Introduction

Xamarin Official Curriculum

Overview

Building mobile applications can be as easy as opening up your IDE, throwing something together, doing a quick bit of testing, and then submitting your app to an App Store – all done in an afternoon. Or it can be a complex process that involves rigorous up-front design, usability testing, QA testing on thousands of devices, a full beta lifecycle, and then multi-stage deployment.

In this document, we're going to take a thorough introductory examination of building mobile applications, including:

- **Requirements** – We'll enumerate and examine the requirements for building for iOS and Android applications.
- **Tools** – Xamarin offers a complete set of tools for building mobile applications, including an Integrated Development Environment (IDE), an integrated designer, and a compiler. Additionally, Xamarin integrates with a number of third-party tools.
- **Process** – The process of software development is called the Software Development Lifecycle (SDLC). We'll examine all phases of the SDLC with respect to mobile application development, including: Inspiration, Design, Development, Stabilization, Deployment, and Maintenance.
- **Considerations** – There are a number of considerations when building mobile applications, especially in contrast to development for traditional web or desktop applications. We'll examine these considerations and how they affect mobile development.

This document answers fundamental questions about mobile app development, for new and experienced application developers alike. It takes a fairly comprehensive approach to introducing most of the concepts you'll run into during the entire Software Development Lifecycle (SDLC). However, this document may not be for everyone; if you're itching to just start building applications, we recommend jumping ahead to either the Hello, Android or Hello, iPhone tutorials, and then come back to this document later to fill in your knowledge gaps.

Requirements

Before we get too far into this process, a couple of important requirements should be mentioned. First, if you'd like to deploy to a device, rather than just to the iOS Simulator or Android Emulator, you'll need at least a professional version of Xamarin's products, which can be purchased at the Xamarin Store.

Second, if you'd like to develop for iOS, you must have an Apple Macintosh computer running at least OSX Lion. Although Xamarin applications are based on the .NET BCL and are written in C#, Xamarin requires Xcode and the iOS SDK in order to compile. Additionally, the iOS Device Simulator is part of the iOS SDK, and therefore only available on Mac. In order to download the iOS SDK, you must be a member of Apple's Developer Program. This program is free for simulator-

only development (you cannot deploy to your device or the App Store). To deploy beyond the simulator requires an upgraded membership that costs $99 USD/year.

All the tutorials presented here are based on the latest versions of Xamarin's software. Installation is covered in the next tutorial.

Introduction to Xamarin

When considering how to build iOS and Android applications, many people think that the indigenous languages, Objective-C and Java, respectively, are the only choices. However, over the past few years, an entire new ecosystem of platforms for building mobile applications has emerged. These new solutions include Xamarin, and HTML solutions such as PhoneGap and Appcelerator, etc., just to name a couple.

Xamarin is unique in this space by offering a single language (C#), a class library, and a runtime that work across all three mobile platforms of iOS, Android, and Windows Phone (Windows Phone's indigenous language is already C#), while still compiling native (non-interpreted) applications that are performant enough even for demanding games.

Each of these platforms has a different feature set and each varies in its ability to write native applications–that is, applications that compile down to native code and that interoperate fluently with the underlying Java subsystem. For example, some platforms only allow you to build apps in HTML and JavaScript (such as Appcelerator and PhoneGap), whereas some are very low level and only allow C/C++ code. Some platforms (such as Flash) don't even utilize the native control toolkit.

Xamarin is unique in that it combines all of the power of the indigenous platforms and adds a number of powerful features of its own, including:

- **Complete Binding for the Indigenous SDKs** – Xamarin contains bindings for nearly the entire underlying platform SDKs in both iOS and Android. Additionally, these bindings are strongly typed, which means that they're easy to navigate and use, and provide robust compile-time type checking and auto completion during development. This leads to fewer runtime errors and higher quality applications.
- **Objective-C, Java, C, and C++ Interop** – Xamarin provides facilities for directly invoking Objective-C, Java, C, and C++ libraries, giving you the power to use a wide array of 3rd party code that has already been created. This lets you take advantage of existing iOS and Android libraries written in Objective-C, Java, or C/C++. Additionally, Xamarin offers binding projects that allow you to easily bind native Objective-C and Java libraries by using a declarative syntax.
- **Modern Language Constructs** – Xamarin applications are written in C#, a modern language that includes significant improvements over Objective-C and Java such as *Dynamic Language Features*, *Functional*

Constructs such as *Lambdas*, *LINQ*, *Parallel Programming* features, sophisticated *Generics*, and more.

- **Amazing Base Class Library (BCL)** – Xamarin applications use the .NET BCL, a massive collection of classes that have comprehensive and streamlined features such as powerful XML, Database, Serialization, IO, String, and Networking support, just to name a few. Additionally, existing C# code can be compiled for use in your applications, which provides access to thousands upon thousands of libraries that will let you do things that aren't already covered in the BCL.
- **Modern Integrated Development Environment (IDE)** – Xamarin uses MonoDevelop on Mac OSX, and also MonoDevelop or Visual Studio 2010 on Windows. These are both modern IDE's that include features such as code auto completion, a sophisticated Project and Solution management system, a comprehensive project template library, integrated source control, and many other options.
- **Mobile Cross Platform Support** – Xamarin offers sophisticated cross-platform support for the three major mobile platforms of iOS, Android, and Windows Phone. Applications can be written to share up to 90% of their code, and our Xamarin.Mobile library offers a unified API to access common resources across all three platforms. This can significantly reduce both development costs and time to market for mobile developers that target the three most popular mobile platforms.

Because of Xamarin's powerful and comprehensive feature set, it fills a void for application developers that want to use a modern language and platform to develop cross-platform mobile applications.

> **Note:** This Getting Started series focuses on teaching you how to build iOS and Android applications. If you're interested in building for Windows Phone, Microsoft offers tutorials here. If you're interested in learning more about cross-platform development with Xamarin (including Windows Phone), you can find our guide here.

Let's take a look at how this all works.

How Does Xamarin Work?

Xamarin offers two commercial products, Xamarin.iOS and Xamarin.Android, also known as MonoTouch and Mono for Android, respectively. They're both built on top of *Mono*, an open-source version of the .NET Framework based on the published .NET ECMA standards. Mono has been around almost as long as the .NET framework itself, and runs on nearly every imaginable platform, including Linux, Unix, FreeBSD, and Mac OSX.

On iOS, Xamarin's *Ahead-of-Time* (*AOT*) compiler compiles Xamarin.iOS applications directly to native ARM assembly code. On Android, Xamarin's compiler compiles down to *Intermediate Language* (*IL*), which is then *Just-in-Time* (*JIT*) compiled to native assembly when the application launches.

In both cases, Xamarin applications utilize a runtime that automatically handles things such as memory allocation, garbage collection, underlying platform interoperability, etc.

MonoTouch.dll and Mono.Android.dll

Xamarin applications are built against a subset of the .NET BCL known as the Xamarin Mobile Profile. This profile has been created specifically for mobile applications and packaged in the MonoTouch.dll and Mono.Android.dll (for iOS and Android, respectively). This is much like the way Silverlight (and Moonlight) applications are built against the Silverlight/Moonlight .NET Profile. In fact, the Xamarin Mobile profile is equivalent to the Silverlight 4.0 profile with a bunch of BCL classes added back in.

For a full list of available assemblies and classes, see the MonoTouch Assembly List and the Mono for Android Assembly List.

In addition to the BCL, these .dlls include wrappers for nearly the entire iOS SDK and Android SDK. Availability of these libraries allows you to invoke the underlying SDK APIs directly from C#.

> **Note:** Xamarin applications are compiled against the Xamarin Mobile profile, just like Silverlight/Moonlight apps are compiled against theirs. This means that you cannot use off-the-shelf .NET assemblies without recompiling the C# source against the Xamarin Mobile profile.

Application Output

When Xamarin applications are compiled, the result is an Application Package, either an .app file in iOS, or an .apk file in Android. These files are indistinguishable from indigenous application packages and are deployable in the exact same way.

Mobile Development Considerations

While developing mobile applications isn't fundamentally different from traditional web/desktop development in terms of process or architecture, there are some considerations to be aware of.

Let's take a look at some common considerations and then we'll examine platform-specific issues.

Common Considerations

MULTITASKING

There are two significant challenges to multitasking (having multiple applications running at once) on a mobile device. First, given the limited screen real estate, it is difficult to display multiple applications simultaneously. Therefore, on mobile devices only one app can be in the foreground at one time. Second, having multiple applications open and performing tasks can quickly eat battery power.

Each platform handles multitasking differently. We'll explore that subject in a bit.

FORM FACTOR

Phones and tablets are the two major types of mobile devices. There a few crossover devices that fall in between these two categories. Developing for these form factors is generally very similar; however, designing applications for them can be very different. Phones have very limited screen space, and tablets, while bigger, are still mobile devices with less screen space than even most laptops. Because of this, mobile platform UI controls have been designed specifically to be effective on smaller form factors.

DEVICE AND OS FRAGMENTATION

It's important to take into account different devices throughout the entire software development lifecycle:

- ➔ Conceptualization and Planning – Because different devices can have different hardware and device features, you must keep in mind that an application that relies on certain features may not work properly on some devices. For example, not all devices have cameras, so if you're building a video-messaging application, some devices may be able to play videos, but not take them.
- ➔ Design – When designing an application's User Experience (UX), different screen ratios and sizes should be kept in mind. Additionally, when designing an application's User Interface (UI), different screen resolutions should be considered.
- ➔ Development – When using a feature from code, the presence of that feature should always be tested first. For example, before using a device feature, such as a camera, always query the OS for the presence of that feature first. Then, when initializing the feature/device, make sure to request currently supported features from the OS about that device and then use those configuration settings.
- ➔ Testing – It's incredibly important to test your application early and often on actual devices. On Android, even devices with the same hardware specs can vary widely in their behavior.

LIMTED RESOURCES

Mobile devices get more and more powerful all the time, but they are still mobile devices that have limited capabilities in comparison to desktop or notebook computers. For instance, desktop developers generally don't worry about memory capacities; they're used to having both physical and virtual memory in copious quantities, whereas on mobile devices you can quickly consume all available memory just by loading a handful of high-quality pictures.

Additionally, processor-intensive applications such as games or text recognition can really tax the mobile CPU and adversely affect device performance.

Because of considerations like these, it's important to code smartly and to deploy early and often to actual devices in order to validate responsiveness.

iOS Considerations

MULTITASKING

Multitasking is very tightly controlled in iOS, and there are a number of rules and behaviors that your application must conform to when another application comes to the foreground, otherwise your application will be terminated by iOS.

DEVICE-SPECIFIC RESOURCES

Within a particular form factor, hardware can vary greatly between different models. For instance, some devices have a rear-facing camera, some also have a front-facing camera, and some have no camera at all.

Because of these differences between device models, it's important to check for the presence of a feature before attempting to use it.

OS SPECIFIC CONSTRAINTS

In order to make sure that applications are responsive and secure, iOS enforces a number of rules that applications must abide by. In addition to the rules regarding multitasking, there are a number of event methods out of which your app must return in a certain amount of time, otherwise it will be terminated by iOS.

Also worth noting, apps run in what's known as a Sandbox, an environment that enforces security constraints that restrict what your app can access. For instance, an app can read from and write to its own directory, but if it attempts to write to another app directory, it will be terminated.

Android Considerations

MULTITASKING

Multitasking in Android has two components; the first is the activity lifecycle. Each screen in an Android application is represented by an Activity, and there is a specific set of events that occur when an application is placed in the background or comes to the foreground. Applications must adhere to this lifecycle in order to create responsive, well-behaved applications. For more information, see the Activity Lifecycle guide.

The second component to multitasking in Android is the use of Services. Services are long-running processes that exist independently of an application and are used to execute processes while the application is in the background. For more information, see the Creating Services guide.

MANY DEVICES + MANY FORM FACTORS

Unlike iOS, which has a small set of devices, or even Windows Phone 7, which only runs on approved devices that meet a minimum set of platform requirements, Google doesn't impose any limits on which devices can run the Android OS. This open paradigm results in a product environment populated by a myriad of different devices with very different hardware, screen resolutions and ratios, device features, and capabilities.

Because of the extreme fragmentation of the Android device market, most people choose the most popular 5 or 6 devices to design and test for, and prioritize those.

SECURITY CONSIDERATIONS

Applications in the Android OS all run under a distinct, isolated identity with limited permissions. By default, applications can do very little. For example, without special permissions, an application cannot send a text message, determine the phone state, or even access the Internet! In order to access these features, applications must specify in their application manifest file which permissions they would like, and when they're being installed. The OS reads those permissions, notifies the user that the application is requesting those permissions, and then allows the user to continue or cancel the installation. This is an essential step in the Android distribution model, because of the open application store model, since applications are not curated the way they are for iOS, for instance. For a list of application permissions, see the Manifest Permissions reference article in the Android Documentation.

Windows Phone Considerations

MULTITASKING

Multitasking in Windows Phone also has two parts: the lifecycle for pages and applications, and background processes. Each screen in an application is an instance of a Page class, which has events associated with being made active or inactive (with special rules for handling the inactive state, or being "tombstoned"). For more information, see the Execution Model Overview for Windows Phone documentation.

The second part of Windows Phone multitasking is the provision of background agents for processing tasks that run even when an application is not running in the foreground. More information on scheduling periodic tasks or creating resource-intensive background tasks can be found in the Background Agents Overview.

DEVICE CAPABILITIES

Although Windows Phone hardware is fairly homogeneous due to the strict guidelines provided by Microsoft, there are still optional components that require special attention while coding. Optional hardware capabilities include the camera, compass, and gyroscope. There is also a special class of low-memory (256MB) that requires careful consideration, or developers can choose to opt-out of low-memory support completely.

DATABASE

Both iOS and Android include the SQLite database engine that allows for sophisticated data storage that also works cross-platform. Windows Phone 7 did not include a database, and Windows Phone 7.1 includes a database engine that can only be queried with LINQ to SQL and does not support Transact-SQL queries. There is an open-source port of SQLite available that can be added to Windows Phone applications to provide familiar Transact-SQL support and cross-platform compatibility.

SECURITY CONSIDERATIONS

Windows Phone applications are run with a restricted set of permissions that isolates them from one another and limits the operations they can perform. Network access must be performed via specific APIs and inter-application communication can only be accomplished via controlled mechanisms. Access to the file system is also restricted; the Isolated Storage API provides key-value pair storage and the ability to create files and folders in a controlled fashion (refer to the Isolated Storage Overview for more information).

An application's access to hardware and operating system features is controlled by the capabilities listed in its manifest file (similar to Android). The manifest must declare the features required by the application, so that users can see and agree to those permissions and also so that the operating system allows access to the APIs. Applications must request access to features such as contacts or appointments data, camera, location, and media library. See Microsoft's Application Manifest File documentation for additional information.

Mobile Development SDLC

The lifecycle of mobile development is not very different from the SDLC for web or desktop applications. As with those development lifecycles, there are usually five major portions in the process:

- **Inception** – All apps start with an idea. That idea is usually refined into a solid basis for an application.
- **Design** – The app's User Experience (UX) is defined in the design phase. The UX consists of the basic characteristics of the general layout, how it works, etc., as well as the process of turning the UX into a proper User Interface (UI) design, usually with the help of a graphic designer.
- **Development** – Normally the most resource intensive phase, this is the actual building of the application.
- **Stabilization** – When development is far enough along, QA should begin to test the application and bugs are fixed. Frequently, an application will be put into a limited beta phase in which a wider user audience is given a chance to use it and provide feedback and inform changes.
- **Deployment** – After an application has been stabilized, it's typically published for wider use. This could be to a public app store, or it could be made available for a particular team in an organization.

Often many of the phases of the lifecycle overlap, for example, it's common for development to continue while the UI is being finalized, and this ongoing, late-stage development may even inform the UI design. Additionally, an application may be going into a stabilization phase at the same that new features are being added to a new version.

Furthermore, these phases can be used in any number of SDLC methodologies such as Agile, Spiral, Waterfall, etc.

Let's cover how each of these phases plays a part in Mobile Development.

Inception

Mobile devices are almost literally everywhere in the modern world. This ubiquity and the consequent level of interaction people have with mobile devices means that nearly everyone has an idea for a mobile app. That's probably because mobile devices open up a whole new way to interact with computing, the web, and even corporate infrastructure.

The inception stage is all about defining and refining the idea for an app. In order to create a successful app, it's important to first ask some fundamental questions. For example, if you're developing an app for distribution in a public app store, some considerations are:

- ➔ Competitive Advantage – Are there similar apps out there already? If so, how does this application differentiate itself from others?

If you intend to distribute the app in the enterprise:

- ➔ Infrastructure Integration – What existing infrastructure will it integrate with or extend?

Additionally, you should evaluate the usage of the app in a mobile form factor:

- ➔ Value – What value does this app bring users? How will they use it?
- ➔ Form/Mobility – How will this app work in a mobile form factor? How can I add value by using mobile technologies such as location awareness and the camera?

To help with designing the functionality of an app, it can be useful to define Actors and Use Cases. Actors are roles within an application and are often users. Use cases are typically actions or intents.

For instance, if you're building a task-tracking application, you might have two Actors: *User* and *Friend*. A User might *Create a Task*, and *Share a Task* with a Friend. In this case, creating a task and sharing a task are two distinct use cases that, in tandem with the Actors, will inform what screens you'll need to build, as well as what business entities and logic will need to be developed.

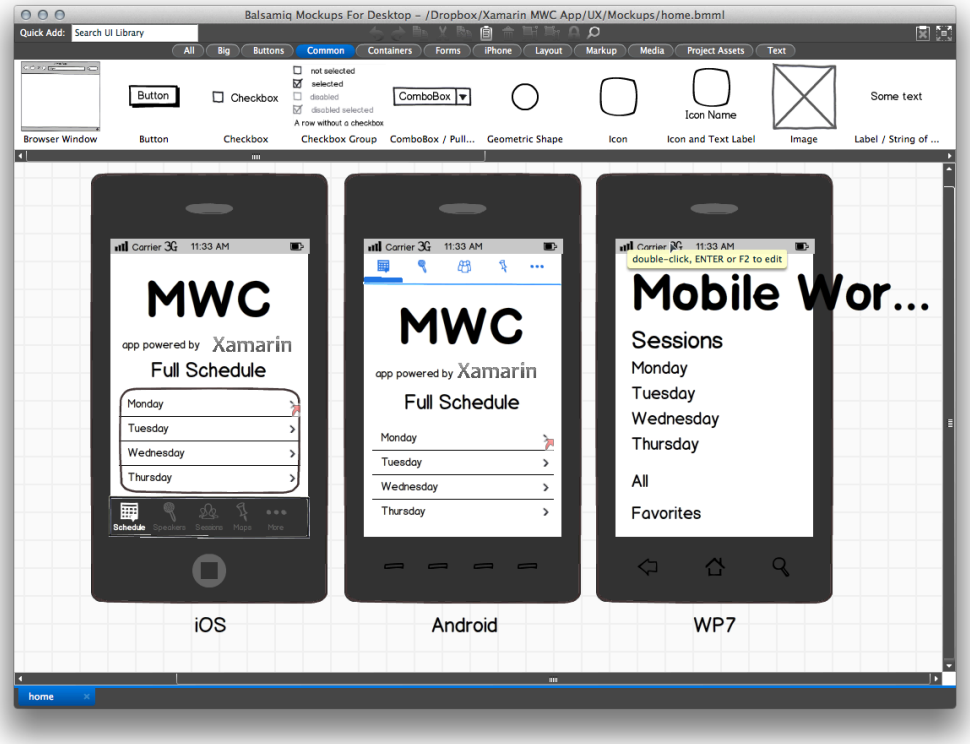
If you've captured the appropriate use cases and actors, it's much easier to begin designing an application because you know exactly what you need to design, so the question becomes how to design it, rather than what to design.

Designing Mobile Applications

Once you have a good idea of what it is you want to design, the next step is to start trying to solve the User Experience or UX.

UX DESIGN

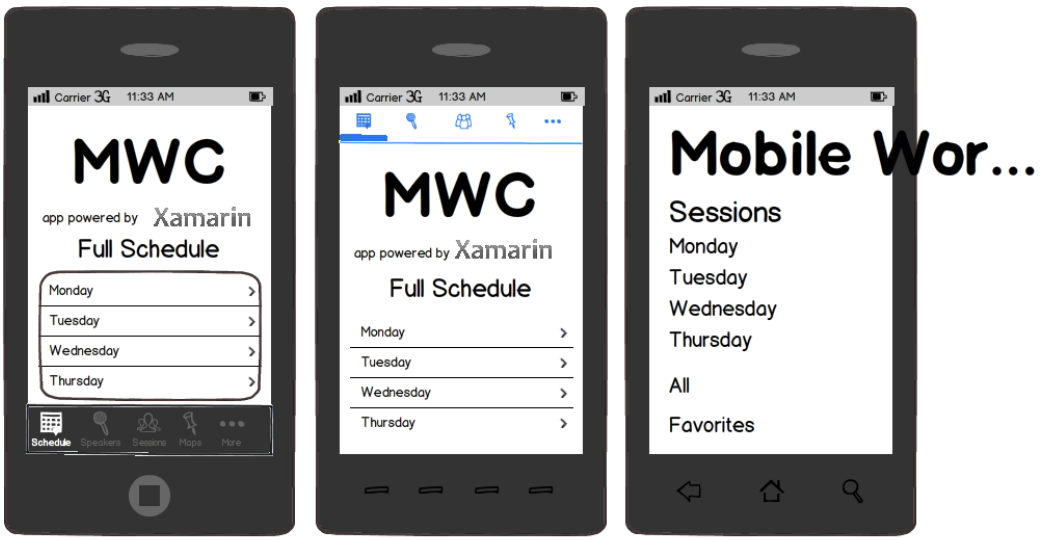
UX is usually done via wireframes or mockups, using tools such as Balsamiq, Mockingbird, Visio, or just plain ol' pen and paper. UX Mockups allow you to quickly design UX without having to worry about the actual UI design:



When creating UX Mockups, it's important to consider the Interface Guidelines for the various platforms that you're designing for. By adhering to platform-specific guidelines, you can ensure that your apps feel at home on each platform. You can find each guide at the following locations:

- ➔ Apple - Human Interface Guidelines
- ➔ Android – Design Guidelines
- ➔ Windows Phone 7 – UX Design Guidelines for WP7

For example, each app has a metaphor for switching between sections in an application. iOS uses a Tab Bar at the bottom of the screen, Android uses a Tab Bar at the top of the screen, and Windows Phone 7 uses the Panorama view:



iOS Android WP7

Additionally, the hardware itself also dictates UX decisions. For example, iOS devices have no physical *back* button, and therefore introduce the Navigation Controller metaphor:

Furthermore, form factor also influences UX decisions. A tablet has far more real estate than a phone, so you can fit more information on it; displaying information that fills multiple screens on a phone can often be compressed into a single screen on a tablet:

And given the myriad of mobile device physical conformations on the market, there are quite a few mid-size form factors (somewhere between a phone and a tablet) that you may also want to target.

USER INTERFACE (UI) DESIGN

Once you've nailed down the UX in your application, the next step is to create the UI design. While UX is typically just black and white mockups, the UI Design phase is where colors, graphics, etc., are introduced and finalized. Spending time on good UI design is important and, generally, the most popular apps have a professional design.

As with UX, it's crucial to understand that each platform has its own design language, so a well-designed application may still look different on each platform:

For UI design inspiration, check out some of the following sites:

- pttrns.com – (iOS only)
- androidpttrns.com – (Android only)
- lovelyui.com – (iOS, Android, and Windows Phone)
- mobiledesignpatterngallery.com – (iOS, Android, and Windows Phone)

Additionally, you can find graphic designer portfolios at sites such as Behance.com and Dribbble.com. Designers from all over the world can be found there, frequently in places where the exchange rate is favorable, so good graphic design doesn't necessarily have to cost a lot.

Development

The development phase usually starts very early in an app's lifecycle. In fact, once an idea has spent some time in the conceptual/inspiration phase, often a working prototype is developed that validates functionality, assumptions, and helps provide an understanding of the scope of the work.

In the rest of the tutorials, we'll focus largely on the development phase.

Stabilization

Stabilization is the process of working out the bugs in your app. Not just from a functional standpoint, for example: "It crashes when I click this button," but also from Usability and Performance perspectives. It's best to start stabilization very early within the development process so that course corrections can occur before they become costly. Typically, applications go into *Prototype*, *Alpha*, *Beta*, and *Release Candidate* stages. Definitions of these stages vary a good deal, but they generally proceed in the following pattern:

- **Prototype** – The app is still in proof-of-concept phase. At this point, perhaps only core functionality is working, or specific parts of the application may be operational, but it is unlikely that all parts of the app are executing properly. Major bugs are present.

- ➔ Alpha – Core functionality is generally code-complete (built, but not fully tested). Major bugs are still present, outlying functionality may still not be present.
- ➔ Beta – Most functionality is now complete and has had at least light testing and bug fixing. Major known issues may still be present.
- ➔ Release Candidate – All functionality is complete and tested. Barring new bugs, the app is a candidate for release to the wild.

It's never too early to begin testing an application. For example, if a major issue is found in the prototype stage, the UX of the app can still be modified to accommodate it. If a performance issue is found in the alpha stage, it's early enough to modify the architecture before a lot of code has been built on top of false assumptions.

Typically, as an application moves further along in the lifecycle, it's opened to more people to try it out, test it, provide feedback, etc. For instance, prototype applications may only be shown or made available to key stakeholders, whereas release candidate applications may be distributed to customers that sign up for early access.

For early testing and deployment to relatively few devices, deploying straight from a development machine is sufficient. However, as the audience widens, this can quickly become cumbersome. In such cases, there are a number of test deployment options that make this process much easier by allowing you to invite people to a testing pool, release builds over the web, and provide tools that allow for user feedback.

Some of the most popular test deployment options include:

- ➔ TestFlight (testflightapp.com) – This is an iOS-only product that has direct IDE support for Xamarin.iOS applications.
- ➔ LaunchPad (launchpadapp.com) – Designed for Android, this service is very similar to TestFlight.
- ➔ Zubhium (zubhium.com) – Another Android service, it's much like LaunchPad.

For Windows Phone development, Microsoft offers a beta program built into its marketplace/app center.

Distribution

Once you've stabilized your application, it's time to get it out into the wild. There are a number of different distribution options, depending on the platform.

IOS

Xamarin.iOS and Objective-C apps are distributed in exactly the same way:

- ➔ Apple App Store – Apple's App Store is a globally available online application repository that is built into Mac OSX via iTunes. It's by far the most popular distribution method for applications and it allows developers to market and distribute their apps online with very little effort.

- ➔ **Enterprise Deployment** – Enterprise deployment is meant for internal distribution of corporate applications that aren't available publicly via the App Store.
- ➔ **Ad-Hoc Deployment** – Ad-hoc deployment is intended primarily for development and testing and allows you to deploy to a limited number of properly provisioned devices. When you deploy to a device via Xcode or MonoDevelop, it is known as ad-hoc deployment.

ANDROID

All Android applications must be signed before they are distributed. Developers sign their applications by using their own certificate protected by a private key. This certificate can provide a chain of authenticity that ties an application developer to the applications that developer has built and released. It must be noted that while a development certificate for Android can be signed by a recognized certificate authority, most developers do not opt to utilize these services, and self-sign their certificates. The main purpose for certificates is to differentiate between different developers and applications. Android uses this information to assist with enforcement of delegation of permissions between applications and components that run within the Android OS.

Unlike other popular mobile platforms, Android takes a very open approach to app distribution. Devices are not locked to a single, approved app store. Instead, anyone is free to create an app store, and most Android phones allow apps to be installed from these third party stores.

This distribution method creates a potentially larger and more complex distribution channel for developers and their applications. Google Play is Google's official app store, but there are many others. A few popular ones are:

- ➔ AppBrain
- ➔ Amazon App Store for Android
- ➔ Handango
- ➔ GetJar

For more information, see the Publishing an Application guide.

Windows Phone 7

Windows Phone applications are distributed to users via the Windows Store (previously known as the Marketplace). Developers submit their apps to the Windows Phone Dev Center (formerly App Hub) for approval, after which they appear in the Store.

Applications can be tested on any device that has been registered for development–simply build the XAP file and it can be installed (or "side-loaded") with the Windows Phone Application Deployment tool. A maximum of ten apps can be installed and tested at any one time using this tool.

Microsoft recently added the ability to distribute apps for beta testing via the Marketplace. Developers can submit their apps and then provide an install link to testers, before the app is reviewed and published.

Summary

This guide gave a comprehensive introduction to mobile development in general, as well as an examination of the SDLC as it relates to mobile development. It introduced general considerations for building mobile applications and examined a number of platform-specific considerations including design, testing, and deployment.

It serves as a primer and provides context for working through the Getting Started Series. From here, the next step is to read either the Hello, iOS or the Hello, Android guide, depending on which platform you'd like to begin developing for first.