Introduction to Cross-Platform Development and Xamarin for Visual Studio

Evolve Fundamentals Track, Chapter 8

Xamarin Inc.

Overview

Rarely does an organization have the luxury of building mobile apps for a single mobile platform. The fact is, the smartphone and tablet space is dominated by three big platforms: iOS, Android and Windows. As such, in order to reach users, apps must be designed and built for all three of them. Traditionally this means using each platform's provided indigenous technology and SDK, i.e. Objective-C for iOS, Java for Android and .NET for Windows.

Most cross-platform mobile toolkits fall short in this space because they provide a lowest-common-denominator experience and prevent developers going "to the metal" on any given platform. With Xamarin, however, this limitation does not exist. Not only do you get a single, modern language (C#) and framework (.NET) across all three platforms, but you also get a native experience on each, with code running as a native peer with direct access to the underlying SDK and device metal, including platform-specific UI and device capabilities.

By choosing Xamarin and keeping a few things in mind when you design and develop your mobile applications, you can realize tremendous code sharing across mobile platforms, reduce your time to market, leverage existing talent, meet customer demand for mobile access, and reduce cross-platform complexity.

This chapter is part of a two-part series that provides that introduces cross-platform concepts and practices using Xamarin. In this chapter we're going to:

- Examine what it means to build cross-platform applications.
- Introduce cross-platform architectural concepts.
- Learn how to setup a file-linked cross-platform solution.
- Introduce other code sharing options.
- Handle platform divergence.

Later on, in the Advanced Cross-Platform Patterns chapter, we'll take a deeper examination of alternative approaches and challenges.

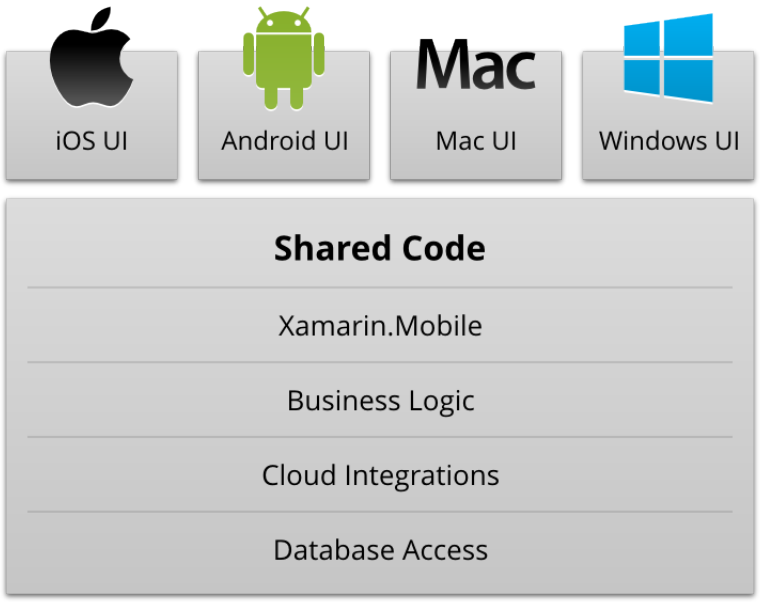
The approach used in both of these chapters is generally applicable to both productivity apps and game apps, however the focus is on productivity and utility (non-game applications).

Cross-Platform Development with Xamarin

The phrase "write-once, run everywhere" is often used to extol the virtues of a single codebase that runs unmodified on multiple platforms. While it has the benefit of code re-use, that approach often leads to applications that have a lowest-common-denominator feature-set and a generic-looking user interface that does not fit nicely into any of the target platforms.

Xamarin is not just a "write-once, run everywhere" platform, because one of its strengths is the ability to implement native user interfaces specifically for each platform. However, with thoughtful design it's still possible to share most of the

non-user interface code and get the best of both worlds: write your data storage and business logic code once, and present native UIs on each platform:



This document discusses a general architectural approach to achieve this goal.

Platform + IDE Availability

Xamarin development can be done in either Xamarin Studio or Visual Studio. The IDE you choose will be determined by the platforms you wish to target.

Because Windows Phone apps can only be developed on Windows in Visual Studio, you must use Visual Studio in order to develop for all three platforms in one Solution. However, it is possible to start writing an application in one IDE, and continue to work for it in another: for example, you could choose to write for Xamarin.iOS in Xamarin Studio to make use of Interface Builder and the Simulator, and later move the solution to Visual Studio to work on the Windows application.

The development requirements for each platform are discussed in more detail below.

Platform	Mac OSX	Windows	
	Xamarin Studio	Xamarin Studio	Visual Studio
iOS	Y		Y
Android	Y	Y	Y
Windows Phone			Y

iOS

Developing iOS applications requires a Mac computer, running Mac OS X. You can also use Visual Studio to write and deploy iOS applications with Xamarin's iOS Visual Studio plugin, however a Mac is still needed for build and licensing purposes.

Apple's Xcode IDE must be installed to provide the compiler and simulator for testing. To test on a real device and submit applications for distribution you must join Apple's Developer Program ($99 USD per year). Each time you submit or update an application it must be reviewed and approved by Apple before it is made available for customers to download.

Code is written with the Xamarin Studio or Visual Studio IDE and screen layouts can be edited with Apple's Interface Builder or built programmatically. Refer to the Xamarin.iOS Installation Guide for detailed instructions.

Java

Android application development requires the Java and Android SDKs to be installed. These provide the compiler, emulator and other tools required for building, deployment and testing. Java, Google's Android SDK and Xamarin's tools can all be installed and run on the following configurations:

- Mac OS X with the Xamarin Studio IDE
- Windows 7 or 8 with the Xamarin Studio IDE
- Windows 7 or 8 with Visual Studio 2010 or Visual Studio 2012

Xamarin provides a unified installer that will configure your system with the pre-requisite Java, Android and Xamarin tools (including a visual designer for screen layouts). Refer to the Xamarin.Android Installation Guide for detailed instructions.

You can build and test applications on a real device without any license from Google, however to distribute your application through a store (such as Google Play, Amazon or Barnes & Noble) a registration fee may be payable to the operator. Google Play will publish your app instantly, while the other stores have an approval process similar to Apple's.

Windows Phone

Windows Phone apps are built with Microsoft's Visual Studio 2010 or 2012 toolset. They do not use Xamarin directly, however C# code can be shared with across Windows Phone, iOS and Android using Xamarin's tools. Visit Microsoft's App Hub to learn about the tools required for Windows Phone development.

General Cross-Platform Considerations

In order to create high quality, native applications and ensure maximum code re-use, there are a few important considerations to keep in mind when developing cross-platform mobile apps.

Users Expect a Native User Experience (UX)

A one-size-fits-all approach doesn't translate well to mobile development. Each platform has specific UX idioms and metaphors that users have come to expect in apps on their respective platforms. Even basic things like navigation vary widely. For example, iOS commonly uses the Navigation Controller, which contains a software back button to manage upward/backward navigation through hierarchical screens. However, Android has a Back button either built into the hardware (pre-Ice Cream Sandwich), or a soft button that is on the screen at all times. Each platform has published design guidelines intended to reinforce these idioms, and there's a strong correlation between top rated apps and native UX. As such, it's important to take platform differences into consideration when designing and building mobile apps.

Devices and their Capabilities Vary Widely

Devices can vary widely, even within a given platform. For instance, there are thousands of different Android devices out there, each with different hardware and capabilities, as well as subtle API differences. Additionally, the form factor can also vary widely between devices. A tablet form factor is very different from a phone form factor, and so should the UX created for them. Tablets can contain much more information and functionality on a single screen than a phone typically does. As such, it's important to consider the impact that form factors and device capabilities have on the design and development of mobile apps.

Architectural Planning and Design is Important

Apps built on Xamarin can share considerable amounts of code across platforms, which can lower cost of development and shorten time to market dramatically. However, in order to achieve maximum code sharing, it's important to take into consideration the architecture of an application before too much has been invested in the developmental cycle, because it's easier to ensure that code is written to maximize reuse than it is to go back and refactor existing code into a more reusable form.

Creating UI

A key benefit of using Xamarin is that the application user interface uses native controls on each platform and is therefore indistinguishable from an application written in Objective-C or Java (for iOS and Android respectively).

When building screens in your app, you can either lay out the controls in code or create complete screens using the design tools available for each platform.

Programmatically Creating Controls

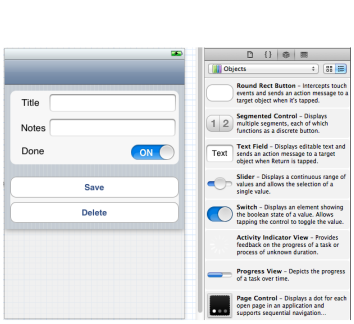
Each platform allows user interface controls to be added to a screen using code. This can be very time-consuming as it can be difficult to visualize the finished design when hard-coding pixel coordinates for control positions and sizes.

Programmatically creating controls does have benefits though, particularly on iOS for building views that resize or render differently across the iPhone and iPad screen sizes.

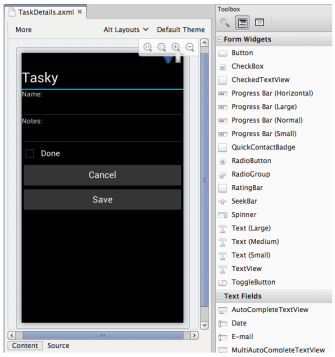
Each platform has a different method for visually laying out screens:

- iOS – Xamarin Studio and Visual Studio integrate with Apple's Xcode Interface Builder which allows you to create individual screen layouts or storyboards that describe multiple screens. This results in *.xib* or *.storyboard* files that are included in your project.
- Android – Xamarin provides an Android drag-and-drop UI designer for both Xamarin Studio and Visual Studio. Android screen layouts are saved as *.axml* files when using Xamarin tools.
- Windows Phone – Microsoft provides a drag-and-drop UI designer in Visual Studio and Blend. The screen layouts are stored as .XAML files.

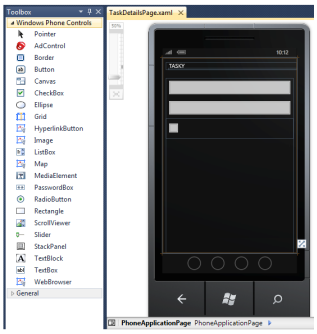
These screenshots show the visual screen designers available on each platform:



iOS



Android



Windows Phone

In all cases the elements that you create visually can be referenced in your code.

User Interface Considerations

A key benefit of using Xamarin to built cross platform applications is that they can take advantage of native UI toolkits to present a familiar interface to the user. The UI will also perform as fast as any other native application.

Some UI metaphors work across multiple platforms (for example, all three platforms use a similar scrolling-list control) but in order for your application to 'feel' right the UI should take advantage of platform-specific user interface elements when appropriate. Examples of platform-specific UI metaphors include:

- iOS – hierarchical navigation with soft back button, tabs on the bottom of the screen.
- Android – hardware/system-software back button, action menu, tabs on the top of the screen.
- Windows Phone – hardware back button, panorama layout control, live tiles.

It is recommended that you read the design guidelines relevant to the platforms you are targeting:

- iOS – Apple's Human Interface Guidelines
- Android – Google's User Interface Guidelines

- **Windows Phone** – User Experience Design Guidelines for Windows Phone

Let's examine cross-platform architectural considerations.

Cross-Platform Architecture

The code sharing capacity of applications can be greatly increased with good architecture practices. The objective is to structure code in such a way that the maximum amount of code will run on all platforms, while allowing platform-specific code to remain tied to the platform. In this section, we offer some guidelines for architecting a cross-platform application, and introduce Application Layers as a way of maximizing the amount of reusable code in an application.

General Considerations

A key tenant of building cross-platform apps is to create an architecture that lends itself to a maximization of code sharing across platforms. Adhering to the following principles helps create well-architected applications:

- **Encapsulation:** Ensure that classes and architectural layers only expose a minimal API, and hide the implementation details. At a class level, this means that objects behave as 'black boxes', where the consuming code does not know how the objects accomplish their tasks.
- **Separation of Responsibility:** Similarly, ensure that each component, whether a class or an architectural layer, has a clear and well-defined purpose. This means that the UI code should only be responsible for displaying screens and accepting user-input, and never interacting with the database directly. Similarly, the data-access code should only read and write to the database, but never interact directly with buttons or labels.
- **Core Code Sharing:** Share as much code as possible in the Core layers, but keep the UI separate.

With these principles in mind, let's see how we can use distinct architectural layers to help separate the responsibilities of the application, and create Core code that we can share.

Application Layers

Separating code into layers make applications easier to understand, test, and maintain. It also makes the code easier to share amongst applications on different platforms. The code in each layer should be kept physically as well as logically separate; this can be achieved by placing code in separate directories, and using difference namespaces.

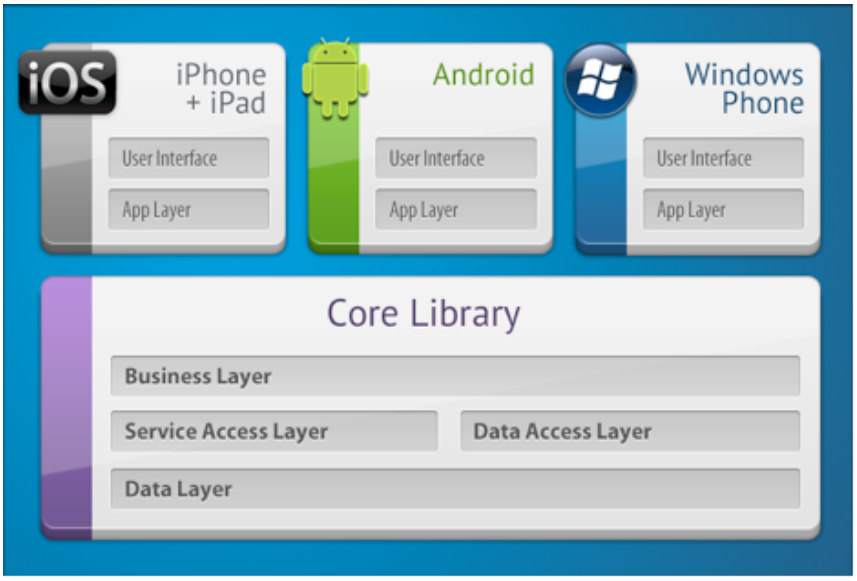
An application might contain some of the following layers:

- **Data Layer (DL)** – Used for non-volatile data persistence. This could take the form of a SQLite database, XML files, or any other suitable mechanism.

- **Data Access Layer (DAL)** – Wrapper around the Data Layer that provides Create, Read, Update, Delete (CRUD) access to the data without exposing implementation details to the caller. For example, the DAL may contain SQL statements to query or update the data, but the referencing code would not need to know this.
- **Business Layer (BL)** – (sometimes called the Business Logic Layer or BLL) contains business entity definitions (the Model) and business logic.
- **Service Access Layer (SAL)** – Used to access services in the cloud. Encapsulates the networking behavior and provides a simple API to be consumed by the Application and UI layers.
- **Application Layer (AL)** – Code that's platform specific or specific to the application, and not generally reusable. A good test of whether to place code in the Application Layer versus the UI Layer is to determine whether the class has any actual display controls (UI Layer), or whether it could be shared between multiple screens or devices (Application Layer).
- **User Interface Layer (UI)** – The user-facing layer. Contains screens, widgets and the controllers that manage them.

An application may not necessarily contain all layers – for example, the Service Access Layer would not exist in an application that does not access network resources. A very simple application might merge the Data Layer and Data Access Layer if the operations are extremely basic.

The following diagram illustrates how the different layers fit together in a cross-platform application. The Core Library contains the layers of code that can be shared between platforms. Similarly, the individual applications contain AL and UI Layer code that cannot be shared:



Now that we understand the basic principles behind cross-platform architecture, let's learn how the code sharing mechanism works.

Code Sharing Options

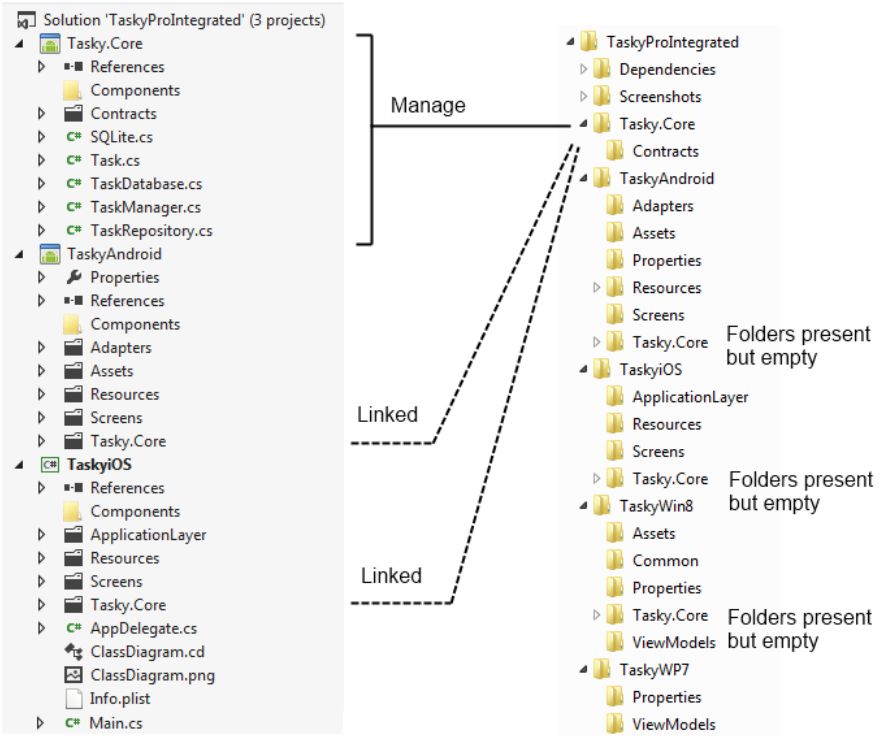
Once a sound architecture is in place and the core logic has been abstracted away from the view, the shared code needs to be made available for the various platform-specific projects to use. Two approaches are covered here: the first option is to link a file or project containing shareable code into the individual platform-specific projects. The second is to create Portable Class Libraries: special projects containing code that can be used by different platforms.

Currently, file linking is generally the recommended approach, because it's easier to set up and because it usually provides the highest value for the lowest effort. However, there are other valid approaches as well. In this chapter we're going to walk through creating a linked cross-platform solution that can be opened in both Xamarin Studio (in Windows or Mac), as well as Visual Studio.

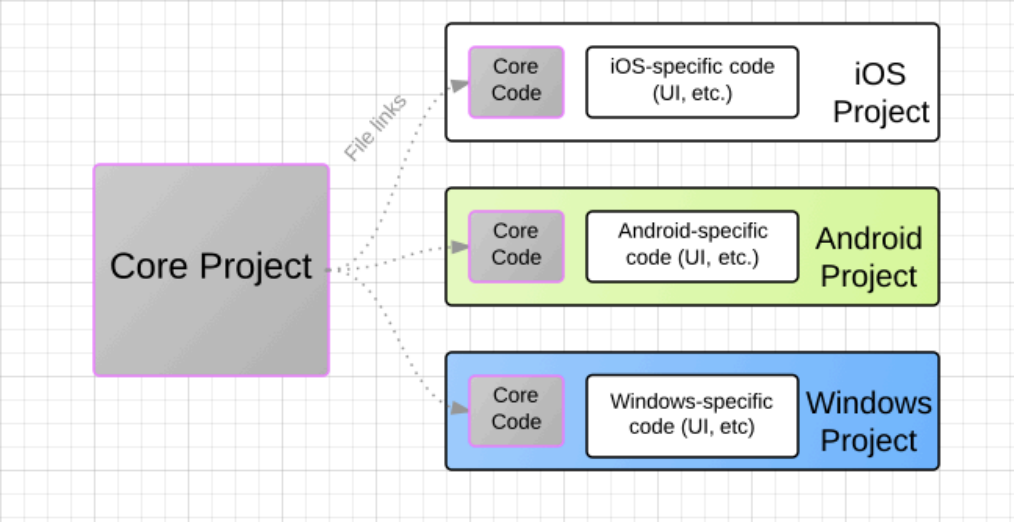
We will examine other methodologies in more detail later, in the Advanced Cross-Platform Patterns chapter.

Sharing Code with File Linking

File linking works by creating symbolic links to the shared files. The easiest approach is to create a core C# library project in your solution, and then link those files into the various platform projects as shown in the following screenshot:



This allows you to manage shared files all in one place, and then consume them in various platforms:



File linking has the following advantages:

- It is simple to understand and easy to implement.
- It allows use of conditional compilation to handle platform divergence. Conditional Compilation is covered later in the chapter.

We will cover how to link files in the Setting Up a Cross-Platform Solution walkthrough in the next section. For a thorough description of the advantages and disadvantages of various file linking options, as well as recommendations of what to link based on the type of application and the IDE, see the Sharing Code Options guide in the Xamarin Documentation Portal.

Using PCL (Portable Class Libraries)

Historically, a .NET project file (and the resulting assembly) has been targeted to a specific framework version. This prevents the project or the assembly being shared by different frameworks.

A Portable Class Library (PCL) is a special type of project that can be used across disparate CLI platforms such as Xamarin.iOS and Xamarin.Android. A PCL contains code that is the "lowest common denominator" of all the different platforms that are targeted. The library can only utilize a subset of the complete .NET framework, limited by the platforms being targeted.

PCL is an elegant solution from an architectural standpoint. In practice, however, file linking is the better option for several reasons: first, Xamarin's support for Portable Class Libraries is still under development. Second, PCL is a lot of work to implement, and can be complicated to use. File linking is officially recommended by Xamarin for sharing code between applications on different platforms because it is simpler to use and more reliable than PCL.

Now that we understand how to create shareable code and add it to the applications running on different platforms, let's cover how to recognize and handle the parts of the application code that can't be shared.

Platform Divergence

Platforms and the families of devices that run on them have different features and capabilities. Handling divergence is not only a way of ensuring an application has the resources it needs to run, but also an opportunity to make use of the rich features that the different platforms and devices have to offer, while still degrading gracefully when specific features are not available.

Divergence isn't just a cross-platform problem; devices on the same platform have different capabilities. Screen-size, keyboard availability, and NFC availability are all examples of within-platform divergence. Just as in the case of cross-platform divergence, these differences require an application to check for certain capabilities, and behave differently based on their presence or absence.

Within-platform divergence renders the task of making applications that run on all devices nearly impossible, and often unnecessary. A more practical approach is to target devices that the application will run well on, and provide it with the tools to degrade gracefully when opened on a device it does not support.

Fundamental Cross-Platform Elements

There are some characteristics of mobile applications that are universal. These are higher-level concepts that are generally true of all devices and can therefore form the basis of your application's design:

- Feature selection via tabs or menus
- Lists of data and scrolling
- Single views of data
- Editing single views of data
- Navigating back

When designing your high-level screen flow you can base a common user experience on these concepts.

Platform-Specific Attributes

In addition to the basic elements that exist on all platforms, you will need to address key platform differences in your design. You may need to consider (and write code specifically to handle) these differences:

- Screen sizes: iOS and Windows Phone have standardized screen sizes that are relatively simple to target. Android devices have a plethora of screen dimensions, which require more effort to support in an application.
- Navigation metaphors: Navigation controls differ across platforms, and applications need to be sensitive to these to feel intuitive on the different platforms. For example, an application that shows a 'back' button on iOS shouldn't have the button on Android, which already provides a hardware 'back' button.

- Keyboards: Some Android and Windows devices have physical keyboards, while others only have a software keyboard. In the case of tablet-laptop devices like the Surface, a hardware keyboard may or may not be connected to the device when an application is launched. Code that detects when a soft-keyboard is obscuring part of the screen needs to sensitive to these differences.
- Touch and gestures: Operating system support for gesture recognition varies, especially in older versions of each operating system. Earlier versions of Android have very limited support for touch operations, meaning that supporting older devices may require separate code.
- Push notifications – There are different capabilities/implementations on each platform (eg. Live Tiles in Windows Phone).

Device-Specific Features

Determine what the minimum features required for the application must be; or when decide what additional features to take advantage of on each platform. Code will be required to detect features and disable functionality or offer alternatives (eg. an alternative to geo-location could be to let the user type a location or choose from a map):

- Camera: Camera functionality differs among devices. Some devices don't have a camera, others have both front- and rear-facing cameras. Some cameras are capable of video recording.
- Geo-location & maps: Support for GPS or Wi-Fi location is not present on all devices. Apps also need to cater for the varying levels of accuracy that's supported.
- Accelerometer, gyroscope and compass: These features are often found in only a selection of devices on each platform, so apps almost always need to provide a fallback when the hardware isn't supported.
- Twitter and Facebook: Only 'built-in' on iOS5 and iOS6 respectively. On earlier versions and other platforms you will need to provide your own authentication functions and interface directly with each service's API.
- Near Field Communications (NFC) – Available only on some Android phones at time of writing.

Dealing with Platform Divergence

There are two different approaches to supporting multiple platforms from the same code-base, each with its own set of benefits and disadvantages:

- Conditional Compilation – Divergent compilation of specific code, depending on the platform target that it's being built for. Conditional compilation is easy and has the advantage of not requiring a lot of architectural maneuvering to handle simple divergence.

- **Architectural Abstraction** – Divergent architectural implementations using abstraction tools such as interfaces, dependencies, and/or platform-specific providers.

Conditional Compilation

An effective and simple way to handle platform divergence is with conditional compilation. Conditional compilation uses pre-defined symbols to delineate code that will only compile on the specified platform. For example, the following code block delineates code that will only run on Android:

```
#if __ANDROID__
      // Android-specific code
#endif
```

The following pre-defined symbols are available on the different platforms:

- **iOS:** The `__IOS__` symbol can be used to delineate iOS-specific code.
- **Android:** The `__ANDROID__` symbol can be used to delineate Android-specific code. Additionally, each API version also defines a new compiler directive. For example, `__ANDROID_11__` will target Honeycomb 3.0 or newer.
- **Windows:** Windows Phone 7 defines two symbols – `WINDOWS_PHONE` and `SILVERLIGHT` – that can be used to target code to the platform.
- **Mobile:** Xamarin also provides the `__MOBILE__` to delineate code that will run on both iOS and Android.

Additionally, new compiler directives can be created in **Project > Settings > Build**.

A simple case-study example of conditional compilation is setting the file location for the SQLite database file. The three platforms have slightly different requirements for specifying the file location:

- **iOS** – Apple prefers non-user data to be placed in a specific location (the Library directory), but there is no system constant for this directory. Platform-specific code is required to build the correct path.
- **Android** – The system path returned by `Environment.SpecialFolder.Personal` is an acceptable location to store the database file.
- **Windows Phone** – The isolated storage mechanism does not allow a full path to be specified, just a relative path and filename.

The following code uses conditional compilation to ensure the **DatabaseFilePath** is correct for each platform:

```
public static string DatabaseFilePath {
    get {
        var filename = "MwcDB.db3";
#if SILVERLIGHT
        var path = filename;
```

```
#else

#if __ANDROID__
    string libraryPath = Environment.GetFolderPath(
        Environment.SpecialFolder.Personal); ;
#else
    // we need to put in /Library/ on iOS5.1 to meet Apple's iCloud terms
    // (they don't want non-user-generated data in Documents)
    string documentsPath = Environment.GetFolderPath (
        Environment.SpecialFolder.Personal); // Documents folder
    string libraryPath = Path.Combine (documentsPath, "..", "Library");
#endif
    var path = Path.Combine (libraryPath, filename)
#endif
    return path;
}
```

Architectural Abstraction

There are a number of architectural tools available to help address platform divergence.

Class Abstraction

Class abstraction involves using either interfaces or base classes defined in the shared code and implemented or extended in platform-specific projects. Writing and extending shared code with class abstractions is particularly suited to Portable Class Libraries, because they have a limited subset of the framework available to them and cannot contain compiler directives to support platform-specific code branches.

Interfaces

Using interfaces allows you to implement platform-specific classes that can still be passed into your shared libraries to take advantage of common code.

The interface is defined in the shared code, and passed into the shared library as a parameter or property.

The platform-specific applications can then implement the interface and still take advantage of shared code to ‘process’ it.

Inheritance

The shared code could implement abstract or virtual classes that could be extended in one or more platform-specific projects. This is similar to using interfaces but with some behavior already implemented. There are different viewpoints on whether interfaces or inheritance are a better design choice: in particular because C# only allows single inheritance it can dictate the way your APIs can be designed going forward. Use inheritance with caution.

The advantages and disadvantages of interfaces apply equally to inheritance, with the additional advantage that the base class can contain some implementation code (perhaps an entire platform agnostic implementation that can be optionally extended).

Cross-Platform Libraries

There are several cross-platform libraries available to help increase the amount of reusable code in an application, and decrease the hassle of handling platform divergence. Some, like MonoCross and MvvmCross, specifically target the issue of code reuse in the UI, providing another layer of abstraction on top of the native UI. Others, like Xamarin.Mobile, provide support for specific issues and application needs. Used alongside well-thought-out architecture, these libraries can help make applications easier to write and maintain.

Xamarin provides the following libraries for cross-platform development:

- Xamarin.Mobile: This library augments the .NET Base Class Libraries with a set of APIs that can be used to access device-specific features in a cross-platform way, and includes support for working with media, location, and contacts. This is available through the Xamarin Components Store.
- Xamarin.Social: A Xamarin library that posts statuses, links, images, and other media to social networks using a simple, cross-platform API, Xamarin.Social provides support for several popular social networks, including Twitter and Facebook. It is available through the Components Store as well.
- Monotouch.Dialog: `Monotouch.Dialog` provides an abstract API that for the "scrolling list" user interface. See the Introduction to MonoTouch.Dialog for more information on building screens using `MonoTouch.Dialog`.

Additionally, a number of third-party libraries are available for cross-platform development, including:

- MonoCross: An open-source adaptation of the MVC pattern for mobile, MonoCross abstracts the view interface, creating shared application code to handle navigation, data, access, and more.
- MvvmCross: A branch off the MonoCross project that replaces MVC with MVVM (Model, View, View Model).
- Vernacular: A localization tool developed by the engineers at Rdio, Vernacular provides a unified location system for Xamarin.iOS, Xamarin.Android, and Windows Phone.
- MonoGame: An open-source version of the Microsoft XNA framework, MonoGame focuses on cross-platform game development across an impressively long list of platforms - including iOS, Android, Mac, PlayStation, and more.
- CrossGraphics: CrossGraphics provides a more cross-platform approach to drawing code.

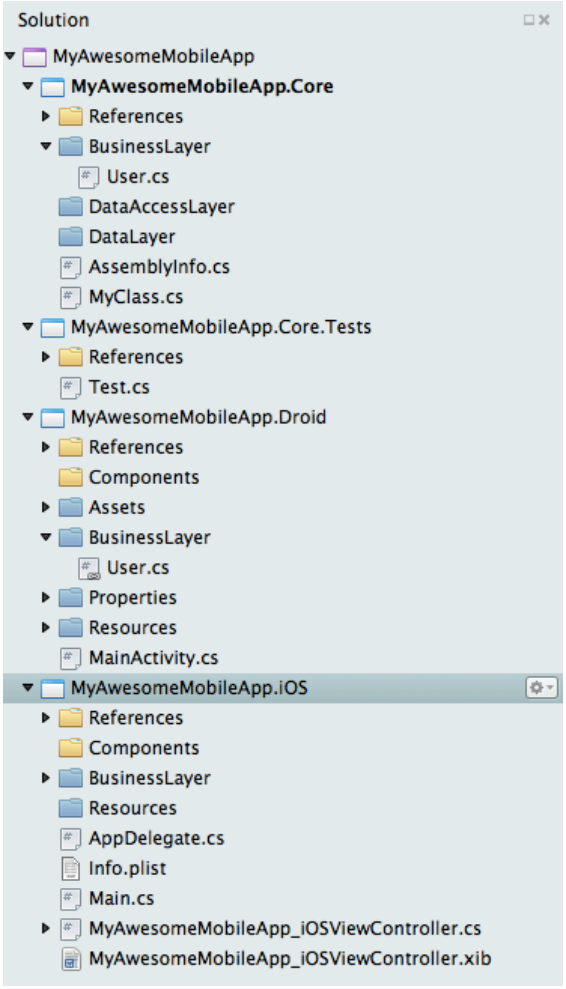
We've covered the basic architectural best practices and code-sharing options required for successful cross-platform applications, and learned to handle platform divergence gracefully. For a more detailed examination of these concepts, refer to the Advanced Cross Platform Patterns chapter, as well as the Building

Cross-Platform Applications guide in the Xamarin Documentation Portal. In the next section, we will walk through setting up a Xamarin cross-platform solution.

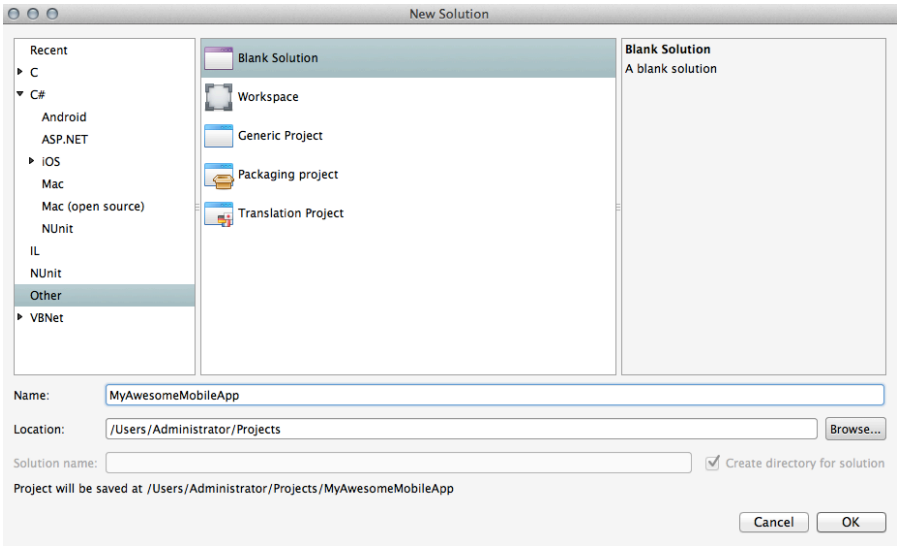
Setting up a File-Linked Cross-Platform Solution Walkthrough

In this walkthrough, we are going to create a simple but well-architected cross-platform application that maximizes the shared code base, and uses file linking to share code between iOS, Android, and Windows. The first half of this walkthrough will focus on creating a cross-platform application that shares code between iOS and Android in Xamarin Studio. Later in this chapter, we will cover how to set up a cross-platform solution in Visual Studio that supports code sharing between the iOS, Android, and Windows Phone platforms.

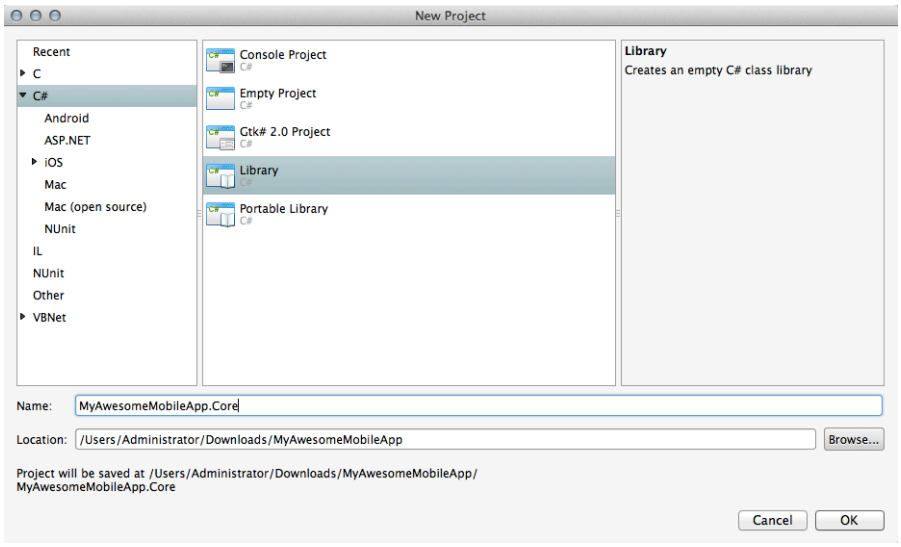
We will create a Solution structure that resembles the following:



1. First, open Xamarin Studio and create a new blank solution by navigating to **Other > Blank Solution**. Name this solution *MyAwesomeMobileApp*:



2. The next step is to add a project to house all of our shared code. Xamarin Studio offers a project type called a *Library Project* for both iOS and Android; however, these projects are platform-specific, and cannot be used to share code across platforms. Instead, we will create a simple C# Library project. We'll call it *MyAwesomeMobileApp.Core*:



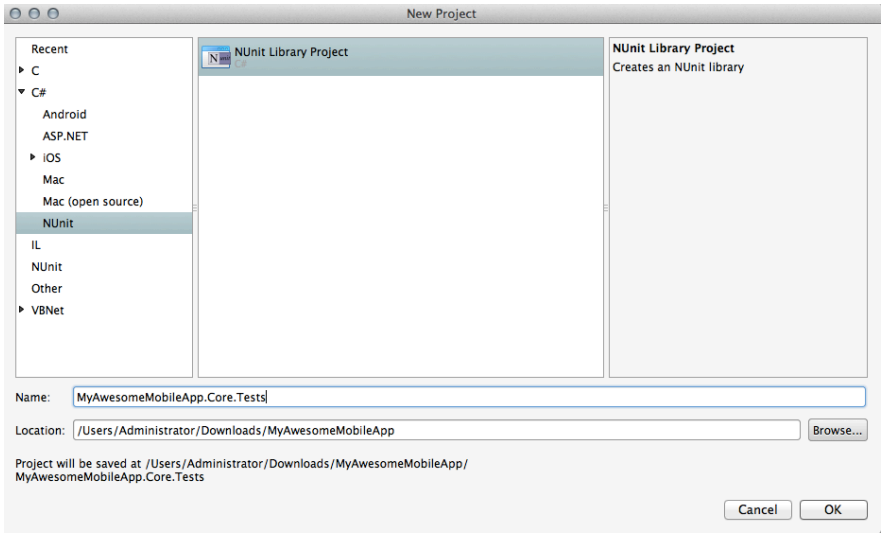
3. Let's add directories to the Core project to house the different Application Layers of our application. Add new folders to the project for our **BusinessLayer**, **DataAccessLater**, and **DataLayer.**

   In the **BusinessLayer**, let's create a sample entity for *User*. Add a new class called User, and add a property called *Name* to the User:

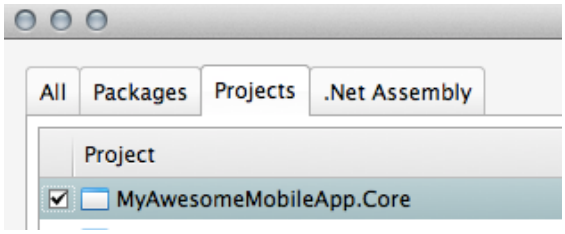
```
namespace MyAwesomeMobileApp.Core
{
    public class User
    {
        public string Name { get; set; }

        public User ()
        {
```

```
            }
        }
    }
```

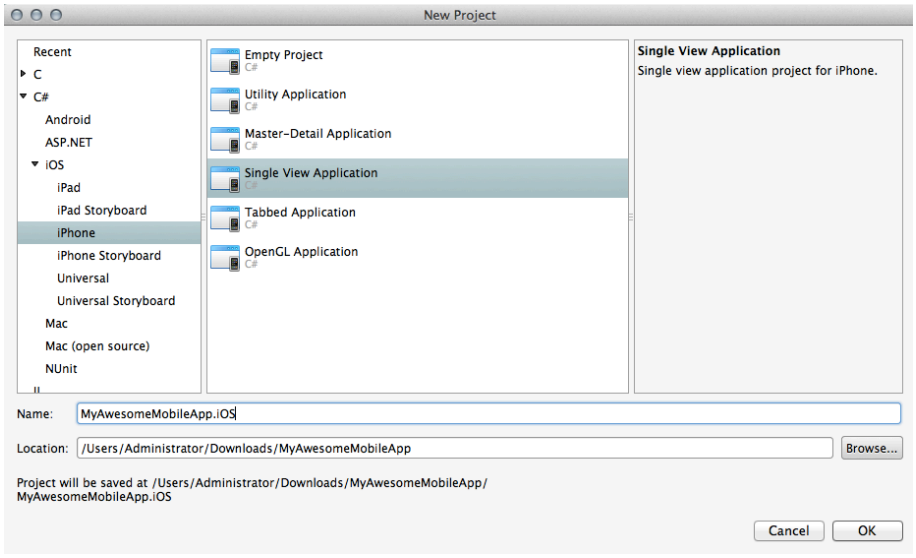
4. It's time to add some unit tests to our application. In Xamarin Studio, we need to use **NUnit** (in Visual Studio, either **NUnit** or **MSTest** will work). Create a new **NUnit Library Project** called *MyAwesomeMobileApp.Core.Tests*:



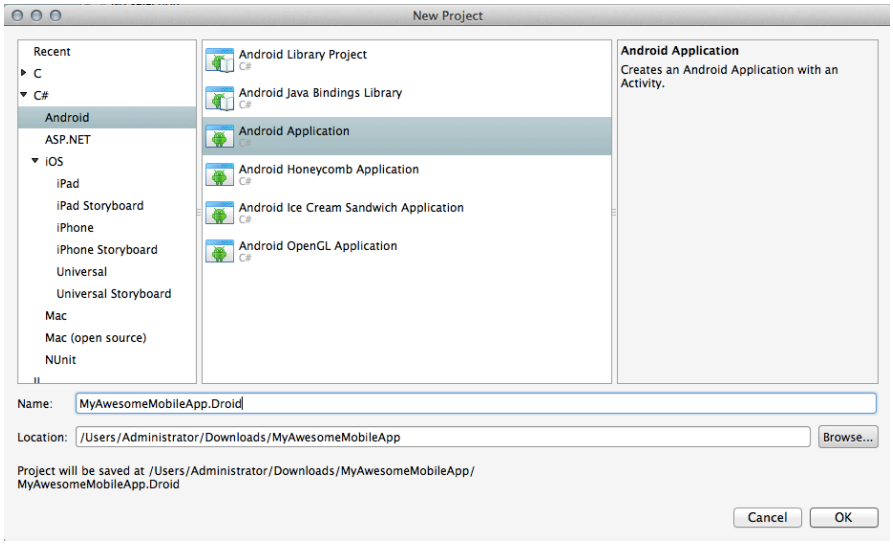
Finally, add a reference to the *MyAwesomeMobileApp.Core* project in the *MyAwesomeMobileApp.Core.Tests* project:



5. We've created a shared code library, and we've added tests to our application. The next step is to create the projects that house the mobile applications. To create the iOS project, add a new **iPhone > Single View Application** called *MyAwesomeMobileApp.iOS*:



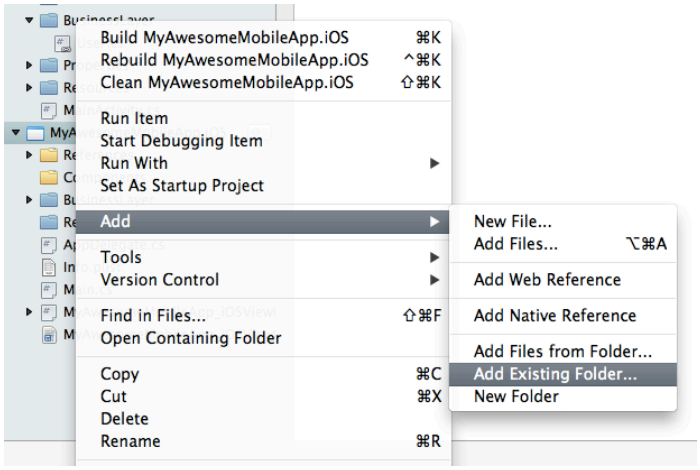
To add a new Android project, select **Android > Android Application** and name it *MyAwesomeMobileApp.Droid*:



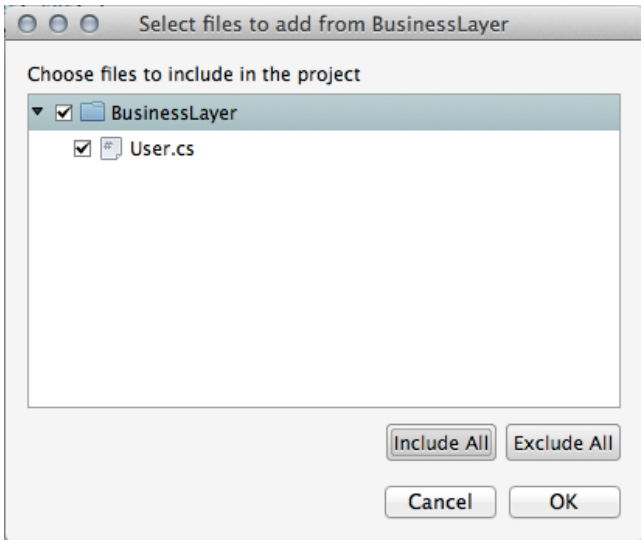
Note: Notice that we name the Android project *MyAwesomeMobileApp.Droid*, rather than using *MyAwesomeMobileApp.Android*. **.Android** in the project name results in namespace collisions, and should never be used.

6. Finally, we need to get the shared code from the Core project into the iOS and Droid projects. The iOS and Android project types are based on different .NET profiles, so adding a reference to the Core project will not work. Instead, we will use file linking to create symbolic links to the shared code.

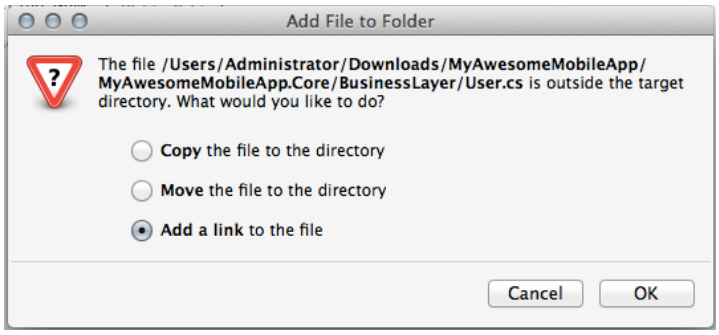
   In the iOS project, select **Add > Add Existing Folder**:



Navigate to the *MyAwesomeMobileApplication.Core* project, and select the **BusinessLayer** directory. Select the option to include all files in the directory:



Finally, select the option to **Add a link to the file**:



Repeat the same process to link files in the Android project for every Core directory that needs to be linked.

An important thing to keep in mind during this process is that file linking creates actual symbolic links. This means that editing a linked file from the iOS or Android project will edit the file in the Core project as well. To demonstrate

this, open the `User.cs` file in the **BusinessLayer** directory from the iOS project, and all a new property called `ID`:

```
namespace MyAwesomeMobileApp.Core
{
    public class User
    {
        public int ID { get; set; }
        public string Name { get; set; }

        public User ()
        {
        }
    }
}
```

Opening the `User.cs` file in the Core project will reveal that the change has been picked up.

In this walkthrough, we've used Xamarin Studio to create a cross-platform application that runs on iOS and Android, and shares a testable Core. In order to add Windows Phone support to our application, we need to move over to Visual Studio.

The next section will cover how to work with iOS and Android projects in Visual Studio. At the end of the section, we will be able to create a cross-platform mobile application that targets iOS, Android, and Windows 8 Mobile in Visual Studio.

Xamarin Development in Visual Studio

So far, we have covered cross-platform best practices and architectural principles that are applicable to application development in both Xamarin Studio and Visual Studio. However, we have only put these concepts into practice in Xamarin Studio on a Mac.

Fortunately, many of the same guidelines apply to cross-platform application development on Windows, both in Xamarin Studio (for Android) and Visual Studio (for iOS and Android), with just a few key differences. These differences mostly affect iOS, and will be covered more in-depth later in the section.

First, let's explore some of the benefits of Visual Studio, and why we might choose to use it for mobile development.

Visual Studio Highlights

Developing for iOS and Android inside Visual Studio provides a number of benefits:

- Creation of a single cross-platform solution for iOS, Android and Windows applications. Recall that Windows application development is not supported in Xamarin Studio, so we will need to turn to Visual Studio to build Windows apps.
- Using Visual Studio tools (such as **Resharper** and **Team Foundation Server**) for all your cross-platform projects, including source code.

- Using the familiar (for existing Visual Studio developers) code editor, keyboard shortcuts, etc.

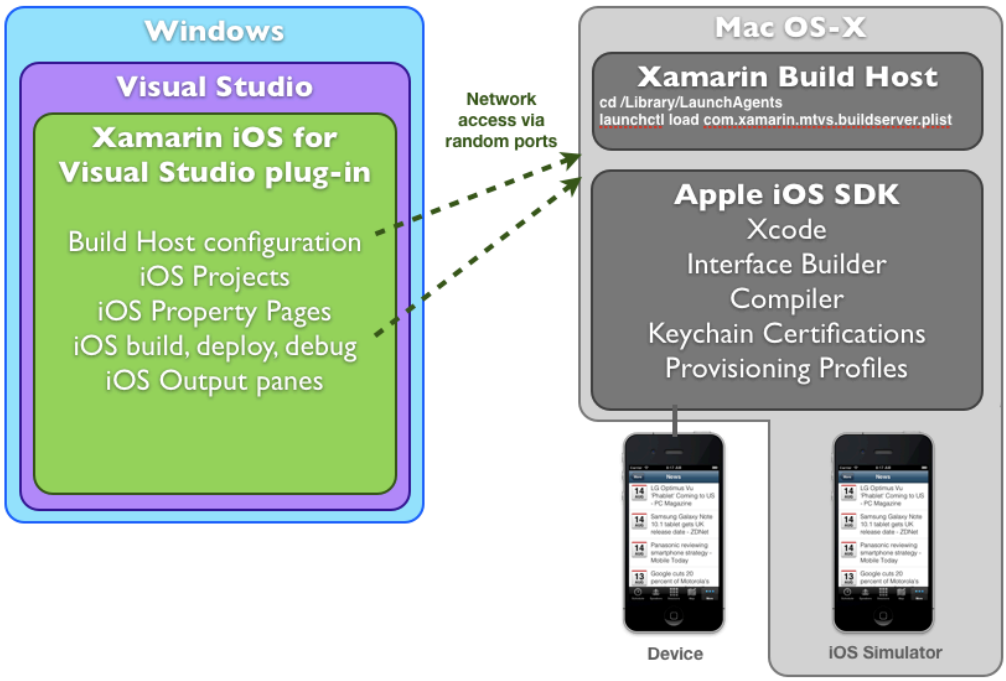
Now that we understand the benefits of mobile development in Visual Studio, let's explore some of the differences and challenges it presents.

Android Development in Windows

Android development in Windows can be done in Visual Studio or in Xamarin Studio. The differences between the two environments are minor: the menus and dialogs differ somewhat, as does the location of certain settings. For example, many of the controls in the **Project Options** dialog in Xamarin Studio are split between the **Project Properties** and the **Tools > Options** dialog in Visual Studio.

iOS Development in Windows

As mentioned earlier, the iOS development experience is somewhat different on Windows than it is on Mac. Xamarin.iOS uses the Xcode build system, and Apple requires a Mac as a part of the license requirement. In order for iOS development on Windows to work, Visual Studio requires a connection to a networked Mac to provide the build and deployment service:



A guide to setting up and troubleshooting the Mac build host is available as part of the Installing Xamarin.iOS on Windows guide in the Xamarin Developer Center.

XS/VS Differences

Because of the way Xamarin.iOS for Visual Studio is configured, several iOS-specific features available in Xamarin Studio may not be as readily accessible from

Visual Studio. Key differences between Xamarin.iOS development in Xamarin Studio and Visual Studio include:

- **Designer** - At the time of writing, there is no official UI designer available for iOS development in Visual Studio. Various options for designing the UI are covered in the next section.
- **iOS Simulator** – You can deploy apps to the Simulator running on the Mac build host, but in order to see it running, the Mac machine must either be physically available, or available via VNC.
- **Deployment to Device** - In order to deploy to a device, the device must be plugged into the Mac build server.
- **Feature Support** - Several Xamarin.iOS features, including TestFlight support, application profiling using Instruments, and the advanced .plist editor, are not available in Xamarin.iOS for Visual Studio.

Running Visual Studio in a Windows virtual machine on a Mac may help to integrate some of these tools back into the workflow. For example, this setup would allow for easy switching between Visual Studio and the iOS Simulator. VMWare and Parallels are examples of tools that enable this kind of setup.

> **Note:** Xamarin Studio on Windows does not support iOS development. To create Xamarin.iOS applications on Windows, you must use Visual Studio.

User Interface Options

There are two options available for creating the UI in Xamarin.iOS for Visual Studio. The first and simplest is to build out UIs programmatically. The second involves moving the entire solution over to a Mac, opening the application in Xamarin Studio, and using Interface Builder to generate the necessary XIB files and code.

Keep in mind that any Storyboard or XIB files you add to the application cannot currently be edited in Visual Studio. If you create an application from a Storyboard template (or a template that includes XIB files), you will have to switch to the Mac and open the solution in Xamarin Studio in order to edit them.

For more information on using Xamarin.iOS in Visual Studio, refer to the Introduction to Xamarin.iOS for Visual Studio guide in the Xamarin Developer Center.

Windows 8 Development in Windows

Windows 8 development is only available in Visual Studio. There is currently no support for Windows 8 applications in Xamarin Studio on Mac or Windows. For a more information on Windows 8 development in Visual Studio, refer to the Getting Started with Windows Store Apps Microsoft documentation.

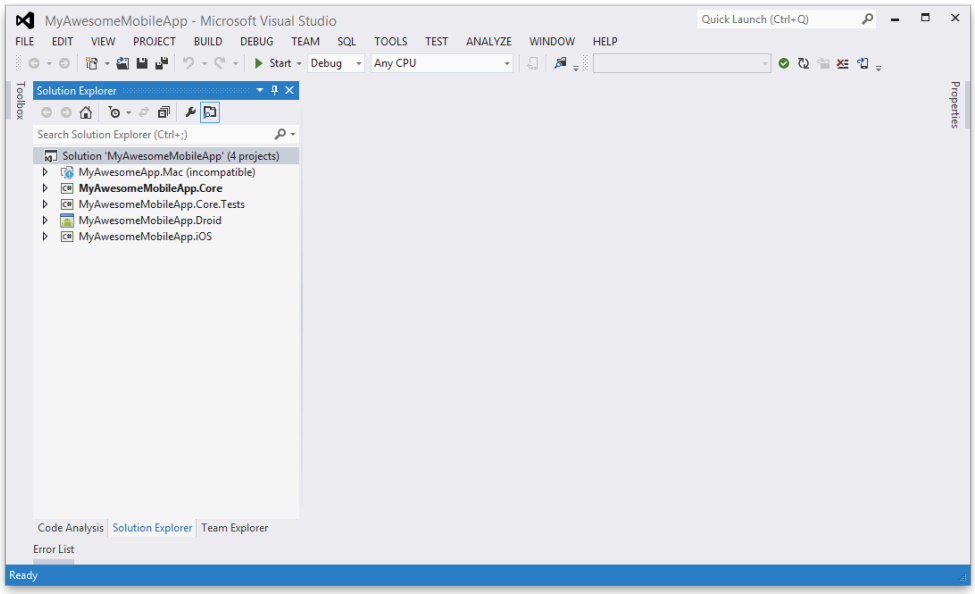
In the next section, we will walk through creating a Windows 8 application that shares code with applications on other platforms.

Visual Studio File-Linking Solution Setup Walkthrough

In this walkthrough, we are going to update the *MyAwesomeMobileApp* application how to share code between iOS, Android and Windows 8 Mobile. To do so, we will first open the Solution in Visual Studio, and add a Windows 8 application.

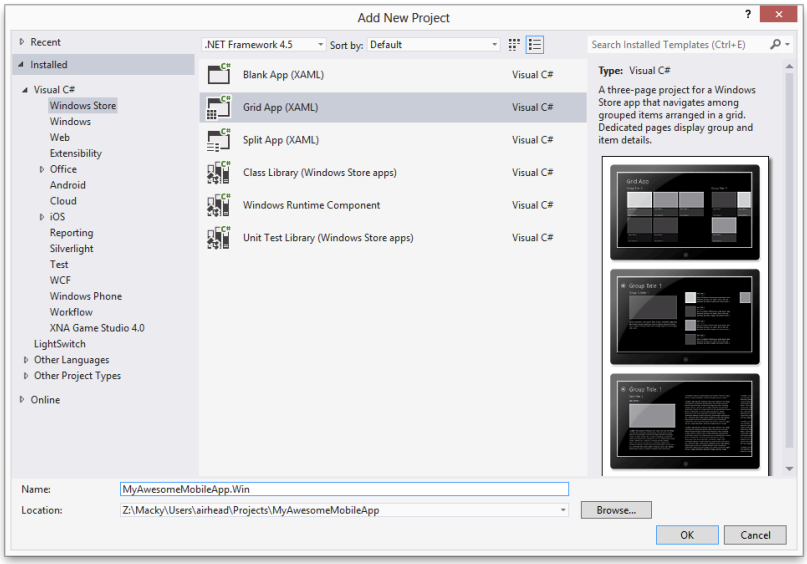
This walkthrough assumes that the Windows 8 SDK is installed.

1. In the previous walkthrough, we created a cross-platform solution named *MyAwesomeMobileApp* that contains our shared code, NUnit tests, iOS, and Android projects. Let's begin by opening the *MyAwesomeMobileApp* solution in Visual Studio:

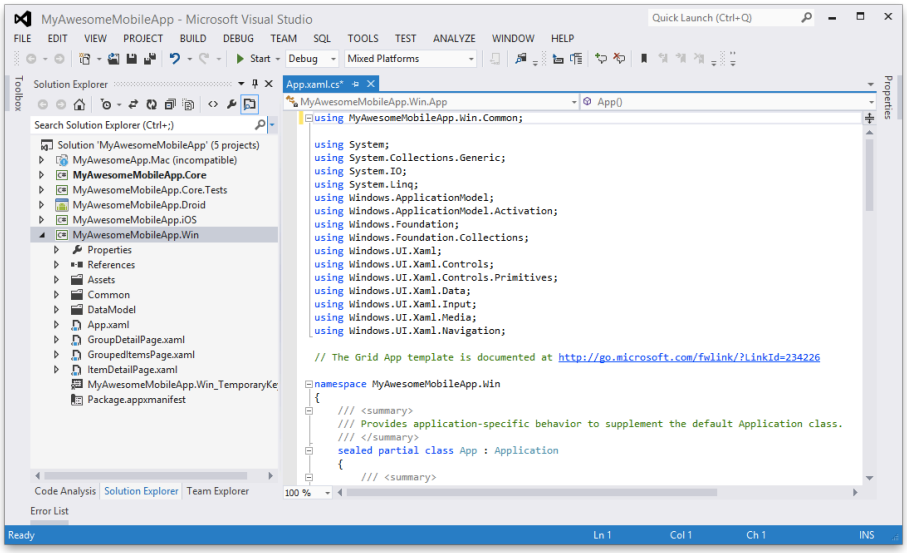


   Notice that Visual Studio is aware of the linked project files, and preserves the symbolic links we created earlier. All that's left is to create the Windows 8 application, and link the shared files.

2. To make the Windows 8 application, create a new Windows Store project. Let's call this one *MyAwesomeApplication.Windows:*



The resulting project should resemble the following:



3. Next, just as we did in Xamarin Studio, add the files by right clicking on the project and choosing **Add > Existing Folders**.

You should now have a solution file that uses a shared code-base across iOS, Android, and Windows 8!

Summary

In this chapter we examined how to use Xamarin to build applications that target multiple mobile platforms. Along the way, we covered cross-platform considerations, offered architectural guidance and patterns, and walked through creating a cross platform solution for iOS, Android, and Windows 8.

We also introduced the role Visual Studio plays in the Xamarin ecosystem and how to use it to develop mobile applications with Xamarin.