pariveda-color-horizontal.emf

Submitted: 11/26/2014

Pariveda Solutions, Inc.

2811 McKinney Avenue, Suite 220

Dallas, TX 75204

TEchnical Recommendations for

Compass Professional Health SErvices

Xamarin Development Best Practices

Contents

1 Overview 3

2 Assumptions 3

3 Glossary 3

4 A Great Xamarin app 3

5 Sample Xamarin App 3

6 Xamarin App Architecture 4

7 Xamarin Solution Organization 4

7.1 Best Practices in Project Organization 4

7.2 Notes on Sample Xamarin App 4

8 Xamarin App Architectural Organization 5

8.1 Xamarin App Layers 6

8.1.1 Shared Project 6

8.2 Third Party Libraries 13

8.3 Native Library Bindings 13

9 Security for Xamarin Apps 13

10 Exception Handling & Error Logging 14

10.1 Exception Handling 14

10.2 Error Logging 15

11 Unit and UI Testing 15

11.1 Unit Testing 15

11.1.1 Best Practices for Unit Testing 15

11.2 Automated UI Acceptance Testing 16

11.2.1 Best Practices for Automated UI Acceptance Tests 16

11.2.2 Resources 17

12 Continuous Integration 17

13 Debugging Xamarin Apps 18

14 Instrumentation 18

15 App Distribution 19

# Overview

This document provides a summary of Xamarin development best practices. It covers topics including code organization, code sharing strategies, web services integration, security best practices, native library bindings, continuous integration, app optimizations, instrumentation, and UI design tips.

# Assumptions

While writing the document, the following assumptions were made:

iOS and Android app development is emphasized over Windows Phone development. However, much of what is discussed here applies to Windows Phone apps as well.

Web service interaction is RESTful.

HIPAA compliance is paramount to successful Compass mobile apps.

Xamarin Studio is leveraged to build apps. However, Visual Studio can be used interchangeably with it.

# Glossary

Portable Class Library (PCL): a special type of .NET binary that can target multiple and dispersed Common Language Infrastructure (CLI) platforms including Xamarin.iOS, Xamarin.Android, Windows Phone, and Silverlight. Depending on the platforms being targeted, a varying but limited subset of the .NET framework would be available.

UI: User interface.

UX: User experience.

# A Great Xamarin app

This development guide is intended to help you create the best Xamarin apps possible. The following sections will go into details on how to achieve that. However, let’s start by looking at what great Xamarin apps looks like:

Native apps that look, feel, and act native in all areas UI, UX, functionality, and performance.

Apps that maximize code and UI sharing across iOS, Android, and Windows Phone so reasonable cross-platform app development costs are reigned in the short and long terms.

Apps that provide rich analytics so Compass can make informed decisions about their products and service offerings.

Apps that allow developers to gain confidence in their releases by incorporating continuous testing and integration strategies into their development lifecycle.

Apps that are easy to build and distribute.

Let’s discuss how we can achieve the above with Xamarin.

# Sample Xamarin App

An example Xamarin (iOS & Android) project was created to showcase the best practices discussed in this document. This project features simple iOS and Android apps that have a login screen and two list screens with one drilling into the other. The first list screen lists European soccer seasons. Upon clicking on a season, corresponding soccer teams are listed. It is highly recommended that you study the sample code as you read this document. It can be found at <https://github.com/CompassPHS/Xamarin-App-Template>. This document pulls screenshots and examples from this project when explaining the implementation details of the concepts discussed.

# Xamarin App Architecture

|  |  |
| --- | --- |
| Native iOS, Android, and Windows Phone apps are developed with native UIs but leverage shared codebases.  Platform specific code and UI design and logic is unique to each platform. The platform specific logic leverages the core library logic to accomplish many of its requirements.  Web service integration, database interaction, and other business logic are shared across the different platforms using a shared project.  A good Xamarin app optimizes for size of the shared code while keeping maintainability in mind. | **Figure 1:** Xamarin App Architecture |
|  |  |

# Xamarin Solution Organization

Xamarin apps are organized using Visual Studio solutions. Xamarin apps can be developed in either Visual Studio (with Xamarin plugins) or through Xamarin Studio. Developers can gracefully switch back and forth between the two during the development process. This is especially necessary if you are developing Windows Phone apps, as these cannot be developed in Xamarin Studio.

As shown in **Figure 1**, Xamarin apps are broken down into various components some that are unique to platforms and others that are shared across these platforms. Each of the components shown in **Figure 1** can be organized into its own project in Xamarin Studio. In order to support iOS, Android, and Windows Phone while also sharing code between them, four projects would need to be created; three would be platform-specific and one would be a Portable Class Library (PCL) that can be used by all three. While this is the minimum number of projects needed to create a basic app, additional projects should be created to cover unit and UI testing. Proper unit testing would require developers to create unit testing projects corresponding to every project of the four above. Additionally, a UI testing project is needed. A single UI testing project may be leveraged to test the three platforms using the Xamarin.UITests framework. We’ll discuss continuous integration, unit testing, and UI testing later in this document.

## Best Practices in Project Organization

Separate OS-specific code into Xamarin Studio projects, all under the same solution.

Create a shared project (as a Portable Class Library) to include common logic that can be re-used from the OS-specific projects.

Unit test your code by adding unit test projects that correspond to the projects described above. For each OS-specific and shared code projects create a corresponding unit testing project.

Increase code reuse. By leveraging a combination of portable class libraries and shared asset projects, a significant piece of your solution’s code can be shared. For instance, in our sample project, we were able to share our business logic, data access, data layer, and services layer across iOS and Android using a PCL while we used a Shared Assets Project (UnitTests.Shared) to share our unit tests across 3 other unit testing projects.

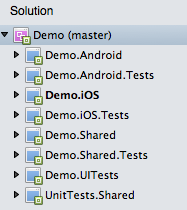
Follow a consistent naming convention for your projects. In the above project, we append “.iOS” for iOS specific projects, and “Android” for Android specific projects. We use “.Shared” to indicate projects that can be shared between OS-specific projects. Additionally, anytime we append “.Tests” or “.UITests” we indicate unit test and UI test projects, respectively. You’re welcome to follow this convention or create your own.

Follow.NET naming conventions across your solution.

## Notes on Sample Xamarin App

Our sample project implements the best practices above by organizing the code into 8 projects in our Xamarin Studio solution:

**Demo.Android:** Android application project. It leverages Demo.Shared project for data access, business logic, and web services integration.



**Figure 2:** Sample App

**Demo.Android.Tests:** unit-testing project for Demo.Android project. In addition to being able to run Android specific unit tests (in the Android runtime), this project also includes references to UnitTests.Shared and runs those as part of its test suite as well.

**Demo.iOS:** iOS application project. It leverages Demo.Shared project for data access, business logic, and web services integration.

**Demo.iOS.Tests:** unit-testing project for Demo.iOS. Similar to Demo.Android.Tests, this project allows unit tests to run on the iOS runtime. It includes the tests in UnitTests.Shared in its test suite.

**Demo.Shared:** a Portable Class Library that can be shared across Xamarin.iOS and Xamarin.Android apps. It contains logic to persist data in an SQLite database, business logic, and logic to integrate with remote web services. Demo.Android and Demo.iOS reference this project.

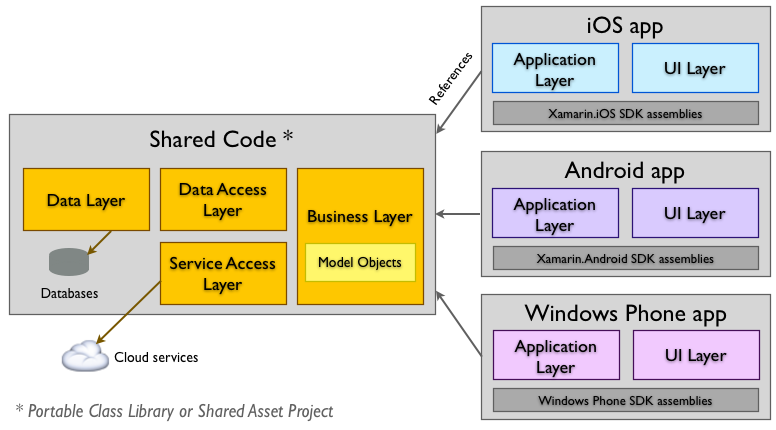
**Demo.Shared.Tests:** An nunit test project that contains logic to unit test the shared code. This project includes a mock web service to mimic the behavior of the real web service that would be used in a non-test environment. The project also leverages the UnitTests.Shared project to include the shared unit tests. This project allows you to unit test the shared code outside of any mobile environment (It will run on your Mac’s Mono/.NET runtime).

**Demo.UITests:** An nunit UI testing project to test iOS and Android apps. This project uses Calabash and Xamarin.UITest framework to test the native UIs created in Demo.Android and Demo.iOS. These tests can run locally or on Xamarin’s TestCloud service.

**UnitTests.Shared:** [A shared assets project](http://developer.xamarin.com/guides/cross-platform/application_fundamentals/shared_projects/) that includes nunit tests that can be shared across many projects. Currently, Demo.Android.Tests, Demo.iOS.Tests, and Demo.Shared.Tests all reference and include the shared unit tests in their projects. This allows for code reuse and better maintainability.

# Xamarin App Architectural Organization

As mentioned above, a Xamarin solution includes OS-specific, code sharing, unit testing, and UI testing projects. In this section, we’ll focus on the OS-specific and code sharing projects and leave testing projects for a later section. Let’s take a look at how we can architect a Xamarin solution to increase code-reuse and enhance maintainability:



**Figure 3:** Recommended Xamarin App Architecture

**Figure 3** above shows the distinction between shared projects and platform specific projects. All three platform specific projects reference the shared project and leverage it to access the database, cloud services, and run some business rules and logic on the data. The platform specific projects in turn handle platform specific logic and host the native UI.

## Xamarin App Layers

A single Xamarin app will typically be built using the following six layers:

**UI Layer:** contains native UI screens, UI logic, and bindings that connect both.

**Application Layer:** contains logic to call platform-specific APIs (ex: Camera, Touch ID, etc…) and application specific logic.

**Business Layer:** contains logic that exposes shared logic’s components using a Manager/Façade pattern. Includes business model definitions.

**Service Access Layer:** contains logic to interact with web services.

**Data Access Layer:** contains logic to interact with underlying persistent storage (ex: SQLite database).

**Data Layer:** hosts persistent store and related logic (ex: database caching logic).

Note that the first two layers will leave in platform specific projects, while the latter four reside inside the shared project.

While complex mobile apps may include all of the layers above, basic Xamarin apps may not. For example, an application that doesn’t interact with web services will not need a Service Access Layer. Additionally, the Data Access Layer may be already encapsulated into an ORM.

### Shared Project

#### Project Organization

To further help organize your solution, it is best to separate the various layers within your shared project into their own folder structures. Here’s a screenshot from our sample app and the following highlights the structure’s approach:

|  |  |
| --- | --- |
|  | * Business Layer, Data Access Layer, and Service Access Layer folders are used to host the respective logic. * Exceptions folder contains custom exceptions we use across the layers * Helpers contain helper classes that we use across the layers * References, Packages, Properties are all structures associated with a Visual Studio project. |

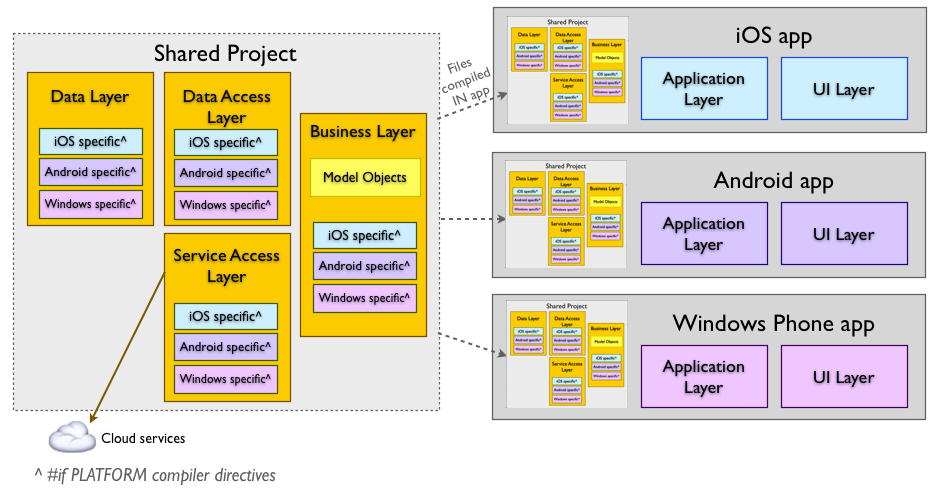
The above structure is the recommended one for organizing shared projects.

#### Shared Project Creation Strategies

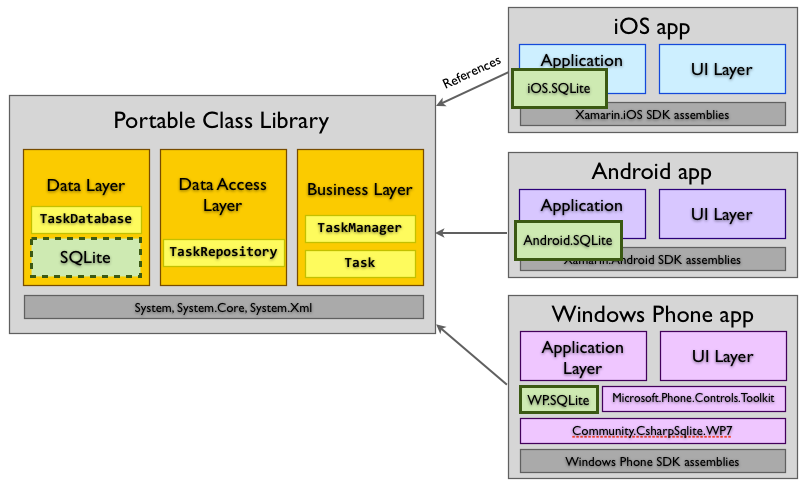
There are two strategies to create shared projects. Developers can either use a Shared Asset Project (SAP) or a Portable Class Library (PCL). SAP projects are easier to setup and use. They allow developers to use conditional logic to perform platform-specific logic. PCLs, on the other hand, are harder to setup and build. They allow developers to perform platform-specific logic using interfaces and dependency injection of concrete implementations of that logic. The following table and the two diagrams that follow help explain the difference between the two:

|  |  |  |
| --- | --- | --- |
| **Feature** | **Shared Asset Project (SAP)** | **Portable Class Library (PCL)** |
| **Ease of use** | Simple | Complicated |
| **Output** | Incorporated into referencing project’s binaries. The compiler literally copies the content of this project into the referencing project’s space | DLL |
| **Platform-specific code allowed?** | Yes, but compiler directives have to be used | No. Compiler directives cannot be used in a PCL |
| **Target platform and available frameworks** | Target is determined at the referencing project’s level. Available frameworks to use depend on referencing projects | Target is specific at the PCL project level. Can only target a subset of .NET |
| **Testability** | Can only be tested through referencing project binary since it doesn’t output own binary. | Can be tested separately by testing outputted DLL |
| **Maintainability** | Harder to maintain. Conditional logic for platform specific logic creates larger classes and fragmented codebases. Refactoring code inside ‘inactive’ logic is not possible in Visual Studio or Xamarin Studio. | Easier to maintain. Platform specific logic can be maintained at the platform-specific projects while interfaces are used to abstract contracts in the shared codebase |
| **Shareable?** | Not suited to share code with other developers | Binary can be distributed to other developers to use |
| **IDE** | Requires Visual Studio 2013 Update 2 or Xamarin Studio 5 and above | Visual Studio 2010 and above. Xamarin Studio 5 and above. Check <http://msdn.microsoft.com/en-us/library/gg597391%28v=vs.110%29.aspx> |

**Table 1:** Differences between an SAP and a PCL



**Figure 4:** SAP code is compiled inside parent project’s binary



**Figure 5:** PCL uses interfaces + DI to run platform-specific logic

#### Dealing with Multiple Platforms

There are several strategies that can be used to deal with multiple platforms. Take a look at the following link for additional information [“Dealing with Multiple Platforms”](http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_4_-_platform_divergence_abstraction_divergent_implementation/).

#### Best Practices for Sharing Projects

Use a PCL to share code between platform specific projects. PCLs allow your apps to be better maintainable:

Branching logic using interfaces rather than conditional statements allows for a cleaner design and better separation of concerns.

Unit testing your PCL by itself is possible because it outputs its own binary.

Conditional logic tends to create spaghetti code.

PCLs do not suffer from the ‘inactive’ code issue that SAPs suffer from. When it comes to refactoring large pieces of your code you would want your code to update references properly across your solution. Logic that is inactive due to conditional logic does not get updated upon refactoring. SAPs tend to have a lot of that logic.

#### Notes on Sample Xamarin App’s usage of Shared Projects

The sample app contains two projects that are shared. One is a PCL and the other is an SAP. The PCL project is the project that contains all business logic, data access, service access, and data logic. The SAP project contains unit tests that are being shared as-is across three other unit-testing projects. With this approach, our unit tests are shared to run inside the iOS, Android, and .NET runtimes. This allows us to share unit tests across the various architectures we are targeting.

#### Dependency Injection

As recommended above, use interfaces to abstract your branching platform logic. Upon app launch, instruct the runtime to inject a concrete implementation of the interface into the shared logic based on the runtime’s platform. Use [‘Unity’](https://www.nuget.org/packages/Unity/) library for dependency injection. Unity is one of the few libraries that work with PCL’s, Xamarin.iOS, and Xamarin.Android. This means that it can be leveraged from PCLs and platform specific projects as well. The sample Xamarin app provided with this documentation leverages Unity and should be inspected for usage examples.

Note: For iOS projects, add a reference to Microsoft’s [Common Service Locator](https://www.nuget.org/packages/CommonServiceLocator/). This is required for iOS apps, particularly when deploying to an iPhone.

#### Shared Project Layers

A shared project includes the four layers: business layer, data access layer, service access layer, and the data layer. For convenience, **Figure 6** is attached again to highlight the recommended structure for a Xamarin shared project.

##### 

**Figure 6:** recommended Xamarin shared project structure. Taken from sample app.

The sections below discuss each of the above layers:

##### Data Layer

###### Text Data

To persist data on iOS and Android Compass should leverage SQLite as its database engine. However, the vanilla SQLite is not HIPAA compliant, as it does not support encryption for data at rest by default. SQLCipher should be leveraged for HIPAA-compliance. SQLCipher is a commercial product that adds a transparent data encryption layer (AES 256) on top of vanilla SQLite. SQLCipher also comes with a basic C# ORM framework.

SQLCipher is developed and supported by Zetetic. Additional information about it can be found [here](https://www.zetetic.net/sqlcipher/).

###### Binary Data

To persist binary data within a PCL, use [PCL Storage](http://www.nuget.org/packages/pclstorage). Alternatively, System.IO namespace may be used from iOS, Android, or Windows.

##### Data Access Layer

The data access layer should contain logic that allows code to access the database to create, read, update, and delete (CRUD) data from/to it. SQLCipher makes data access pretty straightforward with the ORM framework it includes. The framework allows developers to model their database tables using plain old C# classes. It does so using attributes that developers can declaratively use to decorate their C# classes. Access to the database is powered using SQLCipher’s LINQ provider. While developers should stick to using LINQ for data access, the framework does allow them to run SQL queries directly on the database.

SQLCipher supports iOS, Android, and Windows Phone. Xamarin’s Component Store has iOS and Android components that could be purchased and incorporated into Xamarin apps. The Windows Phone libraries are not available on the Component Store and should be purchased directly from [Zetetic’s website](https://www.zetetic.net/sqlcipher/). Every SQLCipher library - whether obtained from Xamarin’s Component Store or through Zetelic - is linked against an OS-specific architecture. Since OS-specific logic cannot be placed inside our shared project (a PCL), an additional layer of abstraction would need to be introduced to solve this problem. An interface that establishes the CRUD contract is introduced in the shared project with OS-specific implementations for it in each corresponding OS-specific project. Upon loading the mobile application, a concrete implementation of the CRUD interface is injected into the shared code. The shared code proceeds to interact with the OS-specific database using the interface. In our sample project “Xamarin App Template”, our CRUD interface is named “ISecureDatabase”. The iOS and Android projects both contain implementations for that interface that are being injected into the shared project upon application launch.

If additional data access functionality beyond the basic CRUD operations furnished by the ORM is needed, developers may create additional data access layer classes to satisfy those needs. In our sample project, we’ve added additional classes, as we needed more support from our data access layer. Our data access layer is organized under the shared project’s “Data Access Layer” folder. Each class is domain-specific and contains data access logic relating to a certain model. The one exception to this rule is “SharedDal”. “SharedDal” is responsible for common functionality shared across all data access classes.

###### Best Practices for Data Access Layer

Separate domain specific logic into unique C# classes.

Share common code across the domain specific classes using a helper or base class.

Limit data access logic to CRUD operations.

Stub or mock data access layer/database for unit tests.

##### Service Access Layer

The Service Access Layer manages our application’s interaction with web services. All logic pertaining to interacting with backend web services should be placed in this layer. Just like our Data Access Layer, our Service Layer code is organized into domain-specific classes. The layer may also contain a shared codebase or helper classes to share common functionalities across its domain specific logic.

In our template project, our Service Layer code is organized into the “Service Access Layer” folder. We have a shared class that provides for some common functionality across all of the layer’s classes. Additionally, we have some domain-specific classes like “SeasonSal” and “TeamSal” that are tailored to fit our app’s web services needs for Seasons and Teams.

For unit testing purposes, you may want to mock your web services. In order to do that, any service layer class you wish to test (you should aim for 100% coverage) should implement an interface. With our service layer relying on interfaces, we can use a mocking framework and a dependency injection container to inject our mock services before running our unit tests. Our template project’s shared service access class implements the “IServiceAccessLayer” interface. With that, we are able to mock this interface in our unit-testing project and inject it at runtime just before running any tests.

###### Best Practices for Service Access Layer

Separate domain specific logic into unique C# classes.

Share common code across the domain specific classes using a helper or base class.

Limit logic to interacting with web services, serializing their requests, and de-serializing their responses.

Mock/stub web services for unit tests.

Use either WebClient or HttpWebRequest classes from a PCL to call your web services. WebClient is the higher-level construct.

##### Business Layer

The business layer encapsulates all the business logic for the application. It contains all the model objects in addition to business logic classes. The business logic classes are responsible for implementing business rules and managing connections between disparate data sources (i.e. local database vs. remote web service).

###### Best Practices for Business Layer

Separate domain specific logic into unique C# classes.

Share common code across the domain specific classes using a helper or base class.

The business logic should be limited to enforcing business rules and processes. It is responsible for servicing the presentation layer and coordinating interaction between lower layers such as data access and service access layers.

#### Platform-Specific Projects

As noted earlier, platform-specific projects should only contain UI and application layer code and should reuse code that’s been placed in the shared project.

##### Project Organization

The best way to explain our recommended organization strategy for platform-specific projects is by showing you what we’ve done in our sample app project. Since the sample app supports Android and iOS, two platform-specific projects are created. Traditionally, iOS and Android apps are organized and structured differently from each other in the native world. That organization has also carried over to Xamarin apps. However, having two or more inconsistent apps structures is not ideal for context switching (when developers are developing both apps) and for maintainability (especially for non-native developers). Because of that, we recommend the following structure for your iOS and Android apps. This creates a more consistent code base across your solution.

|  |  |
| --- | --- |
| **iOS** | **Android** |
|  |  |

As shown above, iOS and Android apps can be organized similarly. Using ‘References’, ‘Components’, and ‘Packages’ developers can manage and references external binaries and sibling projects. The ‘Application Layer’ folder hosts logic that deals with OS-specific functionality. ‘Controllers’ contains the controllers that drive our mobile app’s screens. ‘Resources’ contains any media files that are used across the app. The paragraphs below explain the purpose of each of the sections above with additional details.

###### Application Layer

The folder is intended to host any application and platform specific logic. See ‘Application Layer’ section below for additional information.

###### Assets

The ‘Assets’ folder is part of the standard structure of Android apps. Developers can use it to store raw assets such as text, CSV, and font files. Images should be stored under ‘Resources’ folder. Check that section below for more details.

###### Controllers

‘Controllers’ is used to group UI View Controllers in iOS and Activities in Android. View Controllers and Activities manage the relationship between the UI’s views and the application’s data.

###### Properties

Files residing in this folder define permissions, version information, supported orientations, etc. for the application. On Android, this folder would contain the Android Manifest file. On iOS, it would include the Entitlements.plist file.

###### Resources

The ‘Resources’ folder is used to organize images, UI layout files, and other media files you use in your app. Android and iOS have different Resources folder structure. In Android, you’d find the default ‘[drawable’, ‘layout’, and ‘values](https://developer.android.com/tools/projects/)’ folders. In iOS, those subdirectories are not needed by default. On iOS your images must reside directly under the ‘Resources’ folder and not contained inside any subdirectories. This allows for easier integration with Xcode’s Interface Builder.

Layout files are stored under the ‘layout folder’. On Android, that would include xml layout files. When designing iOS, if you decide to integrate with Xcode when designing your UI your .storyboard or .xib files must reside at the top-level within your project directory. This helps it detect your .storyboard file when editing using Apple’s Interface Builder.

**Note on images:** Android and iOS are very strict in how they handle image files. For Android, it is necessary to create ldpi, mdpi, hdpi, xhdpi, xxdpi, and xxxdpi subdirectories inside ‘drawable’ that contain the same image in different sizes depending on the desired dpi support level. For iOS, images require a specific naming convention (ex: .png, @2x.png, and @3x.png) in order to distinguish between resolutions. These should be placed under the ‘media’ folder.

###### Utils

Each platform is different and may require some utility classes to simplify code and increase reusability. Such classes can reside in this directory.

###### UI Helpers

It’s best practice to subclass certain views (ex: tableview/listview cells) to limit the size and scope of your Controllers/Activities. These UI-related subclasses can exist here for easy reference.

##### Platform Specific Project Layers

###### Application Layer

The application layer has two primary responsibilities:

1. Access to platform-specific features.
   1. For example, if camera access is required on iOS or Android, the corresponding application layer code can call that platform’s Camera API to access the camera.
   2. The other major example that stands out in our sample project is using this layer to provide a concrete implementation of the data access logic required to interact with SQLCipher on Android and iOS. These libraries have been linked against OS-specific architectures and couldn’t have been used in the shared code (a PCL) without abstracting the contract to a data access interface that we inject an implementation into based on the runtime. Those implementations belong to the application layer for each OS-specific project.
2. Load the application and initialize its settings. For example, our applications create a dependency injection container and load concrete implementations into it upon loading.

###### UI Layer

The UI Layer hosts the app’s screens, custom controls, controllers that drive the screens, and validation logic for these screens.

Developing Screens

In Xamarin, there are two ways to develop screens. One is to use Xamarin Forms, which is a new framework that allows you to build cross-platform native UIs. Xamarin Forms allows you to build the UI once and run it on each platform. The other is using the platform’s SDK for building screens. Xamarin Forms is a brand new technology from Xamarin and still has limited functionality. It can be used to build basic user interfaces and prototypes. Building native UIs using the platform’s SDK is currently the preferred way to go when it comes to building feature-rich screens, especially for consumer facing apps.

**Visual Designers**

iOS: Xamarin's iOS Designer for Xamarin Studio and Visual Studio facilitates building Views using drag-and-drop functionality and property fields. Collectively these Views make up a Storyboard, and can be accessed in the .STORYBOARD file that is 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.

Best Practices for Developing Screens

Use Xamarin’s iOS and Android UI Designers to design iOS and Android screens. This helps you avoid jumping back and forth to Xcode’s interface builder or Android Studio’s UI designer and Xamarin Studio. This also help developers design more efficiently as it takes Xamarin Studio some time after you save an Xcode project to sync up your UI changes and create bindings for your UI in the C# code. Use Visual Studio’s UI Designer for Windows Phone development.

Designing for Android requires understanding of developing for multiple screens. It’s highly recommended to read this [article](http://developer.android.com/training/multiscreen/index.html).

Follow the UI design guidelines for each platform: [iOS](https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/), [Android](https://developer.android.com/design/index.html), [Windows Phone](http://msdn.microsoft.com/library/windows/apps/fa00461b-abe1-41d1-be87-0b0fe3d3389d(v=vs.105).aspx).

## Third Party Libraries

Just like any other .NET apps, Xamarin apps can reference third-party apps using Xamarin Studio / Visual Studio references. In addition to having the ability to manually add and use DLL’s and other binaries in your solution, Xamarin Studio comes with two more ways to manage third-party libraries. First, developers can still work with NuGet to manage packages. Using NuGet, developers can search and find many third-party packages to use, especially for Portable Class Library projects. Second, Xamarin Studio comes with access to Xamarin’s Component Store. Think of Xamarin Component Store as a NuGet for everything Xamarin. It hosts many components that can be used in a variety of Xamarin apps. For more information on that, please visit their [website](http://components.xamarin.com). If none of the third-party libraries available through NuGet or Xamarin Component Store fit your needs, you can create wrappers around your favorite libraries (written in Objective-C or Java) and use them in Xamarin apps. See the following section for additional information.

## Native Library Bindings

Xamarin is a powerful platform that exposes a large subset of .NET to developers in addition to fully covering the native SDKs’ APIs for iOS and Android. This empowers developers by allowing them to use C# to accomplish what native developers can in Objective-C/Swift and Java. While wrapping these APIs in .NET is great for developers, its reach does not extend to third party libraries that were developed in Objective-C or Java. However, Xamarin does allow developers to use native libraries developed in these languages by creating bindings for them in C#.

The following Xamarin articles cover how to create these bindings for [iOS](http://developer.xamarin.com/guides/ios/advanced_topics/binding_objective-c/binding_objc_libs/) and [Android](http://developer.xamarin.com/guides/android/advanced_topics/java_integration_overview/binding_a_java_library_(.jar)/). For an example proof of concept on native bindings, we have created a [sample project](https://github.com/CompassPHS/Mobile-Xamarin-Native-Library-Support) for Compass that wraps the native Objective-C and Java libraries for Flurry, a mobile analytics web service. The project also includes integration with Flurry’s Windows Phone library.

This discussion does not apply to Windows Phone libraries. This is because Windows Phone and its libraries are already in .NET and need no wrapping.

Before building any wrappers and bindings for third-party libraries, please make sure that these libraries have not been wrapped already! Check [NuGet](http://www.nuget.org/packages) and [Xamarin’s Component Store](http://components.xamarin.com/).

# Security for Xamarin Apps

Xamarin, .NET, iOS, Android, and Windows Phone SDKs provide a wide variety of tools to help you enforce a security strategy. Below are the main security practices that we recommend you incorporate into your development process.

|  |  |  |
| --- | --- | --- |
| **Security Practice** | **How does it help secure app?** | **Implementation** |
| Create security strategy | Ensures consistent and standardized implementation of security policy | Standardize best practices in security  Standardize security techniques and tools across developers  Ex:  Use AES256 for encrypting data  Use high-entropy encryption keys. Store keys in ‘trusted’ locations |
| Encrypt all outgoing and incoming traffic | Allows app to securely communicate with external services, such as backend web services | All platforms - WebClient and HttpWebRequest support HTTPS  Non-HTTP traffic may need manual app level encryption |
| Encrypt all data stored in a persistent store (eg. Database, filesystem) | Allows app to secure user data from unauthorized access | All platforms – SQLCipher for transparent data encryption on top of SQLite |
| Store user credentials, encryption keys securely | Encryption keys and user credentials are usually the weakest link for your app’s security. Avoiding hardcoding keys in the code and storing credentials in clear text in the app enhances the security of your app | iOS – use Keychain services  Android - use Android Keystore Provider (>= API 18).  Windows Phone – Use isolated storage and app level encryption. [Tutorial](http://msdn.microsoft.com/library/windows/apps/hh487164(v=vs.105).aspx). |
| Avoid leakage of sensitive data | Ensure your users’ privacy | Avoid communicating sensitive information outside of the app. For example in push notifications, emails, or text messages |
| Sanitize input from external sources to your app | Ensures that malicious input is not used | Push notifications or other needs might require you to register your app with a custom URI handler so these can automatically open your app. Validate the input that is passed from opening your app using custom URI handlers |
| Restrict app access to authorized and authenticated | Ensure only authorized users are able to access app. Ensure app access is restricted in duration | Force users to logon with their secure credentials to access privileged data  Force automatic logoff based on your security strategy  Enforce multi-factor authentication where needed |
| Audit | Ensures legal compliance and better discoverability of system vulnerabilities | Audit failed and successful login attempts  Audit user access to sensitive areas |

# Exception Handling & Error Logging

Proper exception handling in a Xamarin app can help you avoid application crashes, report issues for further investigation, and propagate proper alerts to the user. Below are our recommended practices for exception handling and error logging within a Xamarin app. This list was compiled for a Xamarin app with a shared code base (business logic, data access, and service access) with multiple platform specific projects. That said, most of the recommendations apply to all types of Xamarin apps:

## Exception Handling

**Do’s:**

Report execution failures by throwing exceptions

Create a custom base exception class for all custom exceptions to inherit from. This allows for known exceptions (i.e. exceptions generated by your application and not the system) to be easily caught

Create an exception type for each layer and incorporate it as part of the layer’s public contract. For example, a Business Layer would have its own BusinessLayerException (derived from your base exception class). Public methods in BL that need to communicate a failure upwards should only throw that exception. If additional granularity is needed subclass BusinessLayerException and communicate the level of granularity by throwing the different types of child exceptions.

Nest exceptions to maintain valuable debugging information

Log exceptions as errors in the top most layer that handles them without propagating them upwards. As these exceptions bubble in the deeper layers, log them as warnings..

**Don’ts:**

Use exceptions to control normal flow of control

Use error codes to communicate failure type

Use Message property to communicate important information

Return exceptions as the return value or as an out parameter for any method

Throw System.Exception

Catch System.Exception. Unhandled exceptions imply that an unknown event occurred. The application’s state is unstable/unknown and should force a restart so that it can obtain a known context before continuing. Error logging frameworks like Xamarin.Insights automatically logs these unhandled exceptions when the application resumes. Ensure that your error-logging framework of choice does that as well.

Throw or derive from System.ApplicationException

Swallow exceptions

Clear the stacktrace when re-throwing exceptions (throw e; vs throw;). Former clears the stack

## Error Logging

**Do’s**

Use varying log levels to indicate error severity

Catch exceptions at the highest level possible and log accordingly

Associate log statements with app user for better customer service

Track significant user activities if informed analytics is desired

Track asynchronous or long running operations’ time consumption to understand app bottlenecks

Keep intended audience in mind when designing log messages

**Don’ts**

Leak sensitive information in logs (ex: user credentials, HIPAA covered data)

Log the same error multiple times across your app layers

The provided sample application follows the best practices above and uses Xamarin Insights (<https://xamarin.com/insights>) for error logging and some analytics.

# Unit and UI Testing

## Unit Testing

Xamarin allows developers to unit test their code by incorporating NUnit unit tests into their solutions. Every project you wish to unit test should have a corresponding unit testing project created alongside it in the Xamarin Studio solution. Unit tests can target logic in any of the five layers we discussed earlier (Business, Data Access, Data, Service Access, Application). To test logic in the shared codebase (if PCL) a standard NUnit project can be used. However, if you wish to target logic that can only execute on a specific platform (ex: app logic in platform-specific projects), then special NUnit tests projects for iOS, Android, or Windows would need to be created. These platform specific test projects will execute the unit tests within on the specified runtime. The sample Xamarin app provided contains both types of projects. Take a look at Demo.Tests, Demo.iOS.Tests, and Demo.Android.Tests for details.

### Best Practices for Unit Testing

Use standard NUnit unit test projects for testing PCLs

Use platform specific unit test projects for testing platform specific projects and the features within them

Use standard naming conventions across the test projects. We recommend appending “.Tests” to your unit testing projects

Share as much code as possible between your unit test projects. Developers can use a Shared Asset Project to host a common set of unit tests that can be run across multiple runtimes. The platform specific and shared code unit test projects can all reference the same unit test shared assets projects. This allows you to run the shared tests in various runtimes to ensure total coverage. It also allows you to re-use code. The sample Xamarin app contains a Shared Asset Project called “UnitTests.Shared”. This project is referenced from Demo.Tests, Demo.iOS.Tests, and Demo.Android.Tests. Because of that, our three projects are able to share and run the tests inside UnitTests.Shared!

## Automated UI Acceptance Testing

Using Xamarin Test Cloud it is possible to automate UI acceptance testing and test your app on hundreds of real devices. Currently, only iOS and Android apps are supported. Xamarin Test Cloud provides developers with two ways to write tests, either using Calabash (ruby) or Xamarin.UITest (C#). We recommend using Xamarin.UITest (C#) to maintain development environment consistency and minimize maintenance costs (by avoiding investing in ruby resources). This section focuses on using Xamarin.UITest.

We recommend using a single UI testing project to test your iOS and Android Xamarin apps. This allows for better maintainability and sharing of UI tests across the two platforms. This is possible because the tests across the two platforms are essentially testing the same behavior. The difference between the two resides in the way each platform uses to lookup views/elements on its screens to validate them. That part can be abstracted through an interface, with concrete implementations for each platform. With that, we are able to write a single set of UI tests to target both platforms.

### Best Practices for Automated UI Acceptance Tests

Use Xamarin.UITest framework

Use a single Xamarin Studio project to host all UI Acceptance Tests. This simplifies your project structure and enhances maintainability as you share UI tests across the two platforms, iOS and Android.

Separate your UI tests into classes based on business domain

Use the same set of tests to run tests on both platforms. This can be achieved by abstracting away how each platform looks up elements on its screen.

When an unexpected event occurs (ex: an exception is generated) while running a UI test, capture the device’s screenshot for better debugging.

Specifying screenshot names is not currently possible and your screenshot will always default to the same file name. Because of that, every screenshot you take will overwrite the one before. To avoid that, ensure that your code renames the screenshots you take after every test.

Here’s an example of a UI test from our sample Xamarin app:



**The UI test above has several interesting features:**

It is a standard NUnit test

A try catch block surrounds its logic. The catch block takes a screenshot of the screen upon error. Since this is an automated UI test, capturing a screen shot upon failure could help developers efficiently address the issue.

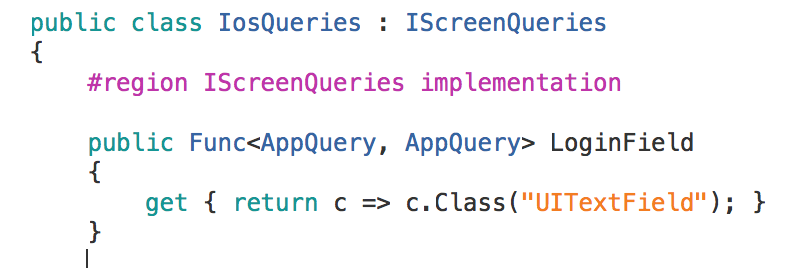
\_app is refers to the currently running application. On various occasions we are instructing the app to enter some text into a text field or tap a certain button.

\_queries is the abstract interface referred to earlier. It abstracts how each platform lookups up UI elements on its screen.

Because \_app and \_queries are abstracted and will work on each of the platforms, this test can be run on either iOS or Android.

Inside the sample app’s “Helpers” folder, there are three classes that we recommend you look at: IScreenQueries, IosQueries, AndroidQueries. The latter two files are implementations of the IScreenQueries interface. The IScreenQueries interface specifies the needed contract to look certain elements up from a device’s screen as required in our tests. The implementation classes specify how that’s done in each platform.

Here’s an example implementation for iOS:



The above snippet has query method that returns a view from the iOS’s screen. This method is essentially returning an object representing a login text field box. The developer looked that field up using a class selector. This is possible because that screen has only one view with class “UITextField”.

Xamarin.UITest comes with a REPL environment that allows developers to interact with the screen while the application is running so they can understand how each element is referenced on the screen.

Note that while the above class selector works for this screen, it may not work for more complicated screens that have more than one textfield on it. Because of that a more unique way of identifying the views is required. The REPL can be used to achieve just that. Refer to [this article](http://developer.xamarin.com/guides/testcloud/uitest/intro-to-uitest/) to understand what selectors are available and how to use REPL. Look for “Using the REPL” section.

### Resources

For resources on setting up Xamarin UI Test projects, refer to Xamarin’s website:

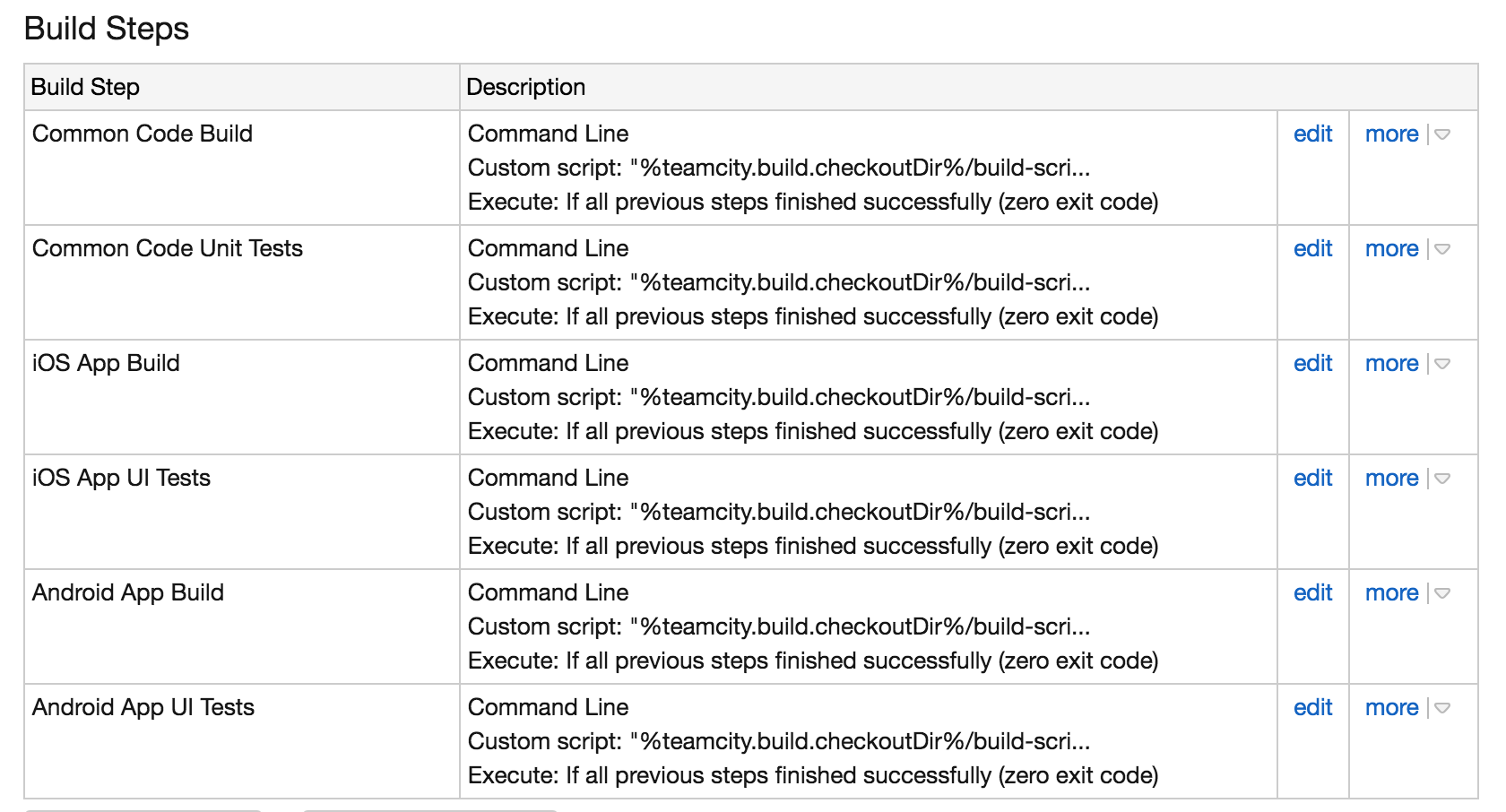
1. Introduction to Test Cloud: <http://developer.xamarin.com/guides/testcloud/introduction-to-test-cloud/>
2. Using Xamarin.UITest: <http://developer.xamarin.com/guides/testcloud/uitest/>
3. Submitting tests to Test Cloud <http://developer.xamarin.com/guides/testcloud/submitting/>

# Continuous Integration

We understand that Compass uses TeamCity for continuous integration. It is possible to setup Xamarin apps for CI to automate building, unit testing, UI testing, and archiving successful builds. It is also possible to setup build scripts to automate packaging the applications in preparation for app submission to the app stores.

TeamCity can be setup with a Xamarin ‘Build Configuration Template’ that can be used with all Xamarin apps. Your CI process can be setup such that this template can work with virtually all apps. This can be done if the template’s build steps are configured to call external command line scripts to build shared project, then iOS, then Android for example. This delegation of responsibility allows you to say something like “build iOS” or “build Android”. Meanwhile, your external build scripts can handle the details of building these apps. However, with this setup, you’re abstracting away the build steps from the build implementation.

This is exactly what we did for Compass. Your TeamCity server is currently setup with a Xamarin template that has 6 build steps:



The build steps above execute external build scripts to build your solution. Here’s an example from the ‘Common Code Build’ step:

"%teamcity.build.checkoutDir%/build-scripts/build.sh" "common-build"

All the step does is call on build.sh script and passes “common-build” to it. Meanwhile build.sh uses F# Make (FAKE) library to build itself. Here’s an excerpt from that script file:



The rest of the steps follow a very similar pattern. Check out the sample app’s build-scripts folder for details into using [FAKE](http://fsharp.github.io/FAKE/) to build Xamarin apps.

For an excellent presentation on CI and Xamarin, refer to [this video](https://www.youtube.com/watch?v=Awl4vGo7Yj0) from Xamarin’s 2014 Evolve conference.

# Debugging Xamarin Apps

Xamarin Studio comes with a powerful debugger that allows you to test apps running on simulators or real devices. The debugger allows developers to specify breakpoints, step through the code, and watch values change during execution. Furthermore, it comes with an expression evaluator that allows you to execute various expressions (code) at breakpoints. Xamarin Studio’s debugger is on par with Visual Studio’s.

To complement Xamarin Studio’s debugger, developers may also use Console.WriteLine(). Xamarin apps will output Console.WriteLine() calls onto the ‘Application Output’ panel in Xamarin Studio.

# Instrumentation

Profiling Xamarin apps is as straightforward as profiling native apps. For iOS, developers should use Xcode’s Instruments. For Android, use the ‘Android Device Monitor’. For information on using Instruments, please refer to [Xamarin’s guide](http://developer.xamarin.com/guides/ios/deployment,_testing,_and_metrics/using_instruments_to_detect_native_leaks_using_markheap/) and [Apple’s Instruments guide](https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html). For additional information on using Android Device Monitor, please refer to the following Android resources: [using DDMS](https://developer.android.com/tools/debugging/ddms.html), [debugging memory](https://developer.android.com/tools/debugging/debugging-memory.html), and Xamarin’s section on [debugging Android apps](https://developer.xamarin.com/guides/android/deployment,_testing,_and_metrics/).

At Xamarin Evolve 2014, Xamarin alpha released [Xamarin Profiler](https://developer.xamarin.com/guides/cross-platform/deployment,_testing,_and_metrics/xamarin-profiler/). The profiler provides a GUI on top of [Mono log profiler](http://www.mono-project.com/docs/debug+profile/profile/profiler/) and adds support for profiling Xamarin iOS and Android apps on Mac and Windows machines. At the time of writing, the profiler was still in Alpha release. Keep an eye on Xamarin profiler as will help with better insight into the workings of your app from a Mono runtime perspective. This could be especially valuable for Android apps.

# App Distribution

Xamarin iOS and Android apps are compiled into their native iOS and Android binaries. Deploying and distributing Xamarin apps in their respective app stores is identical to distributing native apps, because Xamarin apps are native. Follow the guidelines set forth by [Apple](https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/Introduction/Introduction.html) and [Google](http://developer.android.com/distribute/googleplay/start.html) to distribute your apps successfully. Additionally, ensure that you follow Xamarin’s guidelines for [iOS](http://developer.xamarin.com/guides/ios/deployment,_testing,_and_metrics/app_distribution_overview/) and [Android](http://developer.xamarin.com/guides/android/deployment,_testing,_and_metrics/publishing_an_application/).