the data that your application stores in SQLite (or other database) and/or serializes for transfer to and from web services.

Microsoft has some notes on using Data Transfer Objects.

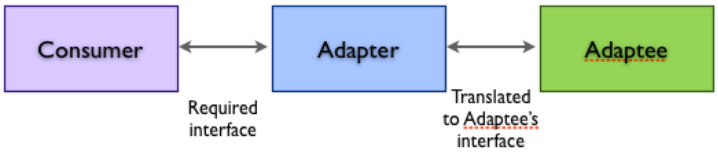
Example: FieldService app Data objects.



Adapter (or Wrapper)

Translates the interface or API from one class into a different interface so that it can be consumed by code that expects the different interface. The Adapter hides the interface of the class it wraps.

Example: iOS' UITableViewSource and Android's ListView.Adapter

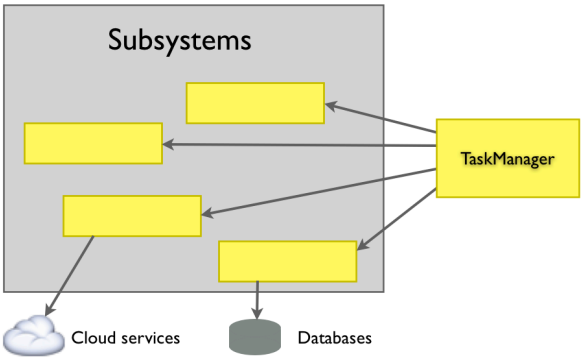


Façade (or Manager)

Provides a simplified point of entry for complex work. For example, in the Tasky sample application, there is a `TaskManager` class with methods such as `GetAllTasks()`, `GetTask(taskID)`, `SaveTask (task)`, etc. The `TaskManager` class provides a Façade to the inner workings of actually saving/retrieving of task objects.

In more complex examples, the Façade or Manager might aggregate methods & behavior from a number of different classes, or even combine results from cloud services or database queries.

Example: Tasky application – `TaskManager` class.

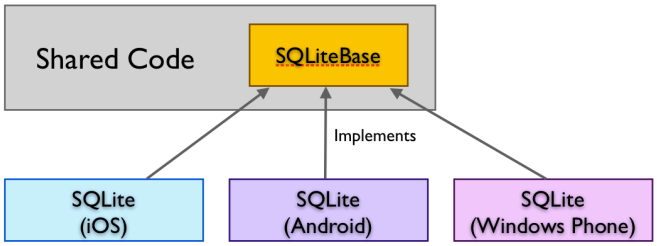


Provider

A pattern coined by Microsoft (arguably similar Dependency Injection) to allow developers to extend their core feature set. The NET 2.0 version of the Provider pattern allowed new implementations configured in the application configuration (eg. SessionStateProvider, Membership and Roles for ASP.NET).

In the same vein, we mean Provider to refer to a statically declared type implementation, not necessarily one that changes at runtime such as a Strategy pattern implementation

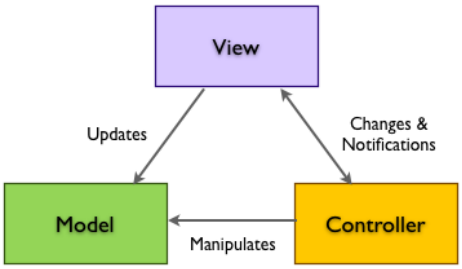
Example: TaskyPortable – `SQLiteProviders`.



Model View Controller (MVC)

A common and frequently misunderstood pattern, MVC is most often used when building user interfaces. MVC provides for a separation between the actual definition of a UI Screen (View), the engine behind it that handles interaction (Controller), and the data that populates it (Model). The model is actually a completely optional piece and, therefore, the core of understanding this pattern lies in the View and Controller.

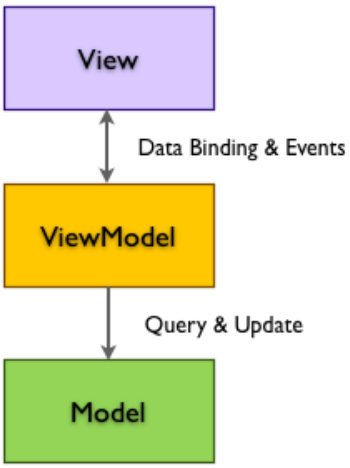
Example: iOS `UITableViewController`.



Model View ViewModel (MVVM)

A variant of the MVC introduced by Microsoft to describe their recommended approach for building applications with XAML.

Example: FieldService application.



Project Structure

Sharing code across different projects requires a way for the common C# files to be managed. There are two main ways to structure your C# solutions to share code across platforms:

- File Linking
- Portable Class Libraries

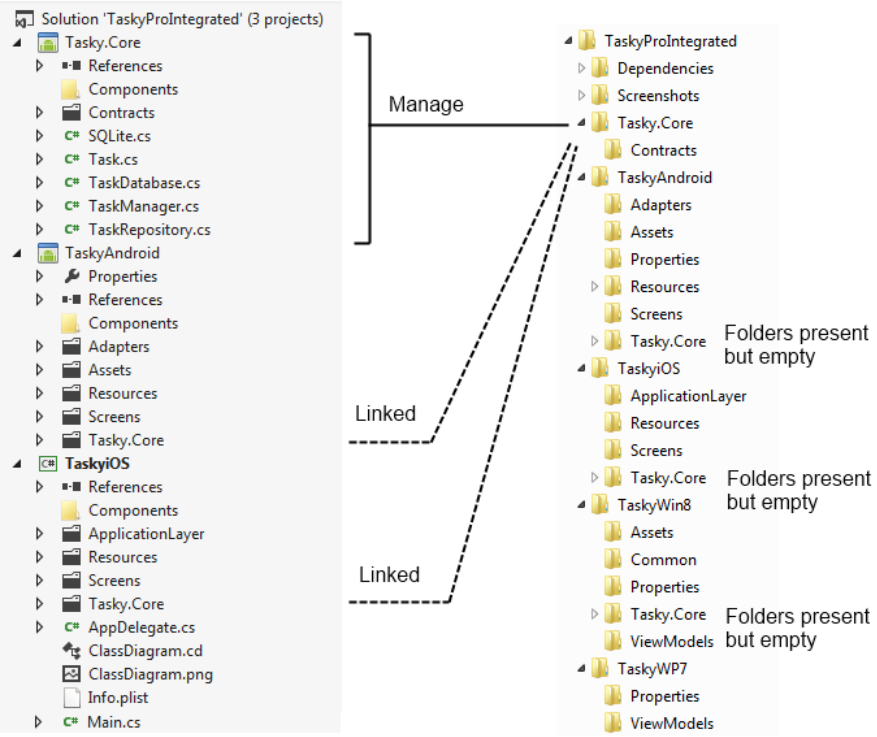
The benefits and pitfalls of these two methods are discussed below, and additional information can be found in the Sharing Code Options documentation.

File Linking

The simplest approach to sharing code files is to place them in a separate directory, and then use file-linking to include them directly into each of your mobile application projects. When a file is added, moved, or deleted, then each mobile application project must be manually updated to reflect the change. You could

also link files into separate library projects if your code base is large and you wish to organize it across multiple assemblies.

This method allows you to share the same code across different platform projects, and then use compiler directives to include different, platform-specific code paths.



Benefits

- Allows you to share code across multiple projects.
- Platform-specific projects can each have different compiler directives defined, so you can branch code based on the platform (eg. using `#if __ANDROID__`, `#if __IOS__`, `#if __MOBILE__`).
- Platform-specific projects can include platform-specific references (such as using `Community.CsharpSqlite.WP7` in the FieldService, Tasky, MWC samples). The code using these references can be managed with compiler directives.

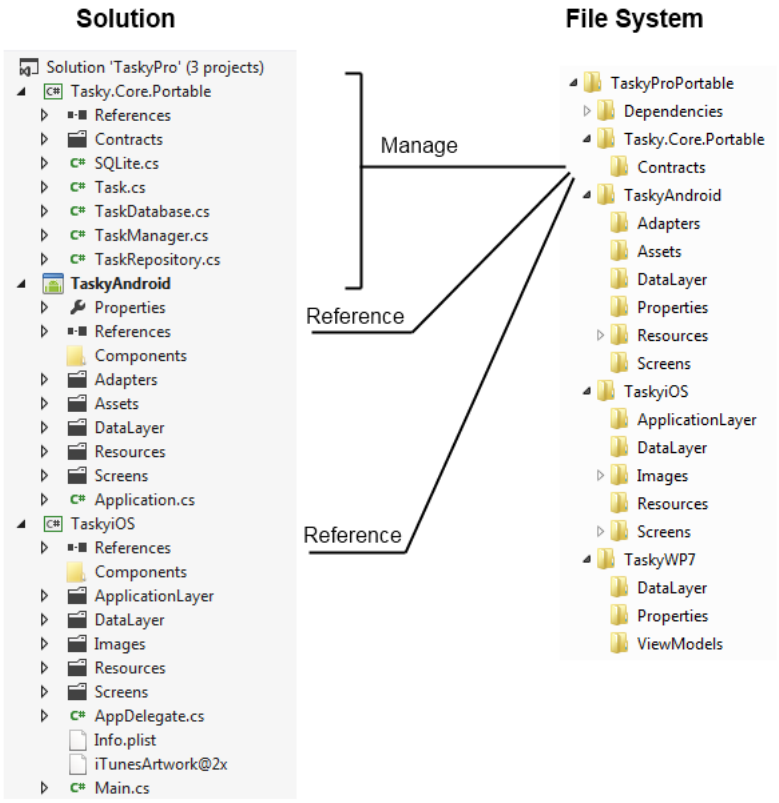
Disadvantages

- In Visual Studio there is no simple way to add an entire filesystem 'tree' to a project, so files must be individually added. The directory hierarchy must also be manually created.
- If you add, delete or move a file in one project you must remember to perform the same action in all the other project files.
- Refactoring will only work within one platform type.

- The project file must live in a separate directory to enable files to be linked. Empty subdirectories must also exist before you can 'link' files contained within a subdirectory.
- If you forget to choose **Add As Link** (in Visual Studio), it will *copy* the file into your target project instead of linking. If you don't notice this has occurred you could accidentally edit both the original file and the copy, getting them out-of-sync.

Portable Class Libraries

Place the shared code in a class library (or libraries) that are separate from the platform-specific applications. The apps then reference these libraries to share the code. Portable Class Libraries contain code that will run unmodified on all the platforms. These libraries are limited in what namespaces, classes and methods they can use. The technical details of how Portable Class Libraries work will be discussed later in this chapter.



Benefits

- Only need to add files once to the shared project – all the application projects reference the same assembly.
- Refactoring operations will affect all code loaded in the solution (the Portable Class Library and the platform-specific projects).

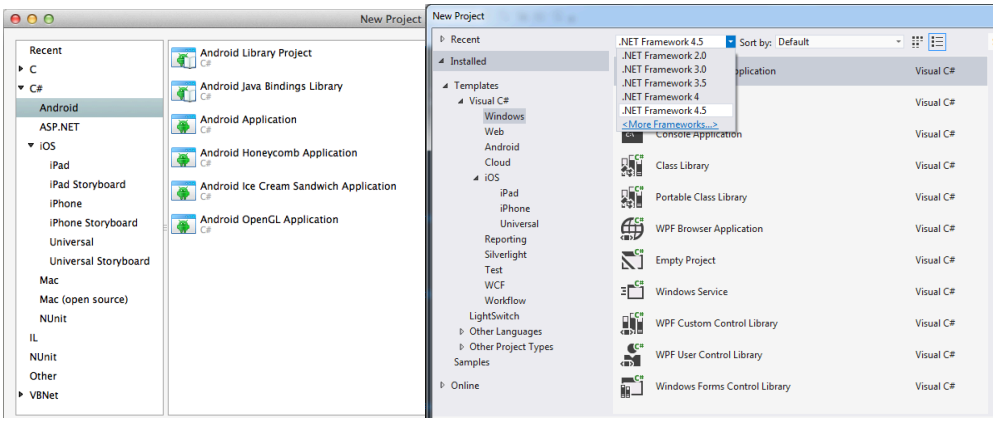
Disadvantages

- Because the same Portable Class Library is shared between multiple applications, platform-specific libraries cannot be referenced (eg. Community.CsharpSqlite.WP7).
- The Portable Class Library subset may not include classes that would otherwise be available in both MonoTouch and Mono for Android (such as `DllImport`).

To some extent both disadvantages can be circumvented using the Provider pattern or Dependency Injection to code the actual implementation in the platform projects against an interface or base class that is defined in the Portable Class Library.

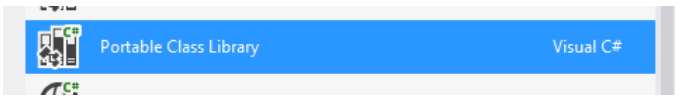
Portable Class Libraries

Regular .NET assemblies (or class libraries) consist of managed code that has been compiled against a specific platform and .NET version. When you create a new library project in Xamarin Studio or Visual Studio, you choose the target platform as part of the **File > New Project** dialog, as shown here (Xamarin Studio and Visual Studio respectively):

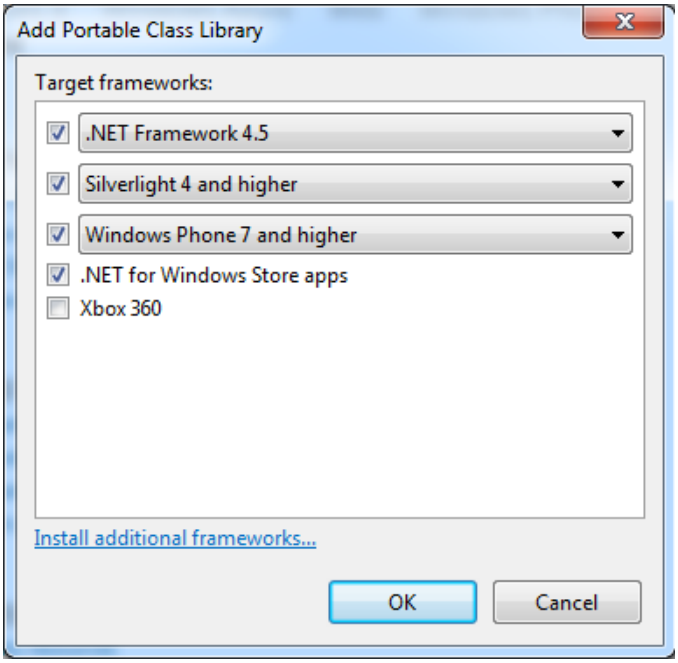


The resulting assembly can then be referenced by other compatible libraries or applications (eg. the referencing assembly or application must generally be targeting the same platform using the same or newer version of the .NET Framework).

Alternatively, it is possible to create a Portable Class Library from within Visual Studio, by selecting this option:



Which prompts for further clarification about exactly which platforms the library should support, as shown in this screenshot.



The choices you make about which platforms the Portable Class Library should support will then affect what features you can include in your code – the IDE will ensure your code only contains the lowest-common-denominator features so that the Library can run on any of the chosen platforms.

> Xamarin is in the process of adding more advanced IDE support for Portable Library Projects to Xamarin Studio.

What is a Portable Class Library?

The platform-support choices you make when creating a Portable Class Library are translated into a "Profile" identifier, which describes which platforms the library supports. The table below shows some of the features that vary by .NET platform. To write an assembly that is guaranteed to run on specific devices/platforms you simply choose which support is required when you create the project.

Feature	.NET Framework	Windows Store Apps	Silverlight	Windows Phone	Xbox 360
Core	Y	Y	Y	Y	Y
LINQ	Y	Y	Y	Y	
IQueryable	Y	Y	Y	7.5+	
Dynamic	4.5+	Y	Y		
Serialization	Y	Y	Y	Y	

Data annotations	4.0.3+	Y	Y		
PCL aliases	Net4 Net403 Net45	NetForWSA	SL4 SL5	WP7 WP75	Xbox360

The combination of supported platforms is called a “Profile”.

There are many different Profiles that are defined for each different combination of platform support. Some examples are given below, but there are many more Profiles defined for all the possible permutations of support.

Profile1 (NetForWSA, Net4, SL4, WP7, XBox360)

Runs on: .NET Framework 4.0, Windows Store, Silverlight, Windows Phone 7 and Xbox360.

A portable class library compiled for this profile will be re-usable across the most platforms, but will also be very restrictive in terms of what functionality can be included in the assembly (mainly due to the requirement to support the Xbox).

Profile2 (NetForWSA, Net4, SL4, WP7)

Runs on: .NET Framework 4.0, Windows Store, Silverlight, Windows Phone 7.

Class libraries using this profile can include a much wider range of .NET framework features, but it is only defined for .NET 4.0 and so does not support newer C# language features.

Profile104 (NetForWSA, Net45, SL4, WP75)

Runs on: Windows Store, .NET 4.5, Silverlight and Windows Phone 7.5.

This profile is similar to **Profile2** but supports newer versions of .NET and Windows Phone, so code written against this profile can use the `dynamic` keyword and `IQueryable` interface.

Current efforts to get Portable Class Libraries working with Xamarin tend to emulate the Profile104, however it is not a perfect match because Xamarin cannot support `dynamic`. Xamarin will provide more guidance about appropriate profiles to share code when the Portable Class Support is finished in Xamarin Studio. Until that support is available you can manually modify your Xamarin Studio configuration according to these instructions: http://stackoverflow.com/questions/12041290/monodevelop-is-it-possible-to-switch-pcls-compiler/.

Example

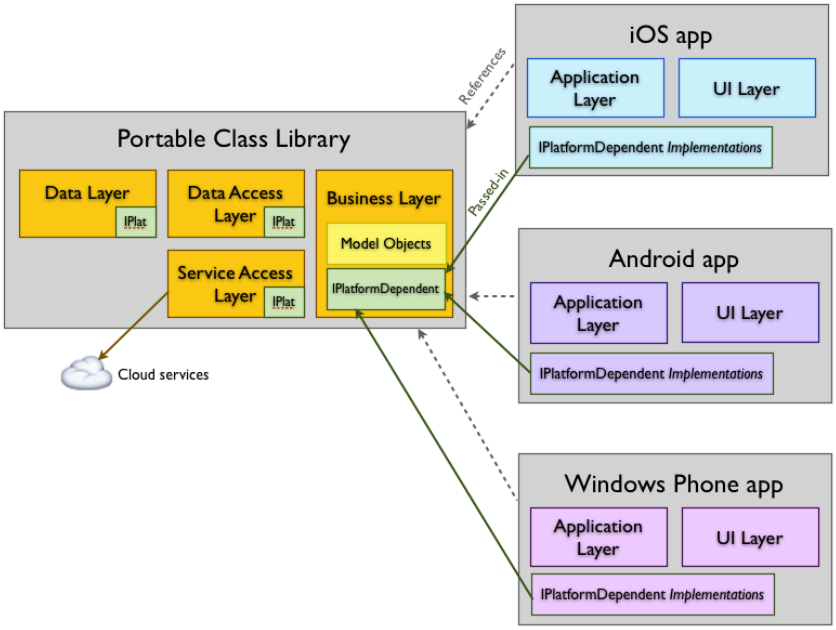
A cross platform application that supports iOS, Android and Windows Phone would require a single shared-Portable Class Library project file and three application projects (one for each platform).

Although Xamarin Studio's support for Portable Class Libraries is currently incomplete, it is possible to create one using Visual Studio.

A solution using a Portable Class Library would contain the following projects:

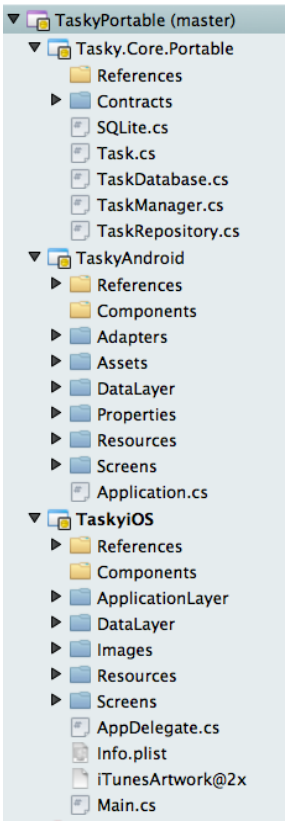
- Core.Portable – Portable Class Library containing all shared code.
- AppAndroid – Mono for Android application that references Core.Portable.
- AppiOS – MonoTouch application that references Core.Portable.
- AppWP – Windows Phone application that references Core.Portable.

If the Core code needs to interact with platform-specific features, it must define an interface or abstract class to represent those features. The interface or abstract class is then implemented in each platform-specific application and passed into the Core library when required.



PCL Example

Refer to the TaskyPortable sample application to see how a Portable Class Library can be used with Xamarin. The solution structure is shown below:



The TaskyProPortable example code uses the "platform-independent implementations" approach with the SQLite-NET library. For demonstration purposes it has been refactored into an abstract class that can be compiled into a Portable Class Library, and the actual code implemented as subclasses in the iOS and Android projects.

Tasky.Core.Portable

The Portable Class Library is limited in the .NET features that it can support. Because it is compiled to run on multiple platforms, it cannot make use of `[DllImport]` functionality that is used in SQLite-NET. Instead SQLite-NET is implemented as an abstract class, and then referenced through the rest of the shared code. An extract of the abstract API is shown below:

```
public abstract class SQLiteConnection : IDisposable {
    public string DatabasePath { get; private set; }
    public bool TimeExecution { get; set; }
    public bool Trace { get; set; }
    public SQLiteConnection(string databasePath)
    {
        DatabasePath = databasePath;
    }
    public abstract int CreateTable<T>();
    public abstract SQLiteCommand CreateCommand(string cmdText, params
object[] ps);
    public abstract int Execute(string query, params object[] args);
    public abstract List<T> Query<T>(string query, params object[] args)
where T : new();
```

```
    public abstract IEnumerable<T> DeferredQuery<T>(string query, params
object[] args) where T : new();
    public abstract List<object> Query(TableMapping map, string query,
params object[] args);
    public abstract IEnumerable<object> DeferredQuery(TableMapping map,
string query, params object[] args);
    public abstract TableQuery<T> Table<T>() where T : new();
    public abstract T Get<T>(object pk) where T : new();
    public bool IsInTransaction { get; protected set; }
    public abstract void BeginTransaction();
    public abstract void Rollback();
    public abstract void Commit();
    public abstract void RunInTransaction(Action action);
    public abstract int InsertAll(System.Collections.IEnumerable objects);
    public abstract int Insert(object obj);
    public abstract int Insert(object obj, Type objType);
    public abstract int Insert(object obj, string extra);
    public abstract int Insert(object obj, string extra, Type objType);
    public abstract int Update(object obj);
    public abstract int Update(object obj, Type objType);
    public abstract int Delete<T>(T obj);
    public void Dispose()
    {
        Close();
    }
    public abstract void Close();
}
```

The remainder of the shared code uses the abstract class to “store” and “retrieve” objects from the database. In any application that uses this abstract class we must pass in a complete implementation that provides the actual database functionality.

TaskyAndroid & TaskyiOS

On iOS and Android the Sqlite database engine is built-in to the operating system, so the implementation can use `[DllImport]` as shown to provide the concrete implementation of database connectivity. An excerpt of the platform-specific implementation code is shown here:

```
[DllImport("sqlite3", EntryPoint = "sqlite3_open")]
public static extern Result Open(string filename, out IntPtr db);
[DllImport("sqlite3", EntryPoint = "sqlite3_close")]
public static extern Result Close(IntPtr db);
```

The full implementation can be seen in the code sample project.

TaskyWP7

In contrast to the implementation used for iOS and Android, the Windows Phone implementation must create and use an instance of the Community.Sqlite implementation. Rather than using `DllImport`, methods like Open are implemented against the Community.Sqlite assembly that is referenced in the **TaskWP7** project. An excerpt is shown here, for comparison with the iOS and Android version above.

```
public static Result Open(string filename, out Sqlite3.sqlite3 db)
```

```
{
    db = new Sqlite3.sqlite3();
    return (Result)Sqlite3.sqlite3_open(filename, ref db);
}

public static Result Close(Sqlite3.sqlite3 db)
{
    return (Result)Sqlite3.sqlite3_close(db);
}
```

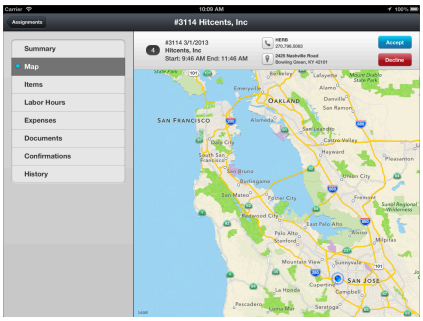
Model View ViewModel (MVVM)

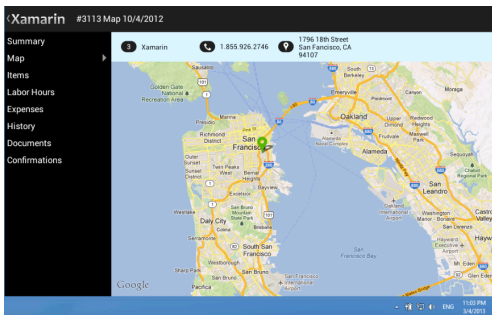
The Model View ViewModel pattern was coined by Microsoft to describe their preferred approach for rendering business objects (the Model) using XAML and its associated databinding capabilities. Whereas the earlier MVC pattern has the Controller orchestrating user interface interactions, in MVVM the XAML and databinding concepts declaratively "wire-up" the user interface controls. The ViewModel class sits between the databinding (View) and the Model to perform business logic.

The MVVM pattern looks a little different in Xamarin, which does not have native "databinding" available, but it's possible to structure your code so that you can create Model and ViewModel code to share across iOS, Android and Windows platforms.

FieldService Pre-Built App

Xamarin offers a pre-built Field Service app (included in the samples for this chapter, and available at http://xamarin.com/prebuilt/fieldservice). The application looks like the following screenshots on iOS and Android:





The app uses a number of patterns and practices that help to share lots of code across all three platforms: iOS, Android and Windows Store. The solution structure uses file linking (not Portable Class Libraries)

Shared Code

The shared code project contains the cross-platform classes that are shared across iOS, Android and Windows. These files are all linked into the three platform-specific projects.

Data – Data Transfer Objects that consist of properties we wish to store in the SQLite database.

ViewModels – Business logic and SQLite-NET calls to save and load objects in the database.

ServiceContainer – A singleton class that resolves ViewModel types,

Platform Application Code

Each application project contains the screens (Views) and mechanisms to bind ViewModels to them.

- **iOS** – The ViewModels are instantiated and properties copied back & forth to populate the user interface. Commands are manually wired up.
- **Android** – The ViewModels are instantiated and properties copied back & forth to populate the user interface. Commands are manually wired up.
- **Windows** – The XAML uses databinding to link the View to the ViewModels declaratively.

Following the patterns and examples set by this app should help you write easy-to-share C# code that can be re-used across Xamarin.iOS, Xamarin.Android and Microsoft platforms.

MvvmCross

MvvmCross is a framework that uses the MVVM pattern to but includes a user-interface engine that allows for cross-platform databinding to further reduce the amount of platform-specific code and let you write simpler views.

Uses Portable Class Libraries to share code across all platforms.

Available for download at https://github.com/slodge/MvvmCross including the complete source and a number of samples implemented for Xamarin.iOS and Xamarin.Android as well as Windows Phone 7, Windows RT and a WPF rich client.

Databinding Example

Using XAML on Windows Phone, a ListBox control using databinding expressions would look like this (databinding attributes are highlighted).

```
<ListBox Name="listBox" ItemsSource="{Binding List}" >
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <i:Interaction.Triggers>
          <i:EventTrigger EventName="Tap">
            <commandbinding:MvxEventToCommand Command="{Binding ViewDetailCommand}" />
          </i:EventTrigger>
        </i:Interaction.Triggers>
        <Image Stretch="None" Source="{Binding AmazonImageUrl}" Margin="3,5,0,5">
        </Image>
        <TextBlock FontSize="24" Text="{Binding Title}" Margin="5,3,3,0" TextWrapping="Wrap" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

The same effect can be accomplished using MvvmCross in iOS like this, where the databinding expression is a JSON string that maps controls, properties and commands:

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();

    var source = new MvxActionBasedBindableTableViewSource(
        TableView,
        UITableViewCellStyle.Subtitle,
        new NSString("BookListView"),
        "{'TitleText':{'Path':'Title'},'DetailText':{'Path':'Author'},'Select
        UITableViewCellAccessory.DisclosureIndicator);

    source.CellModifier = (cell) =>
                              {
                                  cell.Image.DefaultImagePath = "icon.png";
                              };

    this.AddBindings(
        new Dictionary<object, string>()
            {
                { source, "{'ItemsSource':{'Path':'List'}}" }
            });

    TableView.Source = source;
    TableView.ReloadData();
}
```

Refer to the website and examples for more information.

Strongly-Typed Example

Many MvvmCross examples will use the succinct JSON syntax for data binding, as shown here:

```
public const string BindingText =
@"{'TitleText':{'Path':'Name'},'DetailText':{'Path':'FullName'}}";
```

It is also possible to express the data binding in code, like this:

```
// if you don't want to JSON text, then you can use MvxBindingDescription
in C#, instead:
public static readonly MvxBindingDescription[] BindingDescriptions
= new[] {
    new MvxBindingDescription() {
        TargetName = "TitleText",
        SourcePropertyPath = "Name"
    },
    new MvxBindingDescription() {
        TargetName = "DetailText",
        SourcePropertyPath = "FullName"
    },
};
```

These examples are provided to give you an idea of how MvvmCross works. Read more about the framework before deciding whether it is appropriate for your app.

MonoCross is an open source framework that was also the foundation/inspiration of MvvmCross.

Download from http://monocross.net/. There is also a commercial version http://ifactr.com/ for enterprises.

Includes an abstract UI framework and the ability to execute business logic locally or remotely. Works with ERP products like SAP and includes HTML5 support so you target web clients (and other mobile platforms like Blackberry via their web browser).

There are samples implemented on Windows Phone 7, Windows RT and a WPF rich client. It also supports an HTML5 view and even a Console application view.

ITR Mobility are a Xamarin partner and have implemented the iFactr framework for many different clients. Check out the websites for more information on whether MonoCross or iFactr is appropriate for your business.

ReactiveUI

Check out http://www.reactiveui.net for a new take on building cross-platform user interfaces. Look out for more examples for Xamarin.iOS and Xamarin.Android once our async/await support comes out of beta.

Vernacular Localization Framework

Visit https://github.com/rdio/vernacular to learn about the sophisticated cross-platform localization framework used by Rdio in their apps.

Summary

In this chapter we've discussed some of the common patterns that are useful for structuring code in cross-platform mobile applications. We then compared the two different methods of code sharing - file linking and Portable Class Libraries – including a discussion of how Portable Class Libraries work.

Finally, we briefly introduced some 3rd party tools and frameworks that can help you implement cross-platform applications using Xamarin.