Building Cross Platform Applications

Best Practices for Developing Mobile Applications with Xamarin

Xamarin Inc.

BRIEF

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 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 document outlines key guidelines to realizing these advantages for utility and productivity applications. Refer to our separate guidance document for writing cross-platform games with Xamarin.

Sample Code:

Tasky Sample App (github)

MWC 2012 Sample App (github)

Related Articles:

[Case Study: Tasky]

[Case Study: MWC 2012]

[Sharing Code Strategies]

MWC 2012 (blog post)

Related 3rd Party Links:

C# SQLite (Google Code)

3rd party example – Flights Norway (@follesoe)

Books:

Mobile Development with C# by Greg Shackles (O'Reilly)

Professional Cross-Platform Mobile Development in C# by Scott Olson, John Hunter, Ben Horgen, Kenny Goers (Wrox)

Overview

This guide introduces the Xamarin platform and how to architect a cross-platform application to maximize code re-use and deliver a high-quality native experience on all of the main mobile platforms: iOS, Android and Windows Phone.

The approach used in this document is generally applicable to both productivity apps and game apps, however the focus is on productivity and utility (non-game applications). See the [Introduction to MonoGame document] for cross-platform game development guidance.

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.

Here is a summary of the key points for creating Xamarin cross-platform apps:

➔ **Use C#** - Write your apps in C#. Existing code written in C# can be ported to iOS and Android using Xamarin very easily, and obviously used on Windows Phone.

➔ **Utilize the MVC design pattern** - Develop your application's User Interface using the Model/View/Controller pattern. Architect your application using a Model/View/Controller approach or a Model/View/ViewModel approach where there is a clear separation between the "Model" and the rest. Determine which parts of your application will be using native user interface elements of each platform (iOS, Android, Windows Phone and Windows 8/RT) and use this as a guideline to split your application into two components: "Core" and "User-Interface".

➔ **Build native UIs** - Each OS-specific application provides a different user-interface layer (implemented in C# with the assistance of native UI design tools):

1. On iOS use the MonoTouch.UIKit APIs to create native-looking applications, optionally utilizing Apple's Interface Builder.

2. On Android, use Android.Views to create native-looking applications, taking advantage of Xamarin's UI designer

3. On Windows Phone you will be using the XAML/Silverlight presentation layer, using Visual Studio or Blend's UI designer

4. On Windows 8, use the Metro APIs to create a native user experience.

The amount of code re-use will depend largely on how much code is kept in the shared core and how much code is user-interface specific. The core code is anything that does not interact directly with the user, but instead provides services for parts of the application that will collect and display this information.

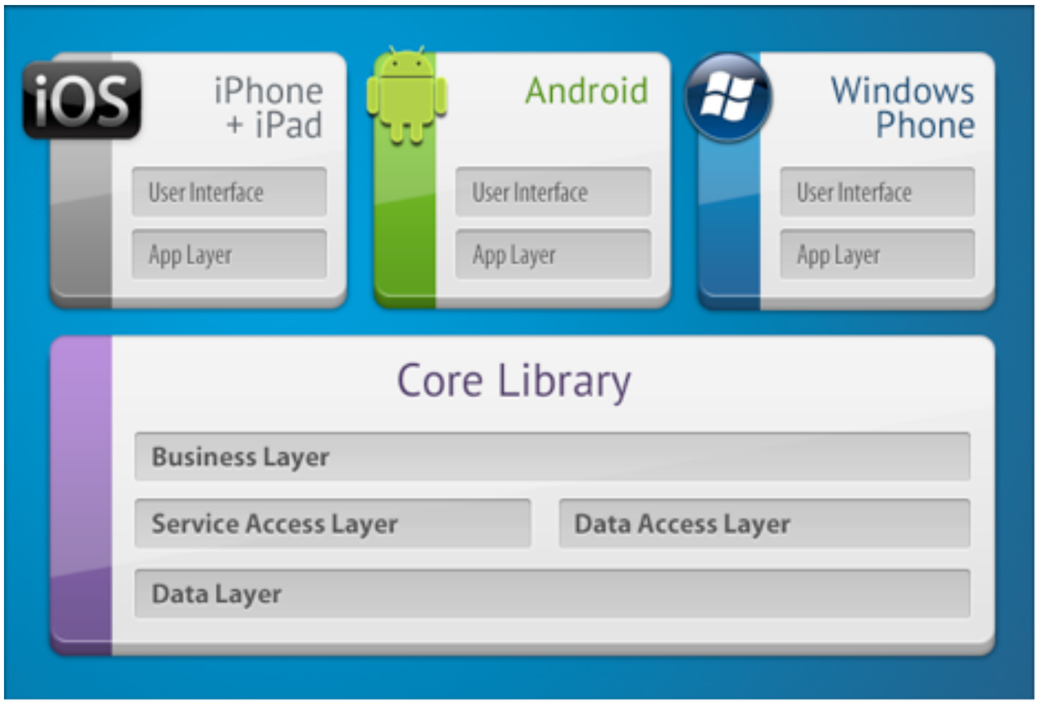
To increase the amount of code re-use, you can adopt cross-platform components that provide common services across all these systems such as:

- SQLite-NET for local SQL storage,
- Xamarin.Mobile for accessing device-specific capabilities including the camera, contacts and geolocation,
- Using framework features for networking, web services, IO and more.

Some of these components are implemented in the *Tasky Pro* and *MWC 2012* case studies.

SEPARATE REUSABLE CODE INTO A CORE LIBRARY

By following the principle of separation of responsibility by layering your application architecture and then moving core functionality that is platform agnostic into a reusable core library, you can maximize code sharing across platforms, as the figure below illustrates:



CASE STUDIES

There are two case studies that accompany this document – *Tasky Pro* and *MWC 2012*. Each case study discusses the implementation of the concepts outlined in this document in a real-world example. The code is open source and available on github.

Understanding the Xamarin Mobile Platform

The Xamarin platform consists of a number of elements that allow you to develop applications for iOS and Android:

- ➔ C# language – Allows you to use a familiar syntax and sophisticated features like Generics, Linq and the Parallel Task Library.
- ➔ Mono .NET framework – Provides a cross-platform implementation of the extensive features in Microsoft's .NET framework.
- ➔ Compiler – Depending on the platform, produces a native app (eg. iOS) or an integrated .NET application and runtime (eg. Android). The compiler also performs many optimizations for mobile deployment such as linking away un-used code.
- ➔ IDE tools – The MonoDevelop IDE and the Xamarin plug-in for Visual Studio allow you to create, build and deploy Xamarin projects.

In addition, because the underlying language is C# with the .NET framework, projects can be structured to share code that can also be deployed to Windows Phone.

Under the Hood

Although Xamarin allows you to write apps in C#, and share the same code across multiple platforms, the actual implementation on each system is very different.

COMPILATION

The C# source makes its way into a native app in very different ways on each platform:

- ➔ iOS – C# is ahead-of-time (AOT) compiled to ARM assembly language. The .NET framework is included, with unused classes being stripped out during linking to reduce the application size. Apple does not allow runtime code generation on iOS, so some language features are not available (see MonoTouch Limitations).
- ➔ Android – C# is compiled to IL and packaged with MonoVM + JIT'ing. Unused classes in the framework are stripped out during linking. The application runs side-by-side with Java/Dalvik and interacts with the native types via JNI (see Mono for Android Limitations).
- ➔ Windows Phone – C# is compiled to IL and executed by the built-in runtime, and does not require Xamarin tools. Designing Windows Phone

applications following Xamarin's guidance makes it simpler to re-use the code on iOS and Android.

The linker documentation for MonoTouch and Mono for Android provides more information about this part of the compilation process.

PLATFORM SDK ACCESS

Xamarin makes the features provided by the platform-specific SDK easily accessible with familiar C# syntax:

➔ iOS – MonoTouch exposes Apple's CocoaTouch SDK frameworks as namespaces that you can reference from C#. For example the UIKit framework that contains all the user interface controls can be included with a simple `using MonoTouch.UIKit;` statement.

➔ Android – Mono for Android exposes Google's Android SDK as namespaces, so you can reference any part of the supported SDK with using statement, such as `using Android.Views;` to access the user interface controls.

➔ Windows Phone – Windows Phone is not part of the Xamarin platform. When building apps for Windows Phone in C# the SDK is implicitly available to your application, including Silverlight/XAML controls for the user interface.

SEAMLESS INTEGRATION FOR DEVELOPERS

The beauty of Xamarin is that despite the differences under the hood, MonoTouch and Mono for Android (coupled with Microsoft's Windows Phone SDK) offer a seamless experience for writing C# code that can be re-used across all three platforms.

Business logic, database usage, network access and other common functions can be written once and re-used on each platform, providing a foundation for platform-specific user interfaces that look and perform as a native application.

Integrated Development Environment (IDE) Availability

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

Development Platform / Target Platform	Mac OS X	Windows	
	MonoDevelop	MonoDevelop	Visual Studio
iOS	Y	-	-
Android	Y	Y	Y
Windows Phone	-	-	Y

Because iOS apps can only be developed on a Mac, and Windows Phone apps can only be developed on Windows, it is impossible to develop for all three platforms on the same operating system. However following the guidance in this document it is possible to reuse code across all these platforms.

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

IOS

Developing iOS applications requires a Mac computer, running Mac OS X.

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 Xamarin's *MonoDevelop* IDE and screen layouts can be edited with Apple's *Interface Builder*. Refer to the MonoTouch Installation Guide for detailed instructions.

ANDROID

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 *MonoDevelop* IDE
- ➔ Windows 7 or 8 with the *MonoDevelop* 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 Mono for 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.

Creating the User Interface (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 CREATE 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.

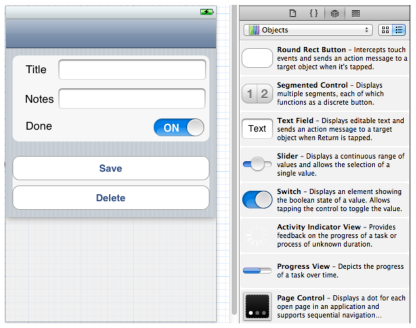
Programmatically creating controls does have benefits though, particularly on iOS for building views that resize or render differently across the iPhone and iPad screen sizes.

VISUAL DESIGNER

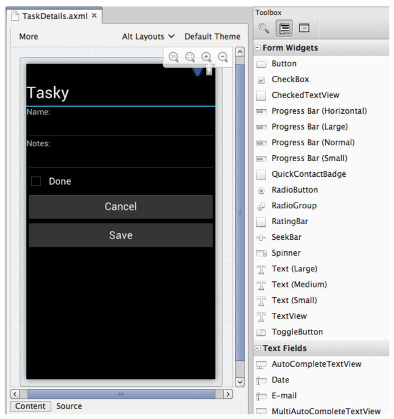
Each platform has a different method for visually laying out screens:

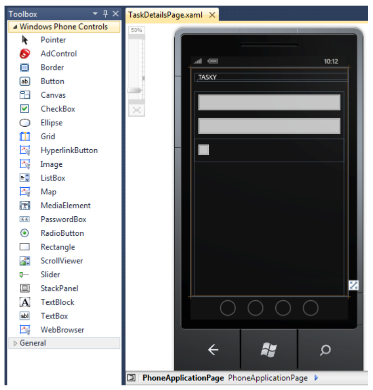
- ➔ iOS – MonoDevelop integrates with Apple's Xcode Interface Builder which allows you to create individual screen layouts or storyboards that describe multiple screens. This results in .XIB or .STORYBOARD files that are included in your project.
- ➔ Android – Xamarin provides an Android drag-and-drop UI designer for both MonoDevelop 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:



Round Rect Button – Intercepts touch events and sends an action message to a target object when it's tapped.
Segmented Control – Displays multiple segments, each of which functions as a discrete button.
Text Field – Displays editable text and sends an action message to a target object when Return is tapped.
Slider – Displays a continuous range of values and allows the selection of a single value.
Switch – Displays an element showing the boolean state of a value. Allows tapping the control to toggle the value.
Activity Indicator View – Provides feedback on the progress of a task or process of unknown duration.
Progress View – Depicts the progress of a task over time.
Page Control – Displays a dot for each open page in an application and supports sequential navigation...





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

Library and Code Re-use

The Xamarin platform allows re-use of existing C# code across all platforms as well as the integration of libraries written natively for each platform.

C# SOURCE AND LIBRARIES

Because Xamarin products use C# and the .NET framework, lots of existing source code (both open source and in-house projects) can be re-used in MonoTouch or Mono for Android projects. Often the source can simply be added to a Xamarin solution and it will work immediately. If an unsupported .NET framework feature has been used, some tweaks may be required.

Examples of C# source that can be used in MonoTouch or Mono for Android include: SQLite-NET, NewtonSoft.JSON and SharpZipLib.

OBJECTIVE-C BINDINGS + BINDING PROJECTS

Xamarin provides a tool called *btouch* that helps create bindings that allow Objective-C libraries to be used in MonoTouch projects. Refer to the Binding Objective-C Types documentation for details on how this is done.

Examples of Objective-C libraries that be used in MonoTouch include: RedLaser barcode scanning, Google Analytics and PayPal integration. Open-source MonoTouch bindings are available on github.

.JAR BINDINGS + BINDING PROJECTS

Xamarin supports using existing Java libraries in Mono for Android. Refer to the Binding a Java Library documentation for details on how to use a .JAR file from Mono for Android.

Open-source Mono for Android bindings are available on github.

C VIA PINVOKE

"Platform Invoke" technology (P/Invoke) allows managed code (C#) to call methods in native libraries as well as support for native libraries to call back into managed code.

For example, the SQLite-NET library uses statements like this:

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

to bind to the native C-language SQLite implementation in iOS and Android. Developers familiar with an existing C API can construct a set of C# classes to map to the native API and utilize the existing platform code. There is documentation for linking native libraries in MonoTouch, similar principles apply to Mono for Android.

C++ VIA CXXI

Miguel explains CXXI on his blog. An alternative to binding to a C++ library directly is to create a C wrapper and bind to that via P/Invoke.

TODO

Architecture

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 Object Oriented Programming principles helps build a well-architected application:

- ➔ Encapsulation – Ensuring that classes and even architectural layers only expose a minimal API that performs their required functions, and hides the implementation details. At a class level, this means that objects behave as 'black boxes' and that consuming code does not need to know how they accomplish their tasks. At an architectural level, it means implementing patterns like Façade that encourage a simplified API that orchestrates more complex interactions on behalf of the code in more abstract layers. This means that the UI code (for example) 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.
- ➔ Separation of Responsibilities – Ensure that each component (both at architectural and class level) has a clear and well-defined purpose. Each component should perform only its defined tasks and expose that

functionality via an API that is accessible to the other classes that need to use it.

- ➔ Polymorphism – Programming to an interface (or abstract class) that supports multiple implementations means that core code can be written and shared across platforms, while still interacting with platform-specific features.

The natural outcome is an application modeled after real world or abstract entities with separate logical layers. Separating code into layers make applications easier to understand, test and maintain. It is recommended that the code in each layer be physically separate (either in directories or even separate projects for very large applications) as well as logically separate (using namespaces).

Typical Application Layers

Throughout this document and the case studies we refer to the following six application layers:

- ➔ Data Layer (DL) – Non-volatile data persistence, likely to be a SQLite database but could be implemented with 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. Candidate for Business Façade pattern.
- ➔ Service Access Layer (SAL) – Used to access services in the cloud: from complex web services (REST, JSON, WCF) to simple retrieval of data and images from remote servers. Encapsulates the networking behavior and provides a simple API to be consumed by the Application and UI layers.
- ➔ Application Layer (AL) – Code that's typically platform specific (not generally shared across platforms) or code that is specific to the application (not generally reusable). A good test of whether to place code in the Application Layer versus the UI Layer is (a) to determine whether the class has any actual display controls or (b) whether it could be shared between multiple screens or devices (eg. iPhone and iPad).
- ➔ 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 because the operations are extremely basic.

Common Mobile Software Patterns

Patterns are an established way to capture recurring solutions to common problems. There are a few key patterns that are useful to understand in building maintainable/understandable mobile applications.

- ➔ Model, View, Controller (MVC) – A common and often misunderstood pattern, MVC is most often used when building User Interfaces and 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.
- ➔ Business Façade – AKA Manager Pattern, provides a simplified point of entry for complex work. For example, in a Task Tracking application, you might have 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 tasks objects.
- ➔ Singleton – The Singleton pattern provides for a way in which only a single instance of a particular object can ever exist. For example, when using SQLite in mobile applications, you only ever want one instance of the database. Using the Singleton pattern is a simple way to ensure this.
- ➔ Provider – A pattern coined by Microsoft (arguably similar to Strategy, or basic Dependency Injection) to encourage code re-use across Silverlight, WPF and WinForms applications. Shared code can be written against an interface or abstract class, and platform-specific concrete implementations are written and passed in when the code is used.
- ➔ Async – Not to be confused with the Async keyword, the Async pattern is used when long-running work needs to be executed without holding up the UI or current processing. In its simplest form, the Async pattern simply describes that long-running tasks should be kicked off in another thread (or similar thread abstraction such as a Task) while the current thread continues to process and listens for a response from the background process, and then updates the UI when data and or state is returned.

Each of the patterns will be examined in more detail as their practical use is illustrated in the case studies. Wikipedia has more detailed descriptions of the Facade, Singleton, Strategy and Provider patterns (and of Design Patterns generally).

Setting Up a Xamarin Cross-Platform Solution

Regardless of what platforms are being used, Xamarin projects all use the same solution file format (the Visual Studio ".SLN" file format). Solutions can be shared across development environments, even when individual projects cannot be loaded (such as a MonoTouch project in Visual Studio).

When creating a new cross platform application, the first step is to create a blank solution. This section what happens next: setting up the projects for building cross platform mobile apps.

Sharing Code

Refer to the [Code Sharing Strategies] document for a detailed description of how to implement code-sharing across platforms.

PLATFORM-SPECIFIC LIBRARY PROJECTS

If a Portable Class Library is not suitable for your needs, it is possible to share code across platform-specific applications using file linking or cloned project files. These project types remain targeted to a specific framework, and share code at the individual C# file level. Both these mechanisms require manual synchronization of the files included in the project.

Platform-specific projects allow the definition and use of compiler directives to include different code-paths in the shared code. They also allow you to reference platform specific assemblies (so long as all the code referencing the assembly is also bounded by a compiler directive).

PORTABLE CLASS LIBRARIES (PCL)

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 MonoTouch and Mono for Android, as well as Silverlight, WPF, Windows Phone and Xbox. The library can only utilize a subset of the complete .NET framework, limited by the platforms being targeted.

Xamarin's support for Portable Class Libraries is currently under development.

Populating the Solution

Regardless of which method is used to share code, the overall solution structure should implement a layered architecture that encourages code sharing. The Xamarin approach is to group code into two project types:

- ➔ Core project – Write re-usable code in one place, to be shared across different platforms. Use the principles of encapsulation to hide implementation details wherever possible.
- ➔ Platform-specific application projects – Consume the re-usable code with as little coupling as possible. Platform-specific features are added at this level, built on components exposed in the Core project.

Within each project the source is grouped physically (using directories) and logically (using namespaces) to enforce architectural separation and make large applications easier to manage.

CORE PROJECT

Shared code projects should only reference assemblies that are available across all platforms – ie. the common framework namespaces like `System`, `System.Core` and `System.Xml`.

Shared projects should implement as much non-UI functionality as is possible, which generally includes the following layers:

➔ Data Layer – Code that takes care of physical data storage eg. SQLite-NET, an alternative database like CoolStorage or even XML files. The data layer classes are normally only used by the data access layer.

➔ Data Access Layer – Defines an API that supports the required data operations for the application's functionality, such as methods to access lists of data, individual data items and also

➔ Service Access Layer – An optional layer to provide cloud services to the application. Contains code that accesses remote network resources (web services, image downloads, etc) and possibly caching of the results.

➔ Business Layer – Definition of the Model classes and the Façade or Manager classes that expose functionality to the platform-specific applications.

PLATFORM-SPECIFIC APPLICATION PROJECTS

Platform-specific projects must reference the assemblies required to bind to each platform's SDK (MonoTouch, Mono for Android or Windows Phone) as well as the Core shared code project.

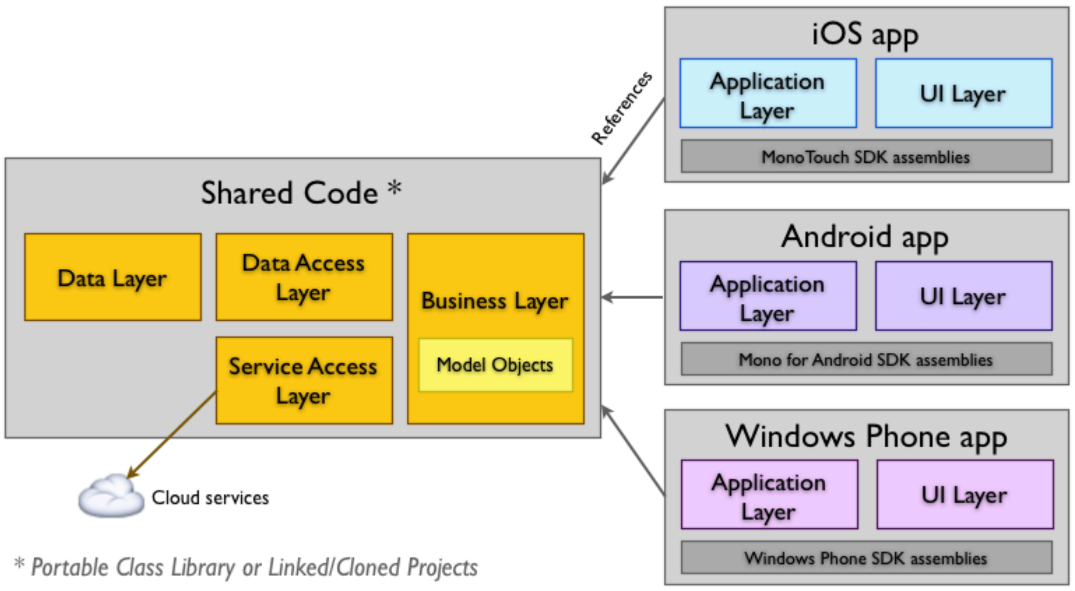
The platform-specific projects should implement:

➔ Application Layer – Platform specific functionality and binding/conversion between the Business Layer objects and the user interface.

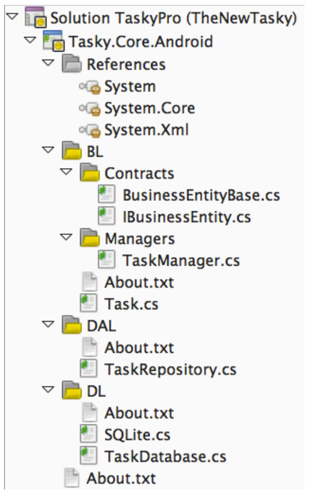
➔ User Interface Layer – Screens, custom user-interface controls, presentation of validation logic.

EXAMPLE

The application architecture is illustrated in this diagram:

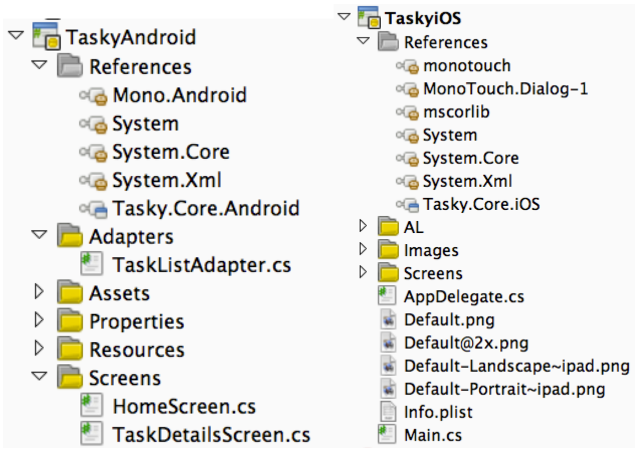


These screenshots show a solution setup with the shared Core project, iOS and Android App Projects. The Core project contains folders for each of the architectural layers, plus minimal references to .NET framework assemblies.



```
Solution TaskyPro (TheNewTasky)
  Tasky.Core.Android
    References
      System
      System.Core
      System.Xml
    BL
      Contracts
        BusinessEntityBase.cs
        IBusinessEntity.cs
      Managers
        TaskManager.cs
      About.txt
      Task.cs
    DAL
      About.txt
      TaskRepository.cs
    DL
      About.txt
      SQLite.cs
      TaskDatabase.cs
    About.txt
```

The application projects each reference their respective platform SDKs (MonoTouch and Mono.Android) as well as the Core project, and contain the user-interface code required to present functionality to the user.



```
TaskyAndroid
  References
    Mono.Android
    System
    System.Core
    System.Xml
    Tasky.Core.Android
  Adapters
    TaskListAdapter.cs
  Assets
  Properties
  Resources
  Screens
    HomeScreen.cs
    TaskDetailsScreen.cs
```

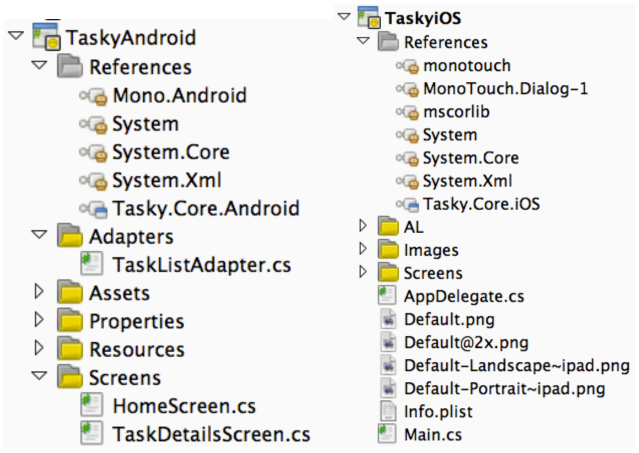
```
TaskyiOS
  References
    monotouch
    MonoTouch.Dialog-1
    mscorlib
    System
    System.Core
    System.Xml
    Tasky.Core.iOS
  AL
  Images
  Screens
  AppDelegate.cs
  Default.png
  Default@2x.png
  Default-Landscape~ipad.png
  Default-Portrait~ipad.png
  Info.plist
  Main.cs
```

Specific examples of how projects should be structured are given in the case studies.

Project References

Project references reflect the dependencies for a project. Core projects limit their references to common assemblies so that the code is easy to share. Platform-specific application projects reference the shared Core, plus any other platform-specific assemblies they need to take advantage of the target platform.

The application projects each reference their respective platform SDKs (MonoTouch and Mono.Android) as well as the Core project, and contain the user-interface code required to present functionality to the user, as shown in these screenshots:



```
TaskyAndroid
  References
    Mono.Android
    System
    System.Core
    System.Xml
    Tasky.Core.Android
  Adapters
    TaskListAdapter.cs
  Assets
  Properties
  Resources
  Screens
    HomeScreen.cs
    TaskDetailsScreen.cs
```

```
TaskyiOS
  References
    monotouch
    MonoTouch.Dialog-1
    mscorlib
    System
    System.Core
    System.Xml
    Tasky.Core.iOS
  AL
  Images
  Screens
  AppDelegate.cs
  Default.png
  Default@2x.png
  Default-Landscape~ipad.png
  Default-Portrait~ipad.png
  Info.plist
  Main.cs
```

Specific examples of how projects should be structured are given in the case studies.

Adding Files

As discussed in the [Code Sharing Strategies] document, when adding shared-code files the same file must be manually added or linked in each Core project (one for each platform).

The platform-specific application projects can be treated like any regular .NET project: code and resources should be managed independently of the other projects. Add code, image, XML and other files directly to the platform-specific projects.

BUILD ACTION

It is important to set the correct build-action for certain file types. This list shows the build action for some common file types:

- All C# files – Build Action: Compile
- Images in MonoTouch & Windows Phone – Build Action: Content
- XIB and Storyboard files in MonoTouch – Build Action: InterfaceDefinition
- Images and AXML layouts in Android – Build Action: AndroidResource
- XAML files in Windows Phone – Build Action: Page

Generally the IDE will detect the file type and suggest the correct build action.

CASE SENSITIVITY

Finally, remember that some platforms have case-sensitive file systems (eg. iOS and Android) so be sure to use a consistent file naming standard and make sure that the file names you use in code match the filesystem exactly.

Handling Platform Divergence & Features

Divergence isn't just a 'cross-platform' problem; devices on the 'same' platform have different capabilities (especially the wide variety of Android devices that are available). The most obvious and basic is screen size, but other device attributes can vary and require an application to check for certain capabilities and behave differently based on their presence (or absence).

This means all applications need to deal with graceful degradation of functionality, or else present an unattractive, lowest-common-denominator feature set. Xamarin's deep integration with the native SDKs of each platform allow applications to take advantage of platform-specific functionality, so it makes sense to design apps to use those features.

See the Platform Capabilities documentation for an overview of how the platforms differ in functionality.

Examples of Platform Divergence

FUNDAMENTAL ELEMENTS THAT EXIST ACROSS PLATFORMS

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 – Some platforms (like iOS and Windows Phone) have standardized screen sizes that are relatively simple to target. Android devices have a large variety of screen dimensions, which require more effort to support in your application.
- Navigation metaphors – Differ across platforms (eg. hardware ‘back’ button, Panorama UI control) and within platforms (Android 2 and 4, iPhone vs iPad).
- Keyboards – Some Android devices have physical keyboards while others only have a software keyboard. 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 – Functionality differs across 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 by each method.

- ➔ **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 services' API.
- ➔ **Near Field Communications (NFC)** – 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.

- ➔ **Platform Abstraction** – Business Façade pattern, provides a unified access across platforms and abstracts the particular platform implementations into a single, unified API.
- ➔ **Divergent Implementation** – Invocation of specific platform features via divergent implementations via architectural tools such as interfaces and inheritance or conditional compilation.

Platform Abstraction

Class Abstraction

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.

Advantages

The implementation can contain platform-specific code and even reference platform-specific external libraries.

Disadvantages

Having to create and pass implementations into the shared code. If the interface is used deep within the shared code then it ends up being passed through multiple method parameters or otherwise pushed down through the call chain. If the shared code uses lots of different interfaces then they must all be created and set in the shared code somewhere.

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).

MonoTouch.Dialog and MonoDroid.Dialog

The 'scrolling list' user interface is a common feature on all mobile platforms. It is implemented via the `UITableView` class on iOS and the `ListView` on Android. Each of these controls requires the creation of additional classes to populate and display their data.

MonoTouch.Dialog provides an abstract API that implements platform-independent `Elements` (or Reflection against a class) without any knowledge of the underlying `UITableView` class. See the Introduction to MonoTouch.Dialog for more information on building screens using MonoTouch.Dialog.

MonoDroid.Dialog is still being developed but aims to provide a similar API for quickly and easily populating `ListViews` in Mono for Android. The source (including a number of forks) for MonoDroid.Dialog is available on github.

Xamarin.Mobile

The Xamarin.Mobile 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.

The services currently offered by Xamarin.Mobile include:

- ➔ Media Picker for taking photos and videos, or selecting existing photos and videos
- ➔ Geolocation information
- ➔ Access to the device address book

Because the API is consistent across all platforms you can write shared code (that references the platform-specific implementation) that can interact with these features, regardless of the underlying operating system.

The assemblies can be downloaded from http://xamarin.com/MobileAPI

Other Cross-Platform Libraries

There are a number of 3rd party libraries available that provide cross-platform functionality:

- ➔ MonoCross - http://code.google.com/p/monocross/
- ➔ Vernacular (for localization) - https://github.com/rdio/vernacular/
- ➔ MonoGame (for XNA games) - http://monogame.codeplex.com/
- ➔ CrossGraphics - https://github.com/praeclarum/CrossGraphics

Divergent Implementation

Conditional Compilation

There are some situations where your shared code will still need to work differently on each platform, possibly accessing classes or features that behave differently. Conditional compilation works best when the same source file is being referenced in multiple projects that have different symbols defined (eg. file linking or clone project files).

IOS

MonoTouch does not include any conditional symbols. You could define your own in the project configuration, but you have to remember to do this for each MonoTouch library and all of its compilation configurations (such as debugging, release as well as AdHoc and App Store distribution).

ANDROID

Code that should only be compiled into Mono for Android applications can use the following

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

Each API version also defines a new compiler directive, so code like this will let you add features if newer APIs are targeted. Each API level includes all the 'lower' level symbols. This feature is not really useful for supporting multiple platforms; typically the `__ANDROID__` symbol will be sufficient.

```
#if __ANDROID_11__
// code that should only run on Android 3.0 Honeycomb or newer
#endif
```

WINDOWS PHONE

Windows Phone 7 defines two symbols – `WINDOWS_PHONE` and `SILVERLIGHT` – that can be used to target code to the platform.

USING CONDITIONAL COMPILATION

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;
}
```

The result is a class that can be built and used on all three platforms, placing the SQLite database file in a different location on each platform.

Practical Code Sharing Strategies

This section gives examples of how to share code for common application scenarios.

Data Layer

The data layer consists of a storage engine and methods to read and write information. For performance, flexibility and cross-platform compatibility the

SQLite database engine is recommended for Xamarin cross-platform applications. It runs on a wide variety of platforms including Windows, Windows Phone, Android, iOS and Mac.

SQLITE

SQLite is an open-source database implementation. The source and documentation can be found at SQLite.org. SQLite support is available on each mobile platform:

- ➔ iOS – Built in to the operating system.
- ➔ Android – Built in to the operating system since Android 2.2 (API Level 10).
- ➔ Windows Phone – Can support SQLite by shipping the open-source C# SQLite assembly with your app. More information can be found at C# SQLite on Google Code.

Even with the database engine available on all platforms, the native methods to access the database are different. Both iOS and Android offer built-in APIs to access SQLite that could be used from MonoTouch or Mono for Android, however using the native SDK methods offers no ability to share code (other than perhaps the SQL queries themselves, assuming they're stored as strings). For details about native database functionality search for `CoreData` in iOS or Android's `SQLiteOpenHelper` class; because these options are not cross-platform they are beyond the scope of this document.

ADO.NET

Both MonoTouch and Mono for Android support `System.Data` and `Mono.Data.Sqlite` (see the MonoTouch documentation for more info). Using these namespaces allows you to write ADO.NET code that works on both platforms. Edit the project's references to include `System.Data.dll` and `Mono.Data.Sqlite.dll` and add these using statements to your code:

```
using System.Data;
using Mono.Data.Sqlite;
```

Then the following sample code will work:

```
string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "items.db3");
bool exists = File.Exists (dbPath);
if (!exists)
    SqliteConnection.CreateFile (dbPath);
var connection = new SqliteConnection ("Data Source=" + dbPath);
connection.Open ();
if (!exists) {
    // This is the first time the app has run and/or that we need the DB.
    // Copy a "template" DB from your assets, or programmatically create
one like this:
    var commands = new[]{
        "CREATE TABLE [Items] (Key ntext, Value ntext);",
        "INSERT INTO [Items] ([Key], [Value]) VALUES ('sample', 'text')"
        };
        foreach (var command in commands) {
            using (var c = connection.CreateCommand ()) {
                c.CommandText = command;
                c.ExecuteNonQuery ();
            }
        }
    }
    // use `connection`... here, we'll just append the contents to a TextView
    using (var contents = connection.CreateCommand ()) {
        contents.CommandText = "SELECT [Key], [Value] from [Items]";
        var r = contents.ExecuteReader ();
        while (r.Read ())
            Console.Write("\n\tKey={0}; Value={1}",
                    r ["Key"].ToString (),
                    r ["Value"].ToString ());
    }
    connection.Close ();
```

Real-world implementations of ADO.NET would obviously be split across different methods and classes (this example is for demonstration purposes only).

SQLITE-NET – CROSS-PLATFORM ORM

An ORM (or Object-Relational Mapper) attempts to simplify storage of data modeled in classes. Rather than manually writing SQL queries that CREATE TABLEs or SELECT, INSERT and DELETE data that is manually extracted from class fields and properties, an ORM adds a layer of code that does that for you. Using reflection to examine the structure of your classes, an ORM can automatically create tables and columns that match a class and generate queries to read and write the data. This allows application code to simply send and retrieve object instances to the ORM, which takes care of all the SQL operations under the hood.

SQLite-NET acts as a simple ORM that will allow you to save and retrieve your classes in SQLite. It hides the complexity of cross platform SQLite access with a combination of compiler directives and other tricks.

Features of SQLite-NET:

- ➔ Tables are defined by adding attributes to Model classes.
- ➔ A database instance is represented by a subclass of `SQLiteConnection`, the main class in the SQLite-Net library.
- ➔ Data can be inserted, queried and deleted using objects. No SQL statements are required (although you can write SQL statements if required).
- ➔ Basic Linq queries can be performed on the collections returned by SQLite-NET.

The source code and documentation for SQLite-NET is available at SQLite-Net on github and has been implemented in both case-studies. A simple example of SQLite-NET code (from the *Tasky Pro* case study) is shown below.

First, the `Task` class uses attributes to define a field to be a database primary key:

```
public class Task : IBusinessEntity
{
    public Task () {}
    [PrimaryKey, AutoIncrement]
    public int ID { get; set; }
    public string Name { get; set; }
    public string Notes { get; set; }
    public bool Done { get; set; }
}
```

This allows a “Task” table to be created with the following line of code (and no SQL statements) on an `SQLiteConnection` instance:

```
CreateTable<Task> ();
```

Data in the table can also be manipulated with other methods on the `SQLiteConnection` (again, without requiring SQL statements):

```
Insert (task); // 'task' is an instance with data populated in its
properties
Update (task); // Primary Key field must be populated for Update to work
Table<Task>.ToList(); // returns all rows in a collection
```

See the case study source code for complete examples.

C# SQLITE

The iOS and Android operating systems include a natively-compiled version of SQLite in the operating system, however Windows Phone 7 does not include a compatible database engine. SQLite has been ported to C# and is open-sourced on Google Code. This code has been downloaded and built as a Windows Phone assembly that is included in the case studies, so that the same database (SQLite) can be used across all three platforms.

The Windows Phone apps in the case studies must reference the `Community.CsharpSqlite.WP7.dll` in their Core projects in order to share the SQLite-NET implementation and database code.

File Access

File access is certain to be a key part of any application. Common examples of files that might be part of an application include:

- SQLite database files.
- User-generated data (text, images, sound, video).
- Downloaded data for caching (images, html or PDF files).

SYSTEM.IO DIRECT ACCESS

Both MonoTouch and Mono for Android allow file system access using classes in the `System.IO` namespace.

Each platform does have different access restrictions that must be taken into consideration:

- iOS applications run in a sandbox with very restricted file-system access. Apple further dictates how you should use the file system by specifying certain locations that are backed-up (and others that are not). Refer to the Working with the File System in MonoTouch document for more details.
- Android also restricts access to certain directories related to the application, but it also supports external media (eg. SD cards) and accessing shared data.
- Windows Phone 7 does not allow direct file access – files can only be manipulated using isolated storage.

A trivial example that writes and reads a text file is shown below. Using `Environment.GetFolderPath` allows the same code to run on iOS and Android, which each return a valid directory based on their filesystem conventions.

```
string filePath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "MyFile.txt");
System.IO.WriteAllText (filePath, "Contents of text file");
Console.WriteLine (System.IO.WriteAllText (filePath));
```

Refer to the MonoTouch Working with the File System document for more information on iOS-specific filesystem functionality. When writing cross-platform file access code, remember that some file-systems are case-sensitive and have different directory separators. It is good practice to always use the same casing for filenames and the `Path.Combine()` method when constructing file or directory paths.

ISOLATED STORAGE

Isolated Storage is a common API for saving and loading files across all three platforms.

It is the default mechanism for file access in Windows Phone that has been implemented in MonoTouch and Mono for Android to allow common file-access code to be written. The `System.IO.IsolatedStorage` class can be referenced across all three platforms in a shared-code project.

Refer to the Isolated Storage Overview for Windows Phone for more information.

Network Operations

Most mobile applications will have networking component, for example:

- Downloading images, video and audio (eg. thumbnails, photos, music).
- Downloading documents (eg. HTML, PDF).
- Uploading user data (such as photos or text).
- Accessing web services or 3rd party APIs (including SOAP, XML or JSON).

The .NET Framework provides two main classes for accessing network resources: `WebClient` and `HttpWebRequest`.

WEBCLIENT

The `WebClient` class provides a simple API to retrieve remote data from remote servers.

Windows Phone operations *must* be async, even though MonoTouch and Mono for Android support synchronous operations (which can be done on background threads).

The code for a simple asychronous `WebClient` operation is:

```
var webClient = new WebClient ();
webClient.DownloadStringCompleted += (sender, e) =>
{
    var resultString = e.Result;
    // do something with downloaded string, do UI interaction on main
thread
};
webClient.Encoding = System.Text.Encoding.UTF8;
webClient.DownloadStringAsync (new Uri ("http://some-
server.com/file.xml"));
```

`WebClient` also has `DownloadFileCompleted` and `DownloadFileAsync` for retrieving binary data.

The *MWC 2012* case study uses `WebClient` to download conference data. The code is in the Core project and is shared across iOS, Android and Windows Phone (refer to the `MwcSiteParser` class).

HTTPWEBREQUEST

`HttpWebRequest` offers more customization than `WebClient` and as a result requires more code to use.

The code for a simple synchronous `HttpWebRequest` operation is:

```
var request = HttpWebRequest.Create(@"http://some-server.com/file.xml ");
request.ContentType = "text/xml";
request.Method = "GET";
using (HttpWebResponse response = request.GetResponse() as
HttpWebResponse)
{
    if (response.StatusCode != HttpStatusCode.OK)
        Console.WriteLine("Error fetching data. Server returned status
code: {0}", response.StatusCode);
    using (StreamReader reader = new
StreamReader(response.GetResponseStream()))
    {
        var content = reader.ReadToEnd();
        // do something with downloaded string, do UI interaction on main
thread
    }
}
```

There is an example in our Web Services documentation, and you can read about the class for Windows Phone on MSDN.

REACHABILITY

Mobile devices operate under a variety of network conditions from fast Wi-Fi or 4G connections to poor reception areas and slow EDGE data links. Because of this, it is good practice to detect whether the network is available and if so, what type of network is available, before attempting to connect to remote servers.

Actions a mobile app might take in these situations include:

- ➔ If the network is unavailable, advise the user. If they have manually disabled it (eg. Airplane mode or turning off Wi-Fi) then they can resolve the issue.
- ➔ If the connection is 3G, applications may behave differently (for example, Apple does not allow apps larger than 20Mb to be downloaded over 3G). Applications could use this information to warn the user about excessive download times when retrieving large files.
- ➔ Even if the network is available, it is good practice to verify connectivity with the target server before initiating other requests. This will prevent the app's network operations from timing out repeatedly and also allow a more informative error message to be displayed to the user.

There is a MonoTouch sample available (which is based on Apple's Reachability sample code) to help detect network availability. The *MWC 2012* case study iOS app includes an example of the reachability code.

WebServices

See our documentation on Working with Web Services, which covers accessing REST, SOAP and WCF endpoints using MonoTouch. It is possible to hand-craft web service requests and parse the responses, however there are libraries available to make this much simpler, including RestSharp and ServiceStack. Even basic WCF operations can be accessed in Xamarin apps.

RESTSHARP

RestSharp is a .NET library that can be included in mobile applications to provide a REST client that simplifies access to web services. It helps by providing a simple API to request data and parse the REST response. RestSharp can be useful

The RestSharp website contains documentation on how to implement a REST client using RestSharp. RestSharp provides MonoTouch and Mono for Android examples on github.

There is also a MonoTouch code snippet in our Web Services documentation.

SERVICESTACK

Unlike RestSharp, ServiceStack is both a server-side solution to host a web service as well as a client library that can be implemented in mobile applications to access those services.

The ServiceStack website explains the purpose of the project and links to document and code samples. The examples include a complete server-side implementation of a web service as well as various client-side applications that can access it.

There is a MonoTouch example on the ServiceStack website, and a code snippet in our Web Services documentation.

WCF

Xamarin tools can help you consume the same Windows Communication Foundation (WCF) services as similar .NET clients. In general, Xamarin supports the same client-side subset of WCF that ships with the Silverlight runtime. This includes the most common encoding and protocol implementations of WCF: text-encoded SOAP messages over the HTTP transport protocol using the BasicHttpBinding.

Due to the size and complexity of the WCF framework, there may be current and future service implementations that will fall outside of the scope supported by Xamarin's client-subset domain. In addition, WCF support requires the use of tools only available in a Windows environment to generate the proxy. Therefore, WCF support should currently be considered "in preview."

Threading

Application responsiveness is important for mobile applications – users expect applications to load and perform quickly. A 'frozen' screen that stops accepting user-input will appear to indicate the application has crashed, so it is important not to tie up the UI thread with long-running blocking calls such as network requests or slow local operations (such as unzipping a file). In particular the startup process should not contain long-running tasks – all mobile platforms will kill an app that takes too long to load.

This means your user interface should implement a 'progress indicator' or otherwise 'useable' UI that is quick to display, and asynchronous tasks to perform background operations. Executing background tasks requires the use of threads, which means the background tasks needs a way to communicate back to the main thread to indicate progress or when they have completed.

PARALLEL TASK LIBRARY

Tasks created with the Parallel Task Library can run asynchronously and return on their calling thread, making them very useful for triggering long-running operations without blocking the user interface.

A simple parallel task operation might look like this:

```
using System.Threading.Tasks;
void MainThreadMethod ()
{
    Task.Factory.StartNew (() => wc.DownloadString
("http://...")).ContinueWith (
        t => label.Text = t.Result,
TaskScheduler.FromCurrentSynchronizationContext()
    );
}
```

The key is `TaskScheduler.FromCurrentSynchronizationContext()` which will reuse the SynchronizationContext.Current of the thread calling the method (here the main thread that is running `MainThreadMethod`) as a way to marshal back calls to that thread. This means if the method is called on the UI thread, it will run the `ContinueWith` operation back on the UI thread.

If the code is starting tasks from other threads, use the following pattern to create a reference to the UI thread and the task can still call back to it:

```
static Context uiContext =
TaskScheduler.FromCurrentSynchronizationContext();
```

INVOKING ON THE UI THREAD

For code that doesn't utilize the Parallel Task Library, each platform has its own syntax for marshaling operations back to the UI thread:

- ➔ iOS: `owner.BeginInvokeOnMainThread(new NSAction(action));`
- ➔ Android: `owner.RunOnUiThread(action);`
- ➔ Windows: `Deployment.Current.Dispatcher.BeginInvoke(action);`

Both the iOS and Android syntax requires a 'context' class to be available which means the code needs to pass this object into any methods that require a callback on the UI thread.

To make UI thread calls in shared code, follow the IDispatchOnUIThread example (courtesy of @follesoe). Declare and program to an `IDispatchOnUIThread` interface in the shared code and then implement the platform-specific classes as shown here:

```
// program to the interface in shared code
public interface IDispatchOnUIThread {
    void Invoke (Action action);
}
// iOS
public class DispatchAdapter : IDispatchOnUIThread {
    public readonly NSObject owner;
    public DispatchAdapter (NSObject owner) {
        this.owner = owner;
    }
    public void Invoke (Action action) {
        owner.BeginInvokeOnMainThread(new NSAction(action));
    }
}
// Android
public class DispatchAdapter : IDispatchOnUIThread {
    public readonly Activity owner;
    public DispatchAdapter (Activity owner) {
        this.owner = owner;
    }
    public void Invoke (Action action) {
        owner.RunOnUiThread(action);
    }
}
// WP7
public class DispatchAdapter : IDispatchOnUIThread {
    public void Invoke (Action action) {
        Deployment.Current.Dispatcher.BeginInvoke(action);
    }
}
```

Platform and Device Capabilities and Degradation

Further specific examples of dealing with different capabilities are given in the Platform Capabilities documentation. It deals with detecting different capabilities and how to gracefully degrade an application to provide a good user experience, even when the app can't operate to its full potential.

Testing

Many apps (even Android apps, on some stores) will have to pass an approval process before they are published; so testing is critical to ensure your app reaches the market (let alone succeeds with your customers). Testing can take many forms, from developer-level unit testing to managing beta testing across a wide variety of hardware.

Test on All Platforms

There are slight differences between what .NET supports on the Windows Phone and Windows Tablets as well as limitations on iOS that prevent dynamic code to be generated on the fly. Either plan on testing the code on multiple platforms as you develop it, or schedule time to refactor and update the model part of your application at the end of the project.

It is always good practice to use the simulator/emulator to test multiple versions of the operating system and also different device capabilities/configurations.

You should also test on as many different physical hardware devices as you can.

DEVICES IN CLOUD

The mobile phone and tablet ecosystem is growing all the time, making it impossible to test on the ever-increasing number of devices available. To solve this problem a number of services offer the ability to remotely control many different devices so that applications can be installed and tested without needing to directly invest in lots of hardware.

Here are some of the available services (Xamarin does not endorse any particular service):

- ➔ perfecto mobile
- ➔ Keynot DeviceAnywhere
- ➔ Testdroid

Test Management

When testing applications within your organization or managing a beta program with external users, there are two challenges:

- ➔ **Distribution** – Managing the provisioning process (especially for iOS devices) and getting updated versions of software to the testers.
- ➔ **Feedback** – Collecting information about application usage, and detailed information on any errors that may occur.

The following services help to address these issues, by providing infrastructure that is built into your application to collect and report on usage and errors, and also streamlining the provisioning process to help sign-up and manage testers and their devices:

- ➔ TestFlight for iOS
- ➔ LaunchPad for Android

Test Automation

Test Studio from Telerik supports recording and replaying repeatable user interface tests on iOS. By installing an application on each device and including some references in the code, MonoTouch applications can have tests recorded against their user interface and then played back on subsequent builds.

Unit Testing

TOUCH.UNIT

MonoTouch includes a unit-testing framework called Touch.Unit which follows the JUnit/NUnit style writing tests.

Refer to our Unit Testing with MonoTouch documentation for details on writing tests and running Touch.Unit.

ANDR.UNIT

There is an open-source equivalent of Touch.Unit for Android called Andr.Unit. You can download it from github and read about the tool on @spouliot's blog.

WINDOWS PHONE

Here are some links to help setup unit testing for Windows Phone:

- ➔ http://www.jeff.wilcox.name/2010/05/sl3-utf-bits/
- ➔ http://www.jeff.wilcox.name/2011/06/updated-ut-mango-bits/
- ➔ http://www.smartypantscoding.com/a-cheat-sheet-for-unit-testing-silverlight-apps-on-windows-phone-7
- ➔ http://mobile.dzone.com/articles/unit-testing-your-windows

App Store Approvals

Apple and Microsoft operate the only store on their platforms: the App Store and Marketplace respectively. Both lock-down their devices and implement a rigorous app review process to control the quality of applications available to download. Android's open nature means there are a number of store options ranging from Google's Play, which is widely available and has no review process, to Amazon's Appstore for Android and hardware-specific efforts like Samsung Apps which have more limited distribution and implement an approval process.

Waiting for an app to be reviewed can be very stressful - business pressures often mean applications are submitted for approval with very little margin for error prior to a "targeted" launch date. The process itself can take up to two weeks and isn't necessarily transparent: there is limited feedback on the progress of your application until it is finally rejected or approved. Rejection can mean missing a marketing window of opportunity, especially if it happens more than once, and weeks pass between your original launch date and when the app is finally approved.

Be prepared

The first piece of advice: plan your app's launch well in advance and make allowances for a possible rejection and re-submission. Each store requires you to create an account before submitting your app - do this as early as possible. While Google Play's signup only takes a few minutes if your apps are free, the process gets a lot more involved if you are selling them and need to supply banking and tax information. Apple and Microsoft's processes are both much more involved than Google's, it could take a week or more to get your account approved so factor this time into your launch plans.

Once your account has been approved, you're ready to submit an app. The actual process to submit apps is covered in the following documentation:

- ➔ Publishing to Apple's iOS App Store
- ➔ Preparing an app for Google Play
- ➔ Windows Phone developers should visit Microsoft's AppHub to read about submitting their apps.

The remainder of this section discusses things you should take into consideration to ensure your app is approved without any hiccups.

Quality

It sounds obvious, but applications will often get rejected because they do not meet a certain level of quality: after all, this is the reason why the curated stores have an approval process in the first place!

Crashes are a common reason for rejection. If it's too easy to make your app crash, it's guaranteed to be rejected. Most developers don't submit their apps with the expectation that they'll crash, yet they often do. Test your app thoroughly before submitting it, focusing not just on making sure everything works but also that you handle common mobile error scenarios such as network problems and resource constraints like memory or storage space. Use both the simulator and physical devices to test - regardless of how well code runs in a simulator, only a device can demonstrate an app's real performance. Use as many different devices as you can find, and enlist a team of beta-testers if you can - services like TestFlight for iOS or LaunchPad for Android can help manage beta distribution and feedback.

All mobile operating systems will kill an application that doesn't start quickly enough. The length of time allowed varies, but in general apps should aim to be responsive in a few seconds and use background tasks to do any work that would take longer. Apps that take too long to load, or are not responsive enough in regular use will be rejected. Always provide user feedback when something is happening in the background, or the app will appear to have crashed and once again, get rejected.

Check Your Edge Cases

A common trap for developers is failing to address edge-cases, especially those that require re-configuring their simulator or device to test properly. It can be easy to forget that not every customer is going to "Allow" your app to access their location because after the developer has accepted the request once, they'll never be prompted again. Permissions and network usage are specifically focussed on during the approval process, which means a small oversight in these areas can result in rejection.

The following list is a good starting point for checking edge-cases that might have been missed:

➔ Customers may 'deny' access to services – especially in iOS, access to data such as geo-location information is only provided if the user grants permission to your application. Application testers should frequently re-install the application in its initial state and disallow any permission requests to ensure the application behaves appropriately. Toggle permission on and off to verify correct behavior as customers change their mind.

➔ Customers are everywhere – don't assume that an app will only be used in the city or country where it was developed! Code that works with GPS coordinates, date and time values and currencies can all be affected by the customer's location and locale settings. All platforms offer a simulator that let you specify different locations and locales - use it to test locations in other hemispheres and with cultures that format dates and currencies differently. Latitude and longitude values can be positive or negative, the decimal separator could be a period or a comma, and dates can be formatted a myriad of ways - deal with it!

➔ Slow network connections – app developers often work in an 'ideal world' of fast, always working network connectivity, which obviously isn't the case in the real world. Testing with slow network connectivity (such as a poor 3G connection) as well as with no network access is critical to ensuring you don't ship a buggy app. The approval process will always include a test with the device in airplane mode, so ensure that you have tested that for yourself.

- ➔ Hardware varies – remember to test on the oldest, slowest hardware that you plan to support. There are two aspects that might affect your app: performance, which might be unusable on an older device, and support for hardware features such as a camera, microphone, GPS, gyroscope or other optional component. Applications should degrade gracefully (and not crash) when a component is unavailable.

Guidelines are more than just a ‘guide’

Apple is famous for being strict about adherence to their Human Interface Guidelines as one of the key strengths of their platform is consistency (and the perceived increase in usability). Microsoft has taken a similar approach with Windows Phone applications implementing the Metro-style UI. The approval process for both platforms will involve your app being evaluated for its adherence to the relevant design philosophy.

That isn’t to say that user interface innovation isn’t supported or encouraged, but there are certain things you “just shouldn’t do” or your app will be rejected.

On iOS, this includes misusing built-in icons or using other well established metaphors in a non-consistent manner; for example using the ‘compose’ icon for anything other than creating new content.

Windows Phone developers should be similarly careful; a common mistake is failing to properly support the hardware Back button according to Microsoft’s guidelines.

Encourage your designers to read and follow the design guidelines for each platform.

Implementing Platform-Specific Features

Things are a little stricter when it comes to implementing platform-specific services, especially on iOS. To avoid automatic-rejection by Apple, there are some rules to follow with the following iOS features:

- ➔ In-App purchases – Applications must NOT implement external payment mechanisms for digital products including in-game currency, application features, magazine subscriptions and a lot more. iOS apps must use Apple’s iTunes-based service for this kind of functionality. There is a loophole - apps like the Kindle Reader and some subscription-based apps let you purchase content elsewhere that gets attached to an “account” which you can then access via the app, however in this case the app must not contain links or references to the out-of-app purchase process (or, once again, it’ll be rejected).
- ➔ iCloud backup – With the advent of iCloud, Apple’s reviewers are much more strict regarding how apps use storage (to ensure customer’s remote backup experience is pleasant). Apps that waste backup-able storage

space may get rejected, so use the Cache folder appropriately and follow Apple's other storage-related guidelines.

- ➔ Newsstand – Newspaper and magazine apps are a great fit for Apple's Newsstand, however apps must implement at least one auto-renewing subscription and support background downloading to be approved.
- ➔ Maps – It's increasingly common to add overlays and other features to mobile maps, however be careful not to obscure the map 'credits' information (such as the Google logo in iOS5) as doing so will result in rejection.

Manage Your Metadata

In addition to the obvious technical issues that can result in an application being rejected, there are some more subtle aspects of your submission that could result in rejection, especially around the metadata (description, keywords and marketing images) that you submit with your application for display within the App Store or Marketplace.

- ➔ Imagery – Follow the platform's guidelines for application icons and store pictures. Don't use trademarked images, we've seen apps get rejected because their icons featured a drawing of an iPhone!
- ➔ Trademarks – Avoid using any trademarks other than your own. Apps have been denied for mentioning trademarks in the app description or even in the keywords on Apple's App Store.
- ➔ Description – Do not use the word 'beta' or in any way indicate that the app is not ready for prime time. Don't mention other mobile platforms (even if your app is cross-platform). Most importantly, make sure the app does exactly what you say it does. If you list a bunch of features in your description, it had better be obvious how to use each of those features or you'll get a "feature mentioned in the application's description is not implemented" rejection.

Put as much effort into the application's metadata as into development and testing. Applications DO get rejected for minor infringements in the metadata so it is worthwhile taking the time to get it right.

App Stores: Not For Everyone

The primary focus of the stores on each platform is consumer distribution - the ability to reach as many customers as possible. However not all applications are targeted at consumers, there is a rapidly growing base of in-house and extranet-like applications that require limited distribution to employees, suppliers or customers. These apps aren't "for sale" and don't need approval, since the developer controls distribution to a closed group of users. Support for this type of deployment varies by platform.

Android offers the most flexibility in this regard: applications can be installed directly from a URL or email attachment (so long as the device's configuration allows it). This means it is trivial to create and distribute in-house corporate applications or publish applications to specific customers or suppliers.

Apple provides an in-house deployment option to developers enrolled in the iOS Developer Enterprise Program, which bypasses the App Store approval process and allows companies to distribute in-house apps to their employees. Unfortunately this license does not address the need for extranet-like app distribution to other closed groups of customers or suppliers. Enterprise (and Ad-Hoc) Deployment

Microsoft has no support for distribution outside the Marketplace, and have attempted to address the enterprise requirement by allowing apps to be published without an official listing (effectively "hiding" them from consumers, unless you know the app's unique identifier). This is clearly a stop-gap measure, and they're expected to significantly boost their enterprise deployment capabilities for Windows Phone 8.

App Store Summary

The review process can be daunting, but like the rest of the development lifecycle you can help ensure success with some planning and attention to detail. It all comes down to a few simple steps: read and understand the user interface guidelines you must adhere to, follow the rules if you are implementing platform-specific features, test thoroughly (then test some more) and finally make sure your application metadata is correct before you submit.

One last word of advice to developers publishing on Google Play: the lack of approval process may seem like it makes your job easier - but your customers will be even more demanding than a review team. Follow these guidelines as though your app could get rejected, otherwise it will be your customers doing the rejecting.

Special Considerations

Things to Avoid

CODE GENERATION

Generating code dynamically with System.Reflection.Emit should be avoided in shared code. Although it is supported on some platforms (such as Android) it will not work across all products.

Apple's kernel prevents dynamic code generation on iPhone and iPad, therefore emitting code on-the-fly will not work in MonoTouch. Likewise the Dynamic Language Runtime features cannot be used with Xamarin tools.

Some reflection features do work (eg. MonoTouch.Dialog uses it for the ReflectionAPI), just not code generation.

LOWEST-COMMON-DENOMINATOR USER INTERFACES

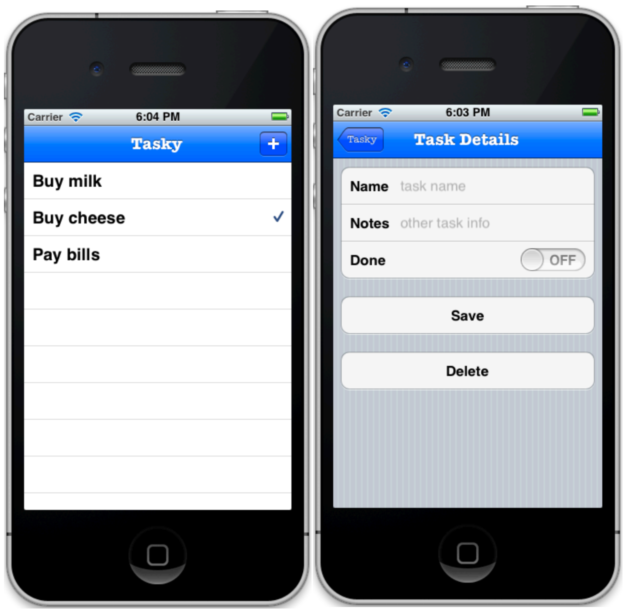
One of Xamarin's strengths is the ability to build a native UI on each platform. Avoid re-using the user-interface design across iOS, Android and Windows Phone without tweaking it for each platform. For example, don't build a 'Back' button into your Android or Windows Phone applications, and don't build a custom tabbed-control for iOS where the tabs sit across the top of the screen. Use the metaphors that the users on each platform are familiar with.

Cross-Platform Mobile Application Case Studies

The principles outlined in this document are put into practice in two sample applications that are available for download: *Tasky Pro* and the *MWC 2012* conference app.

Tasky Pro

Tasky is a simple to-do list application for iOS, Android and Windows Phone. It demonstrates the basics of creating a cross-platform application with Xamarin and uses a local SQLite database.



Read the Tasky Pro Case Study.

MWC 2012

MWC 2012 is a conference schedule application created for the Mobile World Congress in Barcelona, February 2012.

It is a sophisticated application that includes SQLite database support, remote data download, image caching, separate UI for iPad and iPhone as well as support for Android and Windows Phone. Platform-specific features like Live Tiles on Windows Phone are included.

Read the MWC 2012 Case Study.

Summary

This document has introduced Xamarin's application development tools and discussed how to build applications that target multiple mobile platforms.

It covered a layered architecture that structures code for re-use across multiple platforms, and describes different software patterns that can be used within that architecture.

Examples are given of common application functions (like file and network operations) and how they can be built in a cross-platform way.

Finally it briefly discusses testing, and provides references to the case studies that put these principles into action.