Introduction to mobile development with Xamarin

Enterprise & Mobile .Net

DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

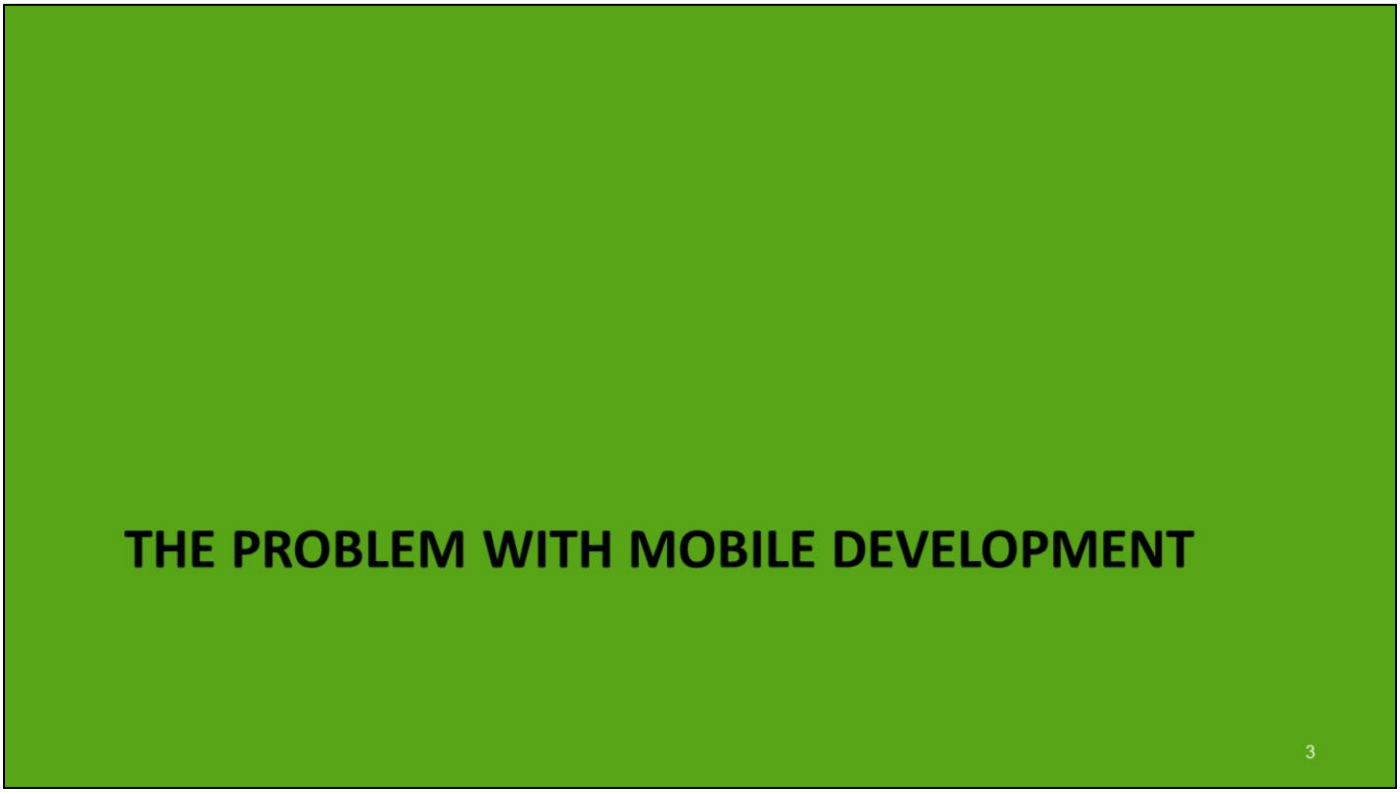
Agenda

- The problem with mobile development
- The Xamarin Ecosystem
- Code-sharing with Xamarin
- Getting your environment ready for Xamarin

PXL IT

These slides are based on the following course:
https://app.pluralsight.com/library/courses/xamarin-big-picture

Detailed documentation on:
https://docs.microsoft.com/en-us/xamarin



When we think about apps, large or very complex apps come to mind probably.

Their creators spend tons of money to give a **great user experience** on just any device.

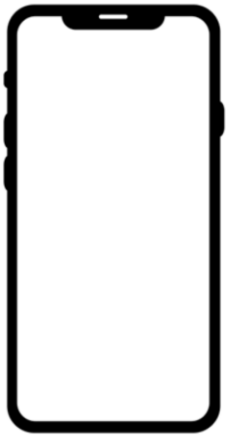
It's the experience that is very important here. We want to give the user the **native experience of the platform**.

A native app in the context of mobile applications is an app that **is typically built with the tools of the vendor**, and therefore gives the developer all the **options that the platform has to offer**.

We can interact with the camera, use the native way to build the UI, have great performance, work with sensors, and yes, like I said, anything really that is available on the platform.

Building Native Apps: iOS

- Native iOS apps are built with:
  - Xcode
  - Swift or Objective-C
  - iOS frameworks



PXL IT

Building Native Apps: Android

- Native Android apps are built with:
  - Android Studio
  - Java
  - Android SDKs

PXL IT

Building Native Apps: Windows

- Native Windows apps are built with:
  - Visual Studio
  - C#
  - .NET APIs
  - Flavours:
    - Windows Forms
    - WPF
    - UWP

PXL IT

6

Windows Forms

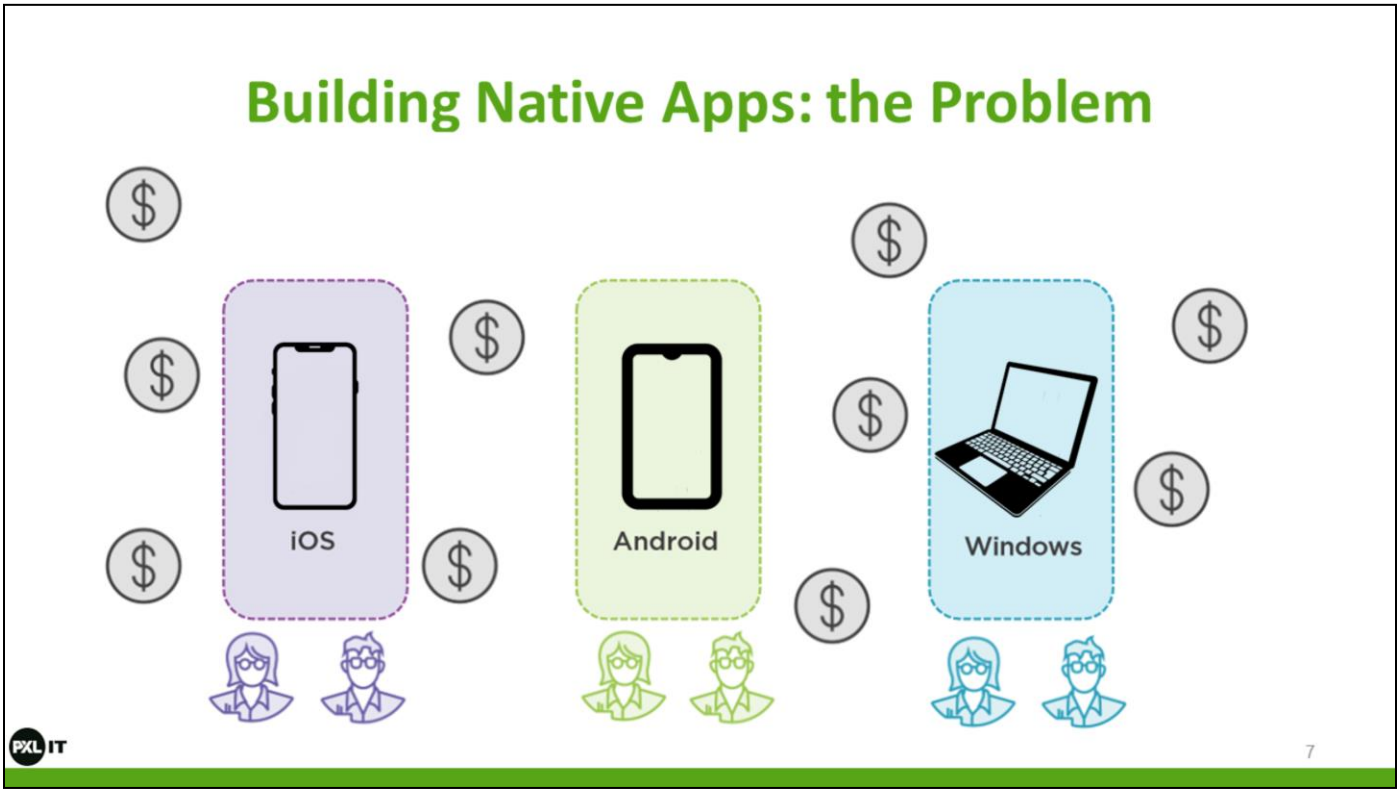
- Oldest kind of native apps on Windows, they run on all (legacy) versions of windows (going Windows 95 and up)
- These apps don't display well on high-res displays
- Drag and drop components onto a form: difficult to built maintainable apps

WPF (Windows Presentation Foundation)

- Introduced on Windows Vista
- XAML development model, allows for better architected apps (e.g. MVVM)

UWP (Universal Windows Platform)

- Runs on all flavours of Windows 10, e.g. HoloLens, Raspberry Pi, laptop
- Evolved XAML development model, including full touch support
- Maybe the platform of the future?



Think back of what I've just said, when we build an iOS application, we're going to have to bring in a team of iOS developers, so people that build applications using Xcode, and have knowledge about Objective-C or Swift.

If we then decide that you also want to build the Android application, we're going to have to bring in a team of Android developers who have knowledge about Java and the Android SDKs.

And if we then also want to build a Windows application, we have to bring in a team of C# developers.

This approach is what you see very often in companies doing mobile today. **While the apps that you are getting this way are native apps, the main problem is cost**.

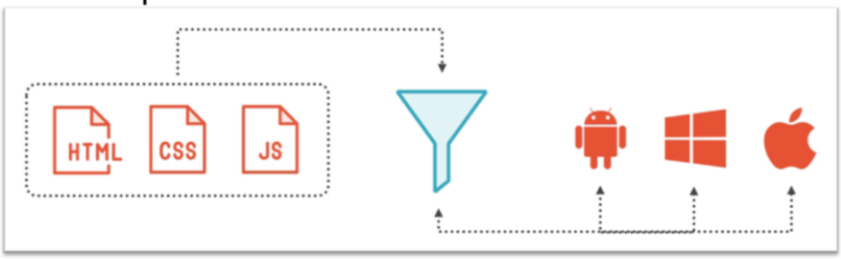
We have **three developer teams** building this same app for a different platform.

These teams can't really share anything between them since they don't speak the same language.

When we want to add a new feature or fix some bug, we'll have to do that three times, again a very costly affair.

Hybrid approach

- Lowest common denominator
  - HTML, CSS, javascript
- Mobile website in wrapped an application package
  - Ionic, Cordova, Sencha Ext JS, Onsen UI
  - Adds extra layer to allow native functionality
- Target the mobile platforms with the same code base



PXL IT

If you think about it, what code runs on iOS, Android, and Windows Phone in roughly the same way?

Exactly: **HTML**, **CSS**, and **JavaScript**.

We can actually write an app in these languages, often backed by framework such as **Ionic** or **Cordova**.

Basically what you're building is a **mobile website wrapped in an application package**.

And using a framework (like Cordova) gives you access to some (not all) of the native features of the operating system.

Now this approach definitely has a **huge benefit**. We can now really **write almost all the code once** and **target multiple mobile platforms** we want to reach with that same code base.

Now on the other hand, this approach has, of course, some **disadvantages**.

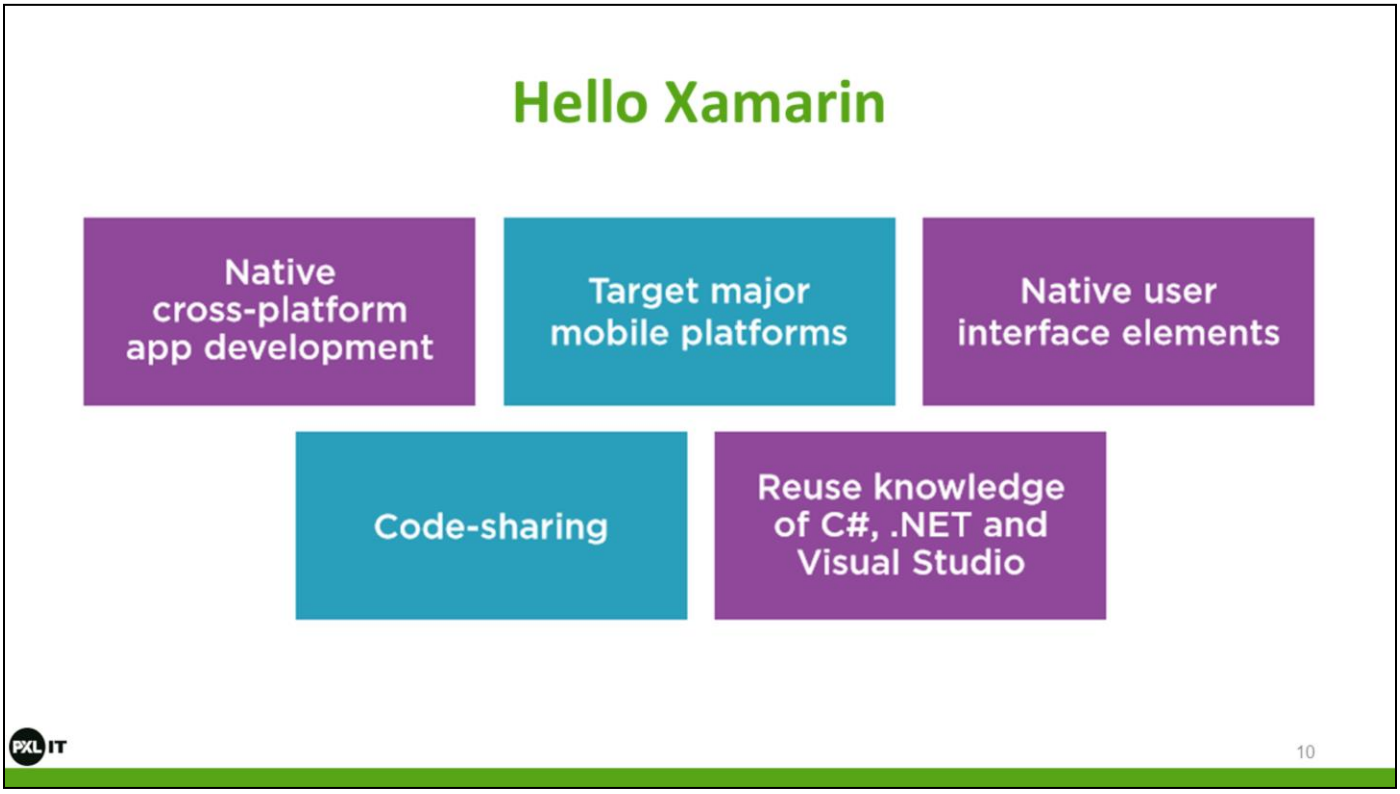
The **performance** isn't always perfect. Like I said, you're running a website, so web code, which can be less optimal than a natively written app.

Secondly, you **don't always have access to all native features** of the platform, which can result in less than optimal experiences for the end user,

But to be clear, compared to the silo'd approach (multiple development teams), the web-based approach will save you a lot of money, there's no question about that.

As you may have picked up, and I hope you did, you'll see that between the different approaches we have seen so far, there are some **decision makers** that play an important role **when selecting a platform** to build mobile applications.

- Of course, the **cost** and the time to get an application ready is an important one, and you have seen that this is not always great with the fully-native approach.
- **Performance** of the app is another one, and the web-based applications sometimes have issues still in this area.
- **Reuse of knowledge** is another important one I think here.
  If you can have your development team build multiple applications, instead of having to bring in multiple teams should definitely also be an important point to take into account when deciding on a platform to build mobile applications.



Xamarin is a framework from Microsoft that allows us to do **cross-platform development**.

The apps that you are building with Xamarin are **native apps**, so that means that the outcome of a compiled Xamarin app is the same as when you would build it with Android Studio and Java or Objective-C and Xcode.

Using Xamarin, we can target **the three major mobile platforms** out there, so **iOS, Android, and Windows**.

It doesn't stop there though, using something called Xamarin.Forms, we'll be able to even target **Windows desktop apps and much more**.

But we'll talk about that later in this course.

We get to build the UI of the applications with the same **native user interface elements**, so we will not get into a situation where you can't really build the application like you were asked to do.

As we have discussed, the cost is important when building mobile apps. In the case of Xamarin, we'll be able **to reuse quite a lot of the code**.

Code sharing between platforms is possible, and it is in fact the default way of working in Xamarin.

Later in this course, we'll have a full module on the aspect of code sharing. It's indeed that important.

The code that we write to target the three platforms is **C# code**. We'll typically write the code using **Visual Studio**, and we are using the **.NET Framework**.

So if you have a background in .NET development, you'll be able to now use that knowledge to target mobile applications running on iOS, on Android, on top of the platforms that you could already target with C#.

This way, Xamarin is really hitting the sweet spot.

Performance of Xamarin Apps

- You create 100% Native Apps
- Therefore you get 100% Native Performance

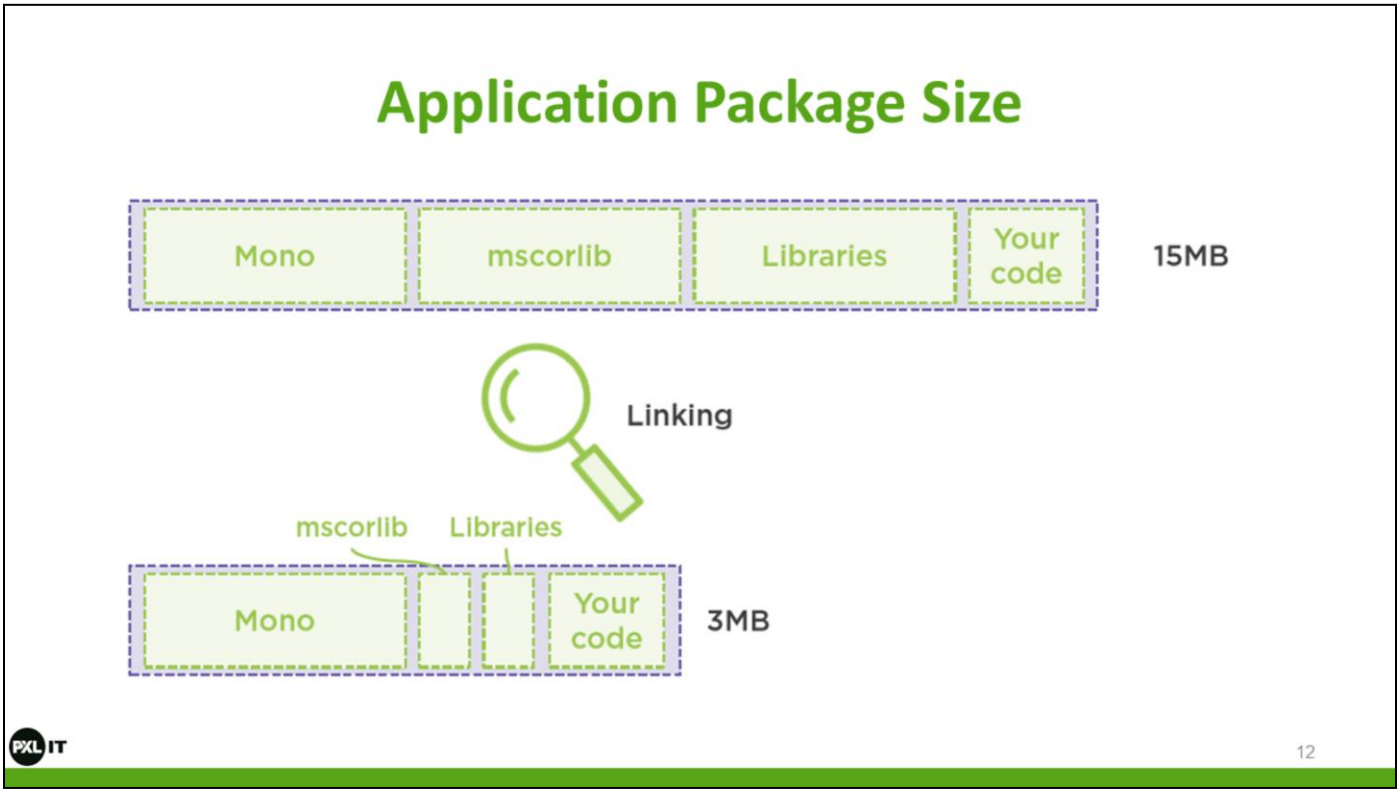
PXL IT

11

When you compare a Xamarin application to an iOS application written in Objective-C or Swift, there won't be any real notable difference.

The same goes for Android applications written in Java.

Xamarin applications are **compiled into a native app**, which means that you get close to no performance impact.



Xamarin application packages tend to be somewhat bigger compared to their real native counterparts.

Now why is that, you may be asking.

Remember that when we build a Xamarin application, what we're really going to do is **write C# code against the Mono Framework**, so the open-source version of the .NET Framework.

When you run a .NET Xamarin application on Android or iOS, they don't come with .NET installed, right?

So the problems, though to say, come down to this. **Xamarin will need to include the Mono Framework in your application package**, hence the increased size.

By default, you may be astonished by the initial package size. A simple Hello World application can already be more than 15 MB.

That is not acceptable in most cases. On the Xamarin website on the link you see here, you'll find an overview of what exactly goes into that package.

You can see that here as well. Most of the application package size comes from the Xamarin Android Framework that's going to be included.

The good news is that the application package size can be made a lot smaller. We'll have to optimize it through a process that can be done automatically when packaging from Visual Studio, and it's called linking.

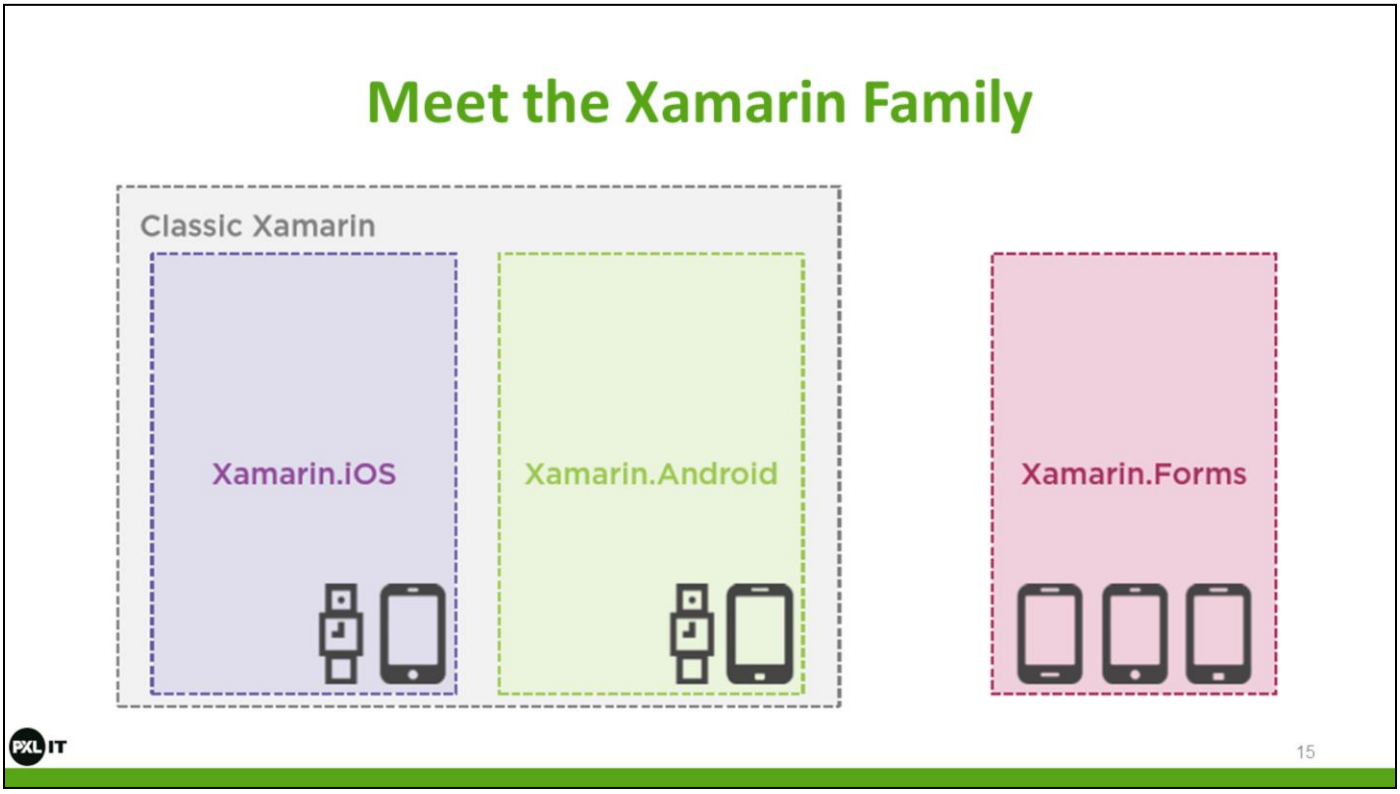
Basically what this will do is it will scan your code and see which APIs you are using. It will throw out the ones that from your code you aren't using. As you can see, this will bring down the application package size to an acceptable level.

More info: https://docs.microsoft.com/en-us/xamarin/android/deploy-test/app-package-size

Disadvantages of Xamarin

- It has a steep learning curve
- It's not a shared UI Platform (yet: Xamarin.Forms)
- You need to understand each platforms UI controls and UX paradigms
- You need a Mac for iOS development

THE XAMARIN ECOSYSTEM



We have seen that Xamarin can be used to build iOS and Android applications, and in fact also Windows applications.

The part of Xamarin that allows us to build iOS applications is conveniently named **Xamarin.iOS**.

Through Xamarin.iOS, we can write native iOS applications using C#.

Very soon, you'll see that the UI can be built with something called a storyboard, the native approach to building an iOS interface.

Xamarin.iOS allows us to build iOS applications that run on a phone, a tablet, or both.

Also, we can target Apple Watch applications from Xamarin.

Next, the product that will allow us to build Android applications is quite logically called **Xamarin.Android**.

Using Xamarin.Android, we can build apps that run on Android phones, on tablets, and in fact everywhere where Android runs.

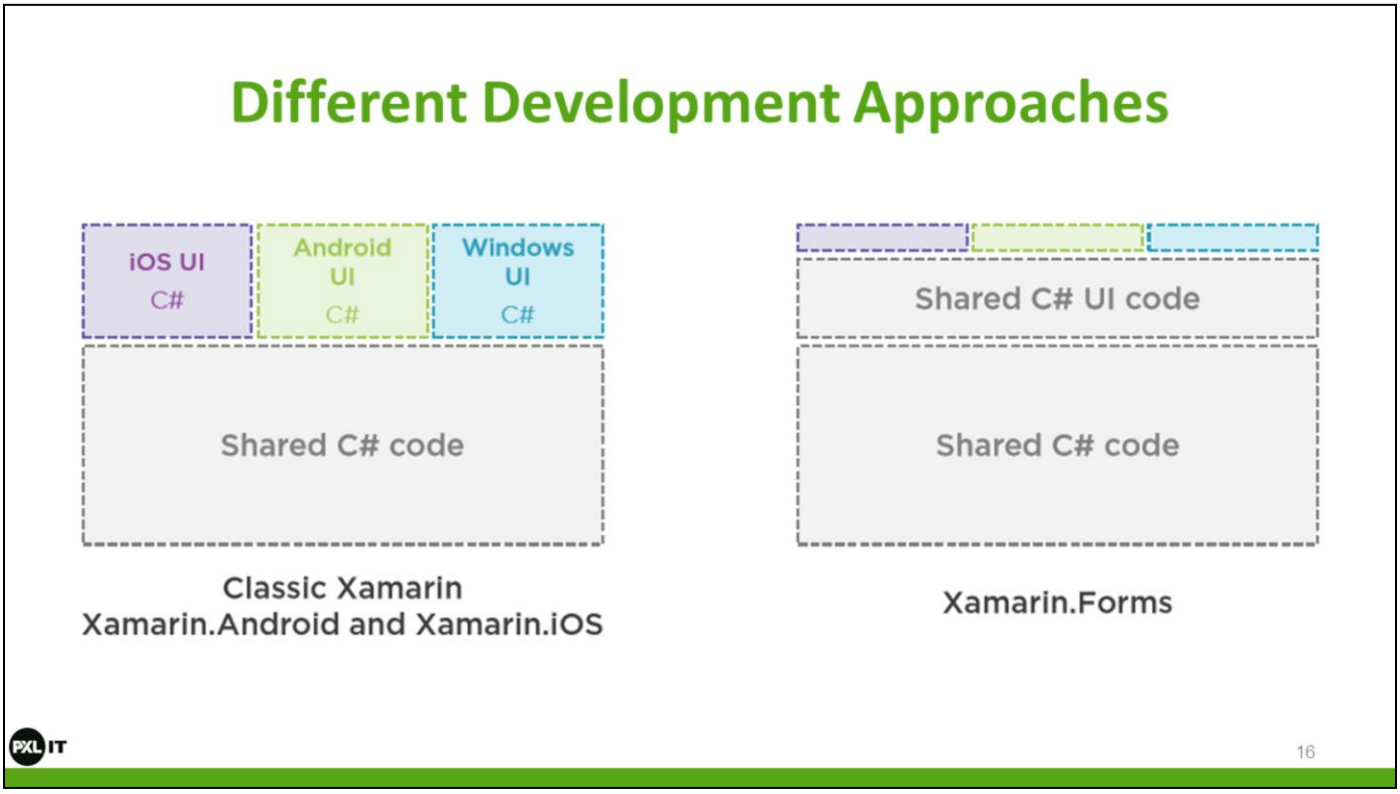
These two building blocks are the original way to building Xamarin applications. **What they have in common is that the UI layer cannot be shared here**.

We have to build the UI again for each platform we are targeting.

These two together are also referred to as **classic Xamarin or traditional Xamarin**.

Now if there's a traditional way, there must also be a non-traditional way, right, and that is **Xamarin.Forms**.

Using Xamarin.Forms, we can build apps that also target iOS, Android, and Windows.



As mentioned, with the **classical Xamarin** approach, we can build apps as we've explained in the previous chapter.

Using this approach, **we write part of the applications once through shared code**.

I hope you remember that this is then the **non-UI code**, such as services, models, data access, and alike.

On top of this shared code, we can then add an application head, the actual application project, which is then Xamarin.Android, Xamarin.iOS, or Windows.

All of these use C#, except for the actual UI layer. Note that although I include Windows here, Windows isn't really part of what Xamarin is covering.

Windows is covered by what Microsoft has created to allow us to build UWP applications, or Universal Windows Applications.

The key takeaway here is that with this approach, we can **share around 60% of the code** depending a bit on the application.

Now with **Xamarin.Forms** then, well take a look a look at the high-level architecture of a Xamarin.Forms application, and you'll notice something different.

Code sharing now is possible also in the UI layer as well. We can indeed now write most, if not all of our UI code once, and use that to create an iOS Android and also a Windows application.

Indeed, using Xamarin forms, Windows apps now also become a target. We'll explore Xamarin.Forms in much more detail later on.

An overview of Xamarin.iOS

- Build native apps for iOS
- Use C# everywhere
- Native way to build the UI → Storyboard
- Requires Mac for compilation, debugging and deployment

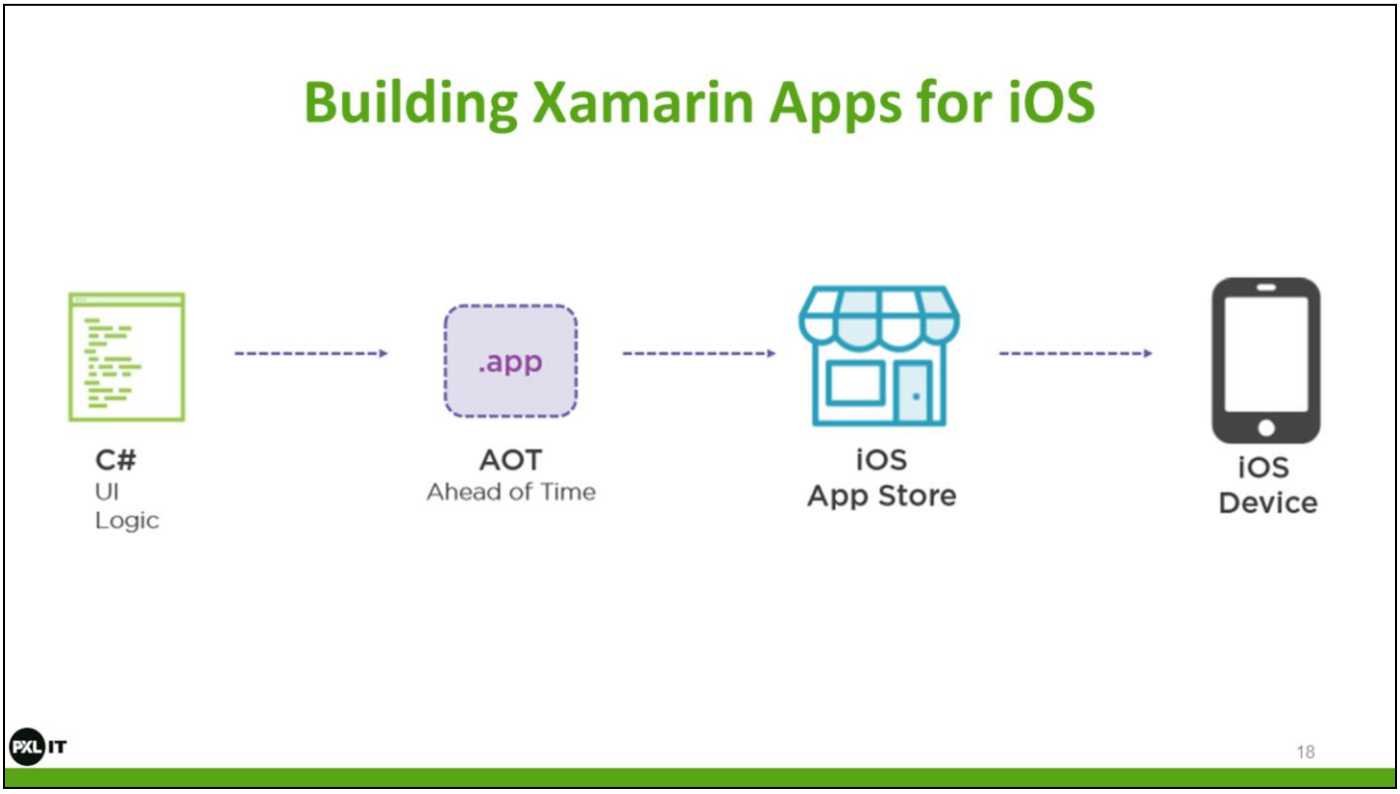
PXL IT

17

https://docs.microsoft.com/en-us/xamarin/ios/get-started/

https://docs.microsoft.com/en-us/xamarin/ios/get-started/installation/windows/?pivots=windows

https://docs.microsoft.com/en-us/xamarin/ios/user-interface/



Let's now break down the flow to build an iOS application using Xamarin.

We'll **write C# code** using Visual Studio.

Visual Studio not only exists for Windows, but the version now is also available for Mac OS, so that you'll be able to have the Visual Studio experience when writing Xamarin.iOS applications on a Mac as well.

Visual Studio will create for you an **app package**, an application package that is.

When building an application for iOS, something called **AOT** is being used, which is short for **Ahead-of-Time compilation**.

Ahead-of-Time is like the opposite of Just-in-Time compilation, which is the default when writing .NET applications.

With Just-in-Time, something called IL or intermediate language is produced when we compile that application, and that is then Just-in-Time compiled again to its machine-level code.

Just-in-Time compilation means that dynamic compilation is happening; however, that very item **isn't allowed on iOS**.

Apple does not allow dynamic compilation of code on an iOS device. For that very reason, Visual Studio will use Ahead-of-Time compilation, which means that **ARM code will be generated**, so directly machine-level code.

Now when your package is then tested and ready to be released, typically in iOS, you'll publish your application to the **iOS App Store**.

From there, your customers can download the application to their iOS devices.

When we build a Xamarin.iOS application, we get **access to all native iOS SDKs**, including UIKit, SiriKit, MapKit, Contacts, ARKit, and many, many more.

An overview of Xamarin.Android

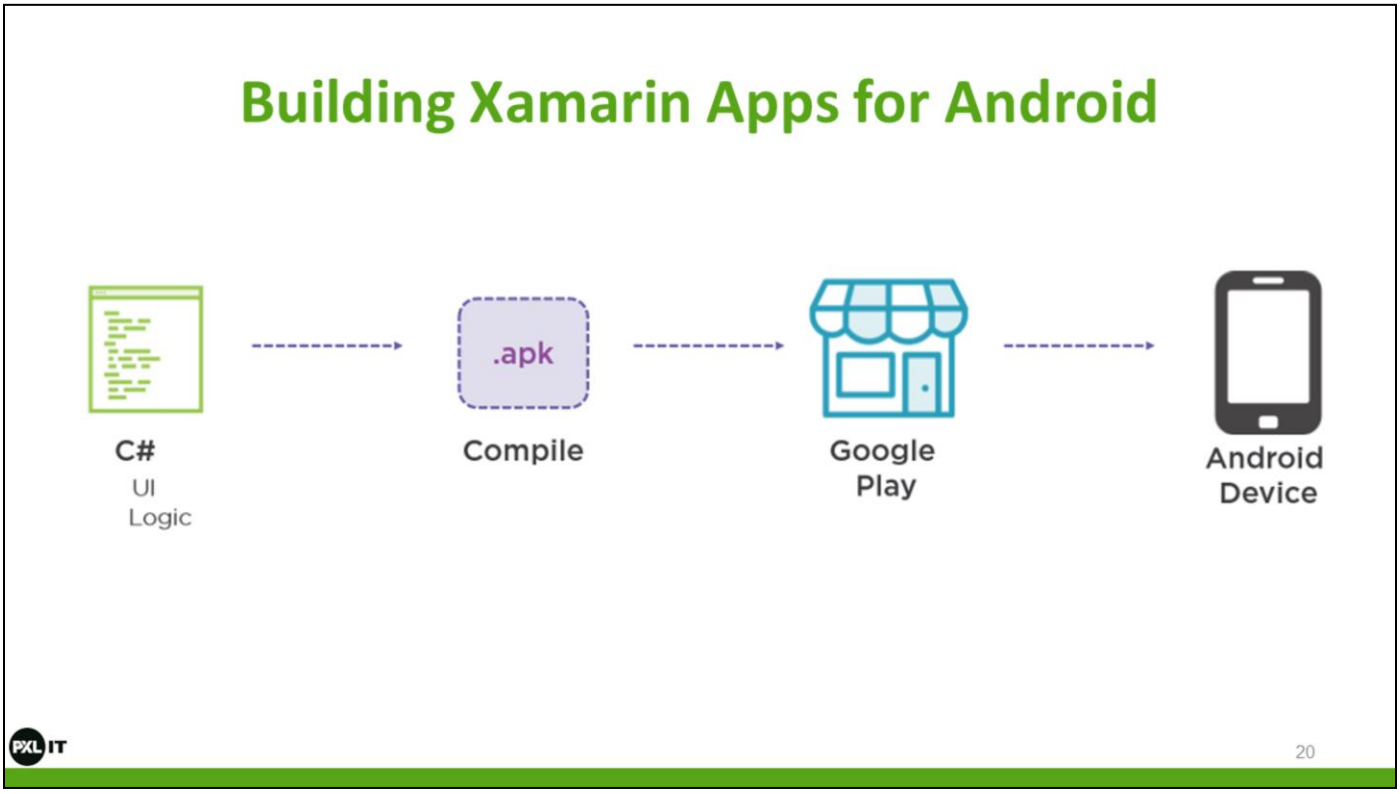
- Build native apps for Android
- Use C# everywhere
- Native way to build the UI → Android XML
- Works on PC and Mac

PXL IT

19

https://docs.microsoft.com/en-us/xamarin/android/get-started

https://docs.microsoft.com/en-us/xamarin/android/user-interface



Now to build **Xamarin.Android app**s, you don't really need anything special, just a simple PC or Mac will do, as you can **write all the code in Visual Studio** on either of the platforms.

We can use Visual Studio to build our application. Both Visual Studio on Windows, as well as Visual Studio on Mac have out-of-the-box support for building Xamarin.Android apps.

When we **compile** our code, what comes out is an **Android APK file**, an Android application package file.

This is what your end users will install on their device. Notice here that we don't use Ahead-of-Time compilation.

When using Xamarin Android, our applications are **compiled into IL code** (Intermediate Langague), which is then **dynamically compiled using Just-in-Time compilation**, much like a .NET application on the PC.

When your application is ready and tested, well then it's time to publish it.

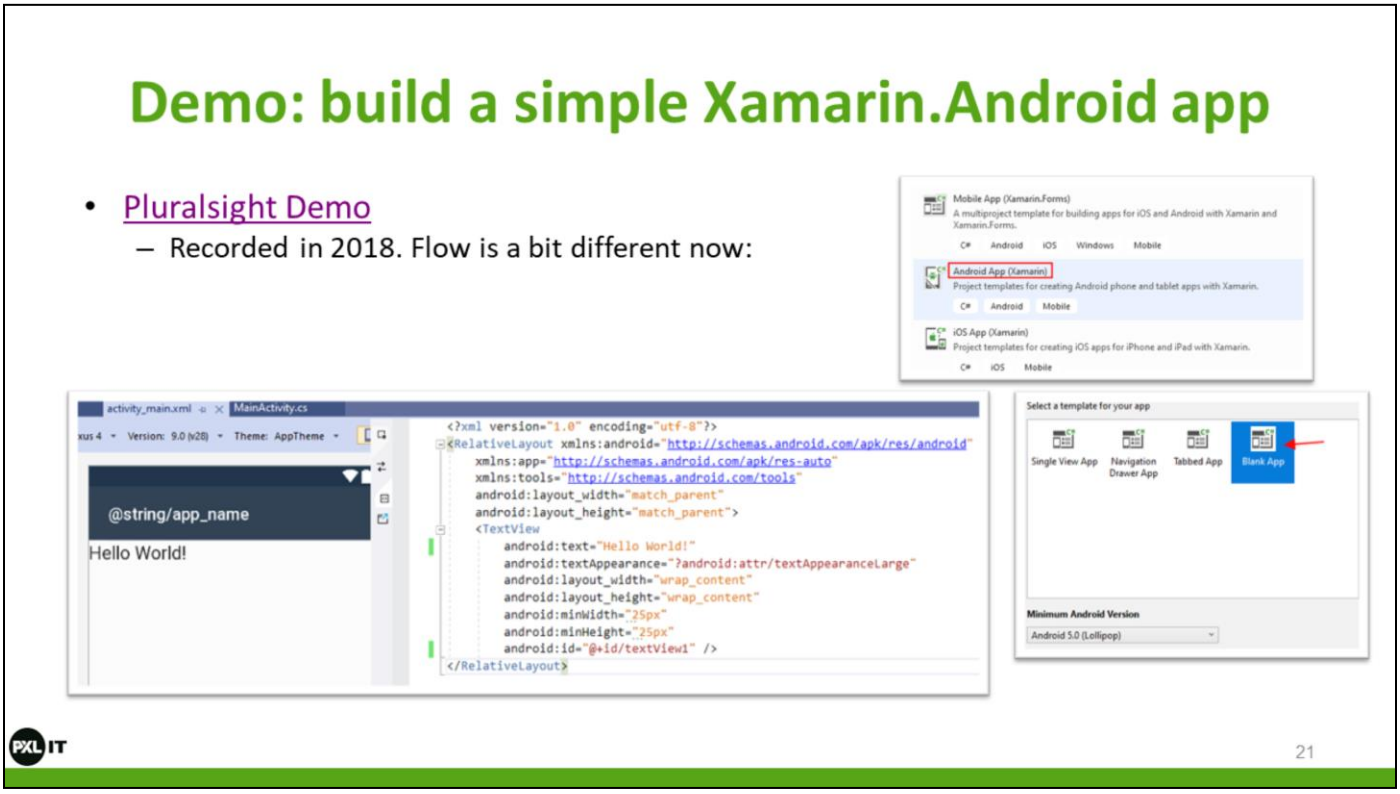
When building Android apps, these will typically be submitted to the **Google Play Store**, and from there, users can install them on their Android devices.

Know that Android is more flexible when it comes to users installing your application. You can in fact deploy your application to other Android stores as well.

Quite a few exist. Next to that, it's very simple as well to just upload your APK somewhere, perhaps to an FTP, and that users download and install the

application from there.

When we build a Xamarin.Android application, we get **access to all native Android SDKs**, including Maps, Layout and Views, Audio, Contacts, ARCore, ….



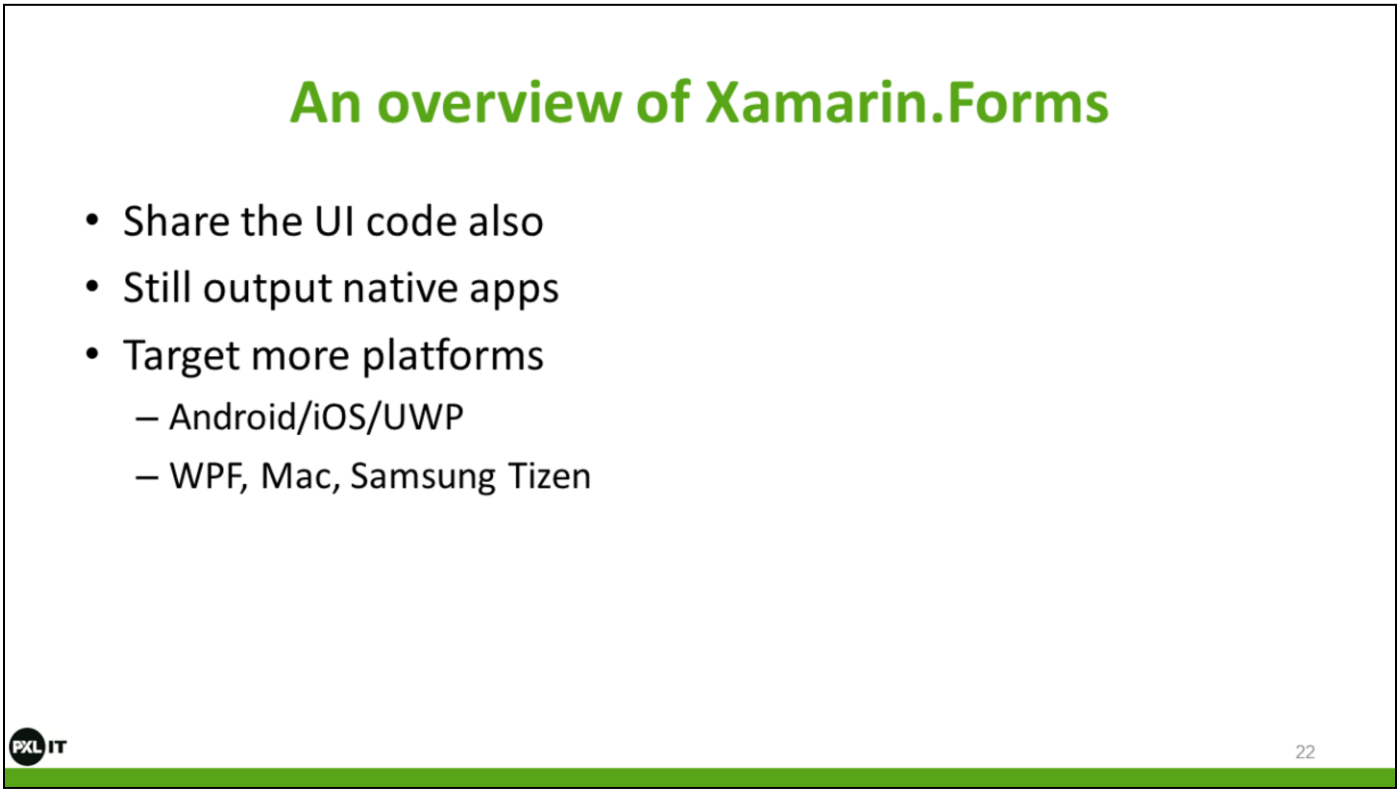
Pluralsight Demo : HelloWorldAndroid

In the Pluralsight Demo the Genymotion emulator is used. We will use the standard Android emulators or a real device.

How to create ‘HelloWorld’ App in 2020:

- Project Type: Android App
- Template: Blank App
- Minimum Android Version: Android 5.0 (Lollipop)
- Add Text (Large) “Hello World” into activity_main.xml

Note on resources: Newer releases of Visual Studio support opening .xml files inside the Android Designer. Both .axml and .xml files are supported in the Android Designer.



We've now covered the two approaches known as **classic Xamarin**.

In the next part, I'm going to introduce you **to Xamarin.Forms**, an approach to building Xamarin applications introduced in 2014.

So while it's been around for some time, it's definitely the newest addition to the Xamarin family. Let's dive in.

If we go to the Xamarin website, we'll find a definition for Xamarin.Forms saying that **Xamarin.Forms allows us to build a native UI for iOS, Android, and Windows form a single, shared codebase**.

Now let's zoom in on that sentence right here.

Using Xamarin.Forms, we can build native apps for the three platforms. Indeed, this already is kind of different in that Xamarin.Forms also allows us to **target Windows**. That was not the case for classic Xamarin, I hope you remember.

The apps that we're going to get are **still native apps**, that hasn't changed, compared to the classic Xamarin approach.

What has changed though is the amount of code that can be shared.

In Xamarin.iOS, we had to create a storyboard for the UI layer.

In Xamarin.Android, we had to create XML for the UI. That UI could not be shared.

Now Xamarin.Forms changes this. **We can also share the UI code**. You should see Xamarin.Forms as an abstraction on top of the three platforms really.

What the folks at Xamarin have done is that they've taken a look at the three platforms, and they said let us see what is similar between these platforms.

Now if you think about it, a button exists on each platform.

Tabs exist on all platforms as well.

Lists also exist on all platforms.

So what they have done is they've moved all that is similar into a **UI abstraction layer**. That layer is Xamarin.Forms.

When we thus create the Xamarin.Forms app, we're building a UI using these abstract layer components.

Behind the scenes, Xamarin.Forms will then handle that when we use an abstract button on iOS, it will get translated into a UI button, so a button on iOS.

Similarly, when that same code runs on Android, Xamarin.Forms will make sure that the abstract button gets translated into an Android button.

So the actual translation, let's say, from the abstract UI layer to the platform-specific layer is done by the platform.

That means that we now only have **to write the code once, and the platform will make sure that the actual UI components are going to be used**, and thus we still get native applications.

Note that Xamarin.Forms will not be generating a storyboard in the case of iOS, or XML in the case of Android.

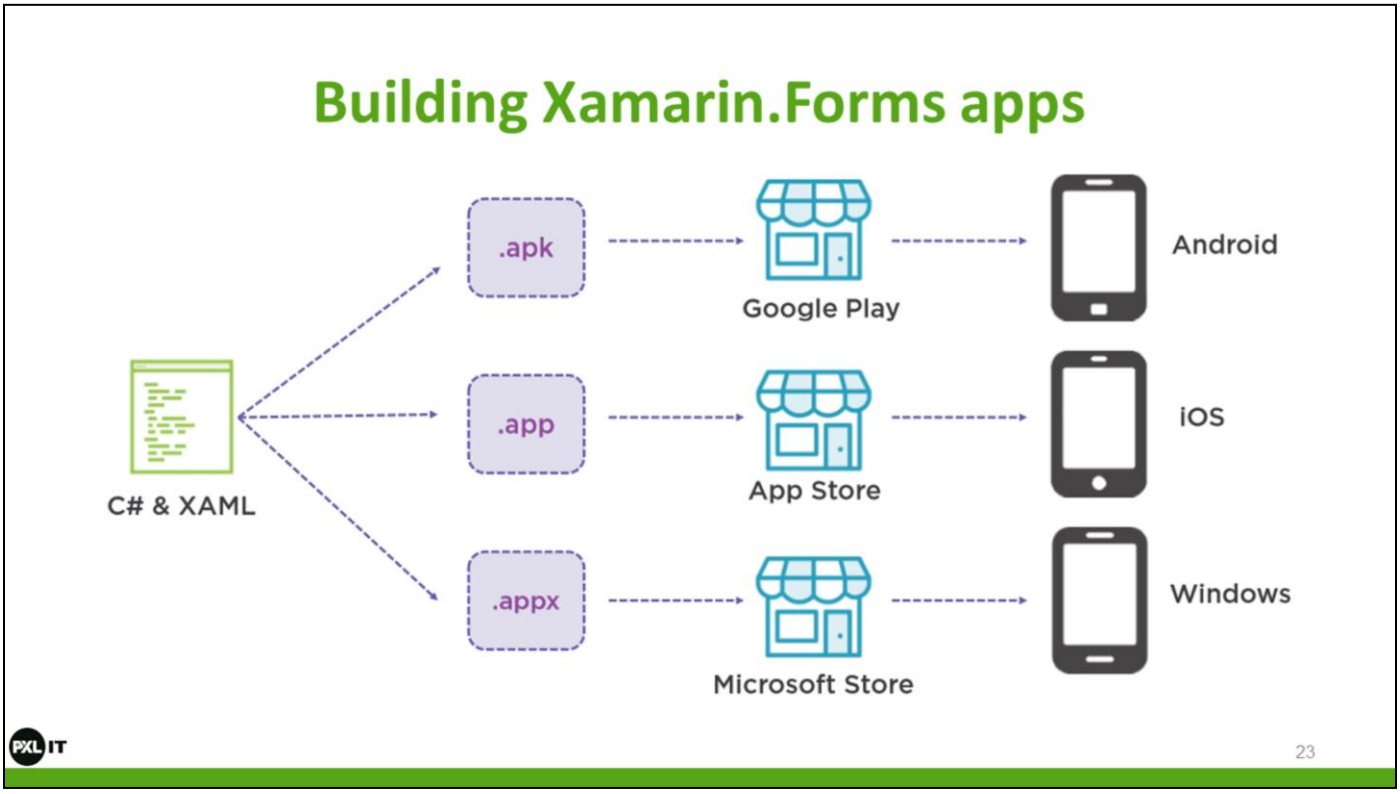
It builds up your UI dynamically.

Xamarin.Forms also allows us to **target more platforms**.

I'm not talking here about iOS, Android, and Windows, but using Xamarin.Forms, you can even **target WPF, Mac, or Samsung Tizen**.

Xamarin website: https://docs.microsoft.com/en-us/xamarin

Supported platforms: https://docs.microsoft.com/en-us/xamarin/get-started/supported-platforms



So as mentioned, Xamarin Forms is really a UI Framework, an abstraction on top of the actual platforms.

Forms itself focuses entirely, let's say, on the UI side of things.

It allows us to **write most of the UI code once, and reuse it to target the different platforms** we want to reach with our app.

Writing Xamarin.Forms code is typically done in **XAML**.

If you're already familiar with XAML, you can leverage that knowledge when building Xamarin.Forms apps.

Note that the XAML is **very similar to Windows XAML**, but some minor things are different.

The outcome is still a native app, and the UI components that are being used in your app are still the native UI components.

It's not now all of a sudden that Xamarin is going to start being a hybrid approach.

Since Xamarin.Forms is an abstraction, it also comes with some consequences.

Being an abstraction out of the box, not everything is there. Not every control that's available in iOS or Android is available by default.

If a control that you need in your app is not included, you'll be happy to hear

that Xamarin.Forms is really **extendable**.

You can add all controls that aren't in there by default, which is of course good news.

Since Xamarin.Forms has been around for quite some time now, a lot of component vendors already have suites of controls available.

Also, you'll find a lot of open-source components that you can use to plug in missing functionality if needed.

Note also that Xamarin.Forms is getting a lot of focus from Microsoft.

Very frequently, new features are being added to make everything you need possible in Xamarin.Forms.

Xamarin.Forms, as said, is getting a lot of attention from Microsoft.

For them, Xamarin.Forms is the part of the vision of having to write the code once, and being able to run it everywhere.

With more and more targets being added, this is becoming a reality now.

Now while we are closer than ever to writing code once and running it everywhere, it's not the case that all code can be shared.

However, with Xamarin.Forms, we can reuse a lot of the code.

In a Hello World application, you can reuse everything.

In a more realistic application, the level of **code** that you can typically **reuse** lies **between 80 and 90%**, which if you think about it, is still great if you ask me.

Now since I said in most cases we won't be able to share 100% of the code, where will the platform-specific code then pop up, you may be asking.

Very good question.

For each platform there is still a project on top of the shared UI code. This layer contains platform-specific code.

This typically includes things such as the code for a control that is not available in Xamarin.Forms by default, and we thus need to add it to each platform.

Depending on how complex the UI really is, you may need to be adding more custom code for each platform.

The more platform-specific code you'll need, the lower the level of shared code will eventually be.

But rest assured, even in real-life applications, that level is still between 80 and 90%.

With Xamarin.Forms, we can write most of the code once.

Typically, **your entire code base will consist now out of C# and XAML**

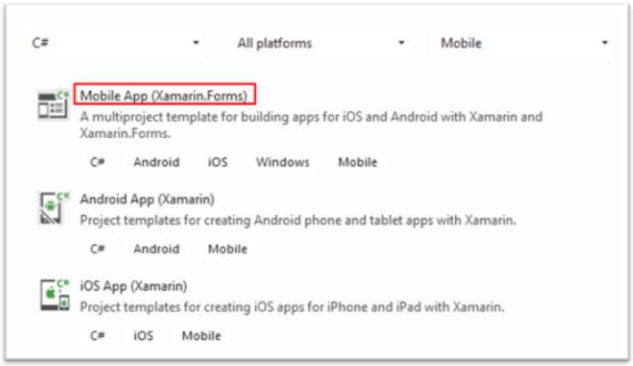
code.

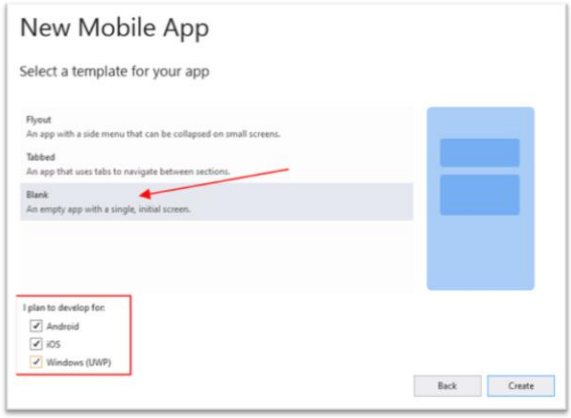
When we compile our application, we'll get an APK for Android that can be deployed to the Google Play Store in much the same way as with the Xamarin.Android application, and from there it can be downloaded to a device. We'll also get an app file for iOS, which can be deployed to the iOS app store from where users can download the app.

Also, a Windows UWP application will be created, and that can be deployed to the Microsoft store.

Demo: build a simple Xamarin.Forms app

- Pluralsight Demo
  - Recorded in 2018. Flow is a bit different now:





PXL IT

24

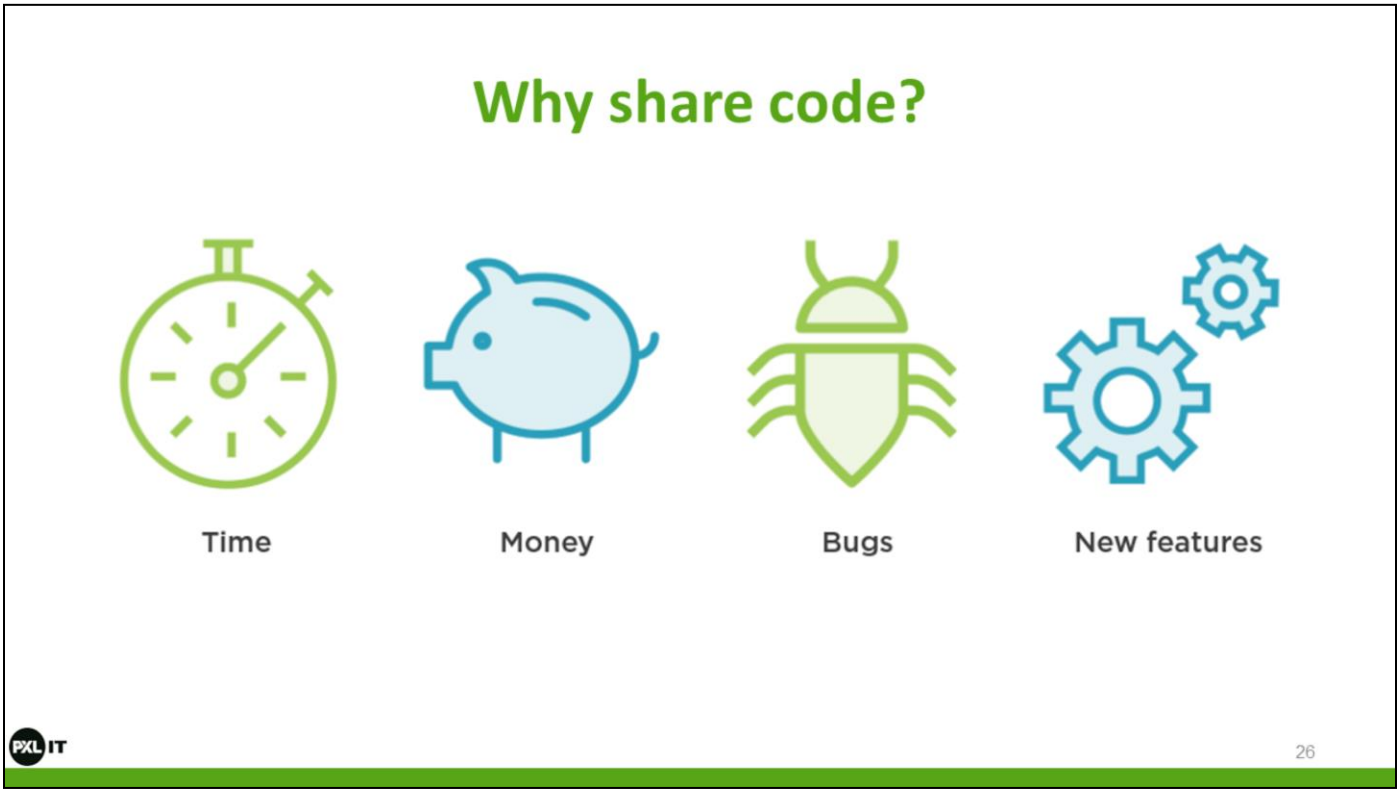
Pluralsight Demo : HelloWorldXamarinForms

FYI: How to create 'HelloWorld' App yourself:

- Project Type: Mobile App (Xamarin.Forms)
- Template: Blank
- Platform: Android, iOS, Windows (UWP)

https://docs.microsoft.com/en-us/xamarin/get-started/first-app

CODE-SHARING WITH XAMARIN



When we share part of our code between different platforms, we really start reaping one of the biggest benefits of Xamarin.

We won't have to write every piece of code and functionality for each platform.

Instead, we will write everything once, and we will reuse that across different platforms.

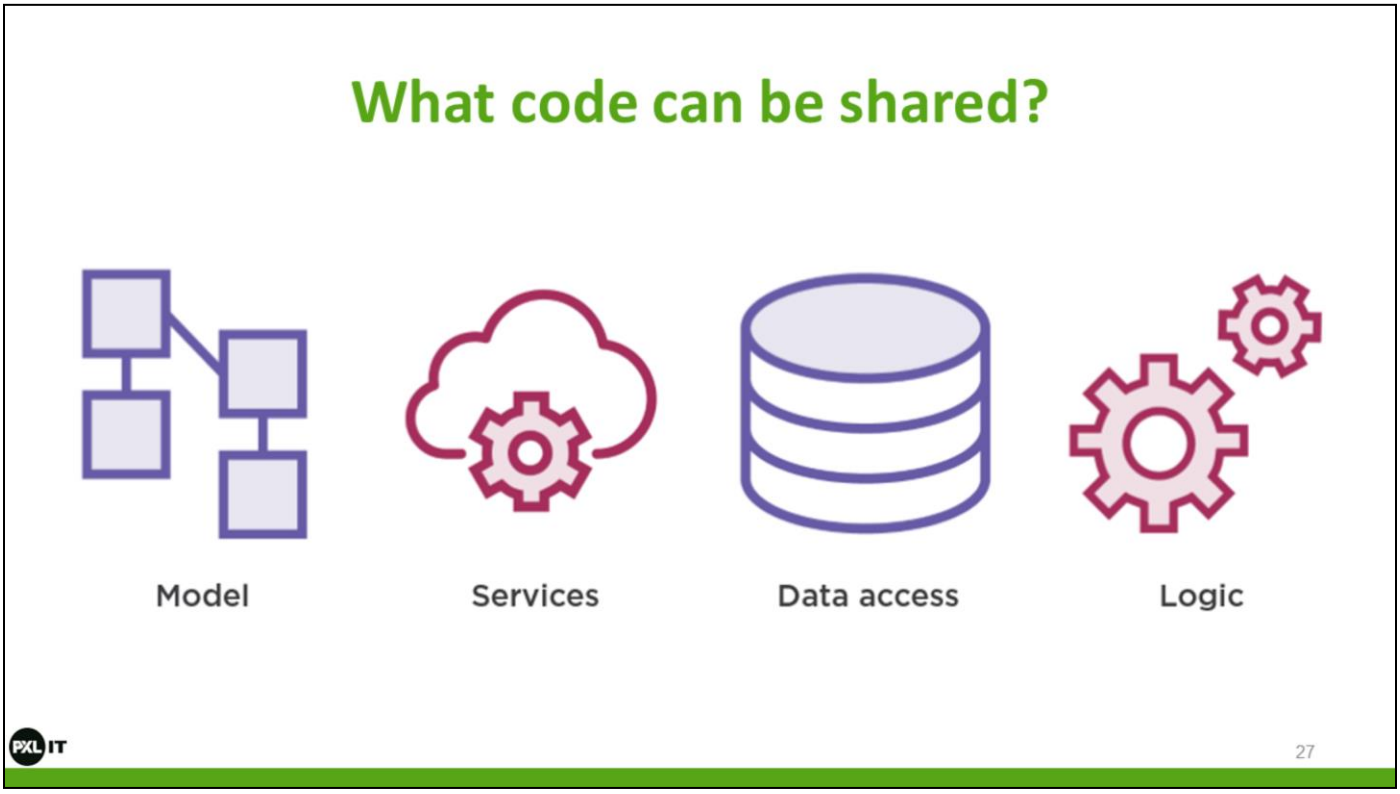
This will definitely **bring down the time to create the application** quite dramatically.

Our time to market when targeting multiple platforms will also be reduced, and **when things go faster, they will cost less**.

No longer will we need to code the same functionality in several languages. Time saved, and thus money saved. Also, when we have a **bug in the shared code, it'll be easier to solve it**.

We have to solve it only once, and the different apps using the shared code will all be fixed at the same time.

The same goes, of course, **for new features**. When the customer is asking for some new feature, we only have to write that functionality once, and it will become available for the different applications.



The parts basically lower in the stack can be shared, so **models** typically can be shared.

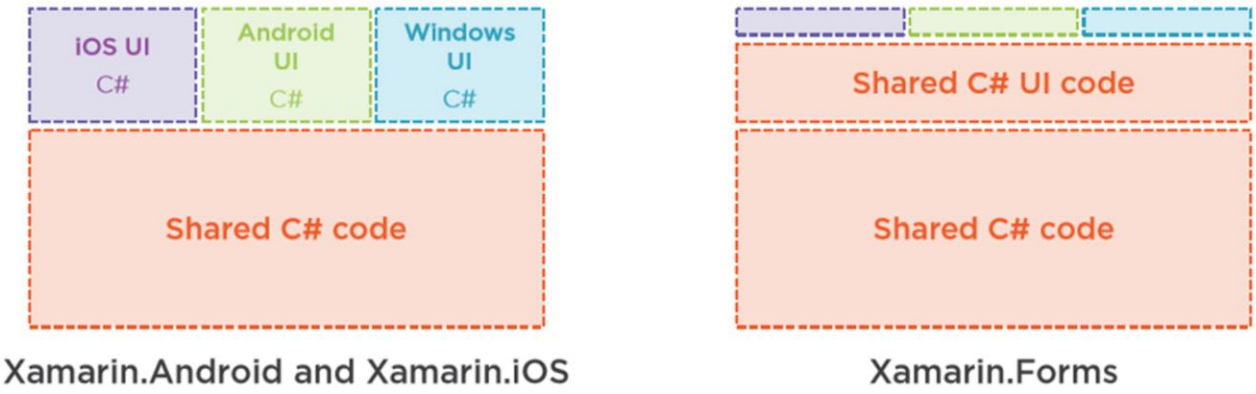
They have no dependency on the platform that we're using the code on, so they can be shared.

Working with **services**, so requesting data from an API, that also can typically be shared.

Data access code typically can also be shared. This includes code that stores and retrieves data from a local database, for example, a SQLite database.

I've also added the term **logic** here, a very general term indeed. What I mean here is that typically business logic can also often be shared.

Code-sharing in Xamarin



PXL IT

28

In **classic Xamarin**, the code sharing basically stops before the UI level.

Only the code that I've just mentioned in the previous slide can in those types of applications be shared.

Xamarin.Forms, as we have seen in the previous module, also makes it possible to share the UI code.

Through XAML and the Xamarin.Forms abstraction framework, we can write the UI once, and the framework will take care of creating the UI specifics for the platform.

Not everything can be shared

Abstractions can be created and used from shared code.
Platform specific implementations will be injected by means of a DI container

PXL IT

29

Now if you think about it, as soon as you start interacting with the **platform-specific service**, you can't share that code any longer.

Take, for example, interacting with the camera. That is an API which is specific for the platform, so iOS, Android, and Windows definitely do this in a different way.

So that code can't be shared in Xamarin apps, not even in a Xamarin.Forms app.

Another example would be working with contacts.

Different platforms work differently when interacting with the contacts. They are stored differently and accessing them is really bound to the platform as well.

Again that's something that we can't do from shared code.

The same goes, for example, for working with Bluetooth, which is again bound to the platform.

Basically as soon as we start **interacting with a system service, that code itself cannot be shared**.

Now you may be thinking, does this then become a problem?

What if I want to interact with Bluetooth from shared code? Is that then entirely impossible?

Well, yes and no, let me explain.

You can’t share the code, but you can however add an abstraction, **an interface**, let's say, **on top of the platform-specific functionality**.

That **abstraction can** then **be used from the shared code**.

Then the applications themselves need to inject an implementation of that abstraction.

In our example here, the Android application needs to plug in an implementation of the interface to interact with Bluetooth, and the iOS application would need to inject an implementation as well.

Most of the time, this is done through **dependency injection**, the implementation is injected into the shared code using a dependency injection container.

Different approaches to share code

- Shared Projects
- **.NET Standard Libraries**
  - The way to go!
- NuGet packages
  - Very general mechanism → not discussed any further
- ~~Portable Class Libraries (PCL)~~
  - Deprecated → replaced by .NET Standard libraries

https://docs.microsoft.com/xamarin/cross-platform/app-fundamentals/

PXL IT

30

https://docs.microsoft.com/xamarin/cross-platform/app-fundamentals/

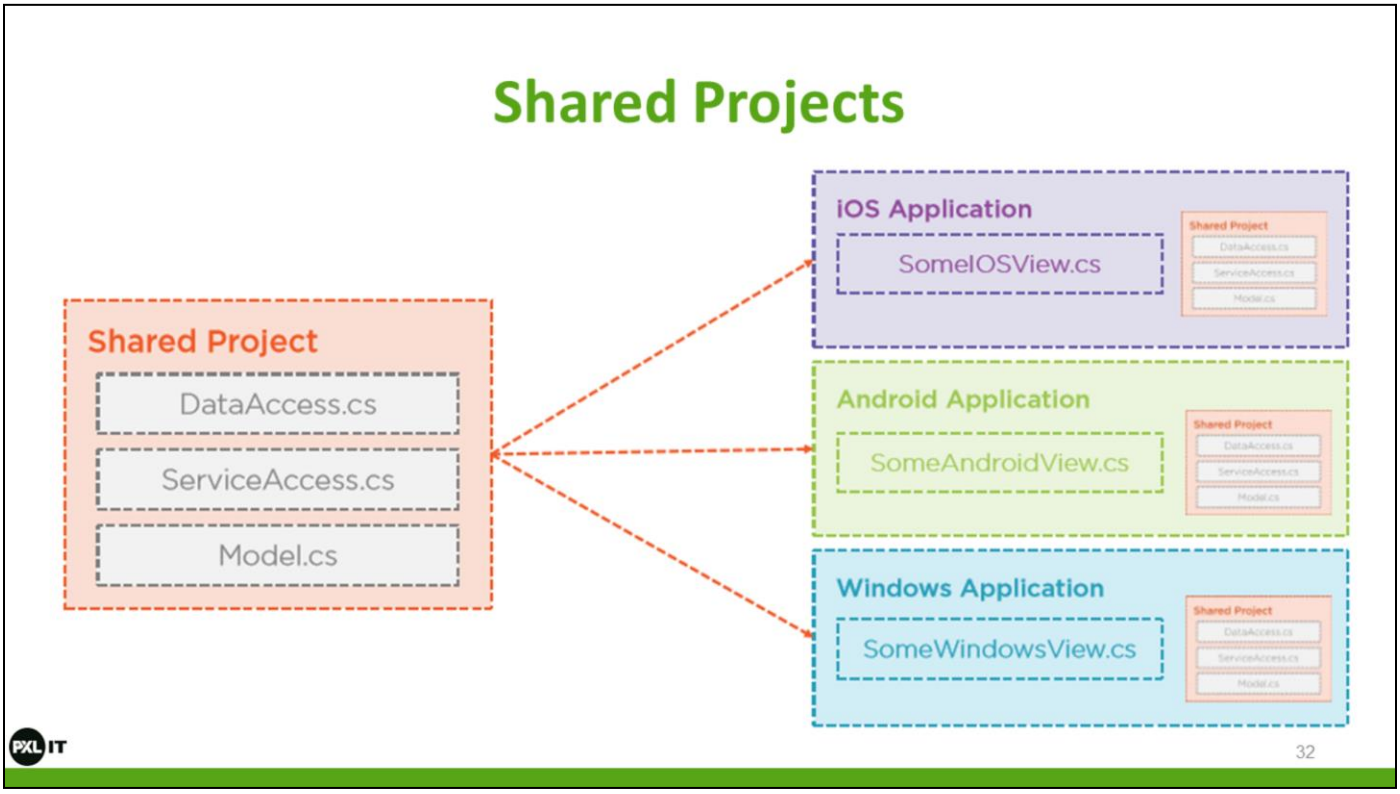
Shared Projects

- Allow sharing code on project level
- Only one copy of a file exists
- Copied to each project upon compilation

PXL IT

31

https://docs.microsoft.com/xamarin/cross-platform/app-fundamentals/shared-projects



On the left, we have our code that we'll be sharing across our applications.

It is a shared project, and it contains our DataAccess code, our ServiceAccess code, and our Models.

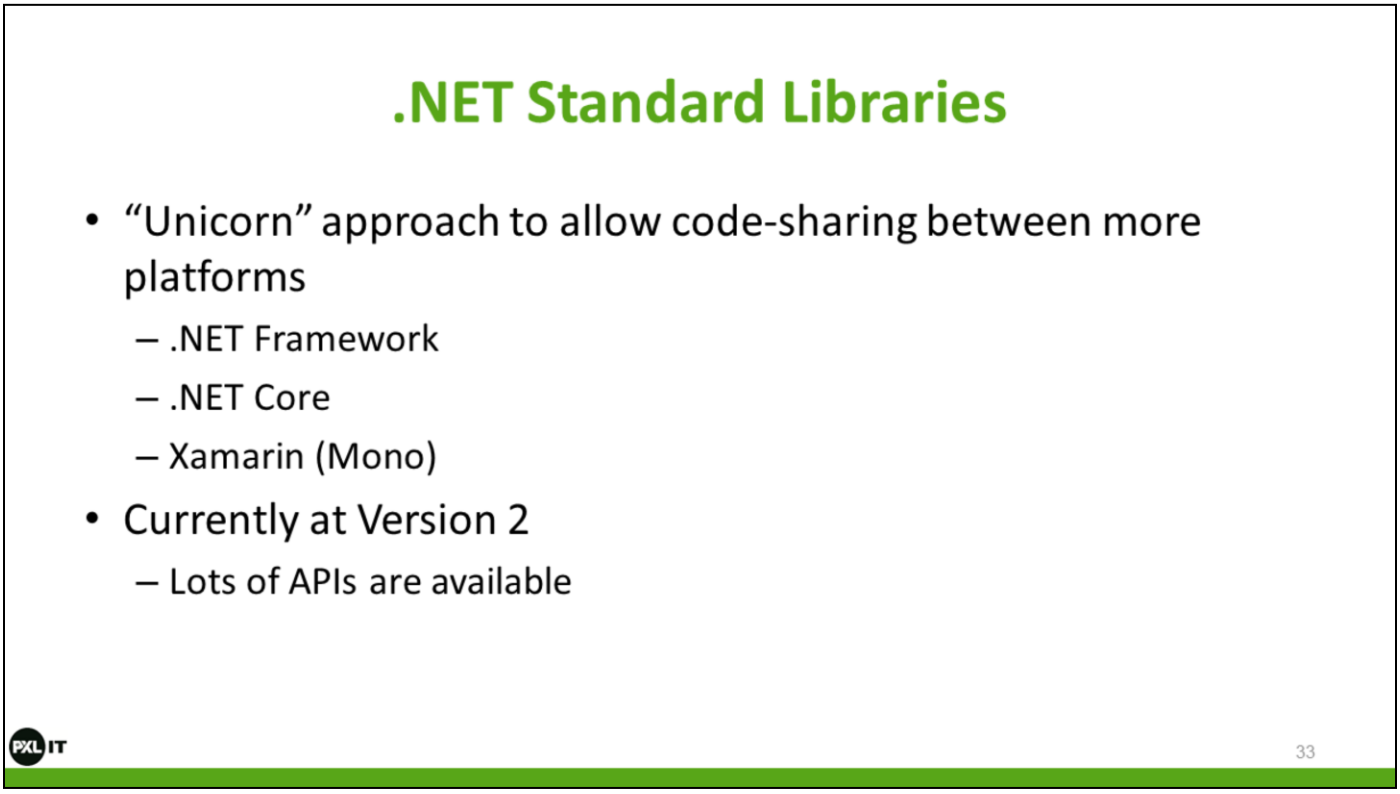
So nothing platform specific is included here. So this code can be shared across each platform.

On the right, we have our different application heads for iOS, Android, and Windows, and each of these apps needs the functionality in the shared project.

So what we can do now, since this is a shared project, is creating a reference from the different projects to the shared project.

Upon compilation, however, the shared project will not be compiled. Instead, the **code in the shared project is copied to the different targets and included there for compilation**.

So the **shared project doesn't result in an assembly, a DLL** being created.



Now through .NET Standard, we can now write code in a .NET Standard library, which can be referenced from more platforms more easily.

.NET Standard allows us to share code between the .NET Framework, .NET Core, and Xamarin projects.

So you can now write a library that can be shared not only between Xamarin applications, but even with an ASP.NET Core application.

I'm not going to give you full overview of .NET Standard, other courses here on Pluralsight exist for that very purpose.

However, I'm going to add that .NET Standard really is the way forward for sharing code also for Xamarin applications.

.NET Standard is currently at version 2, and with that version, a lot of new APIs have been made available.

https://docs.microsoft.com/dotnet/standard/net-standard

https://docs.microsoft.com/xamarin/cross-platform/app-fundamentals/net-standard

Demo: create a .NET Standard Library

- Pluralsight Demo

PXL IT

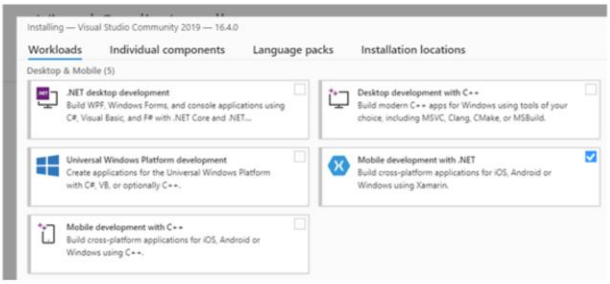
34

Demo: CodeSharingDemo

PREPARING FOR XAMARIN DEVELOPMENT

What you need for Xamarin development

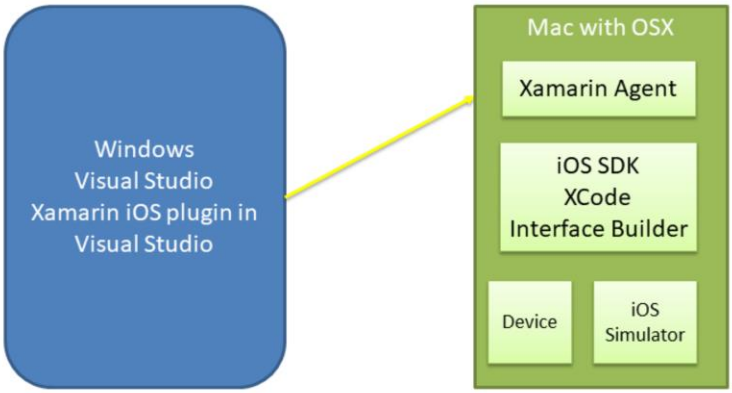
- PC or Mac
- For Android development
  - Visual Studio
    - Android SDK and Emulators
  - Optionally, an android device
- For iOS development
  - Visual Studio
  - Or Visual Studio for Mac
  - A device running Mac OS X (10.11 or higher)
    - Apple's Xcode
    - If you are running Visual Studio on Windows, the Windows computer must be able to reach the Mac via network
  - An Apple developer account



https://docs.microsoft.com/en-us/xamarin/get-started/installation

iOS development on Windows

- Required: a Mac with latest OSX!
  - There are no iOS simulators on Windows!



PXL IT

https://docs.microsoft.com/xamarin/ios/get-started/installation/windows/connecting-to-mac/

iOS development on Windows

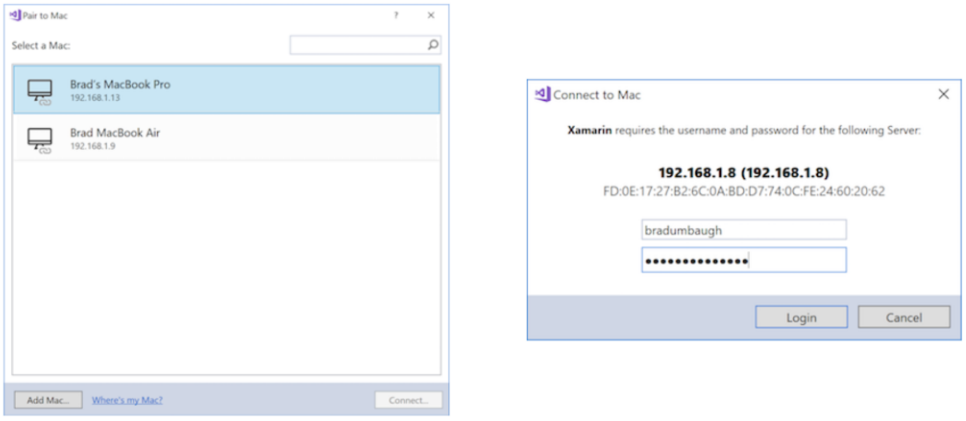
- Enable remote login on the Mac
  - Configure macOS firewall if prompted

PXL IT

https://docs.microsoft.com/xamarin/ios/get-started/installation/windows/connecting-to-mac/

Connecting Windows with your Mac

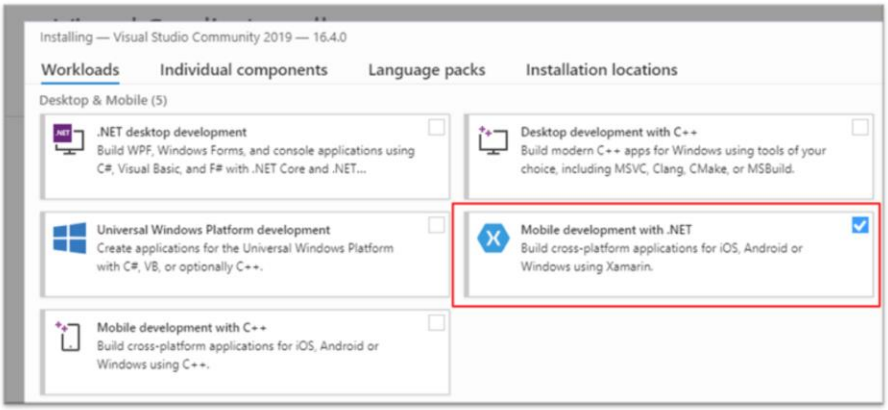
- Open “Pair to Mac” dialog in VS2019
  - Tools > iOS > Pair to Mac



https://docs.microsoft.com/xamarin/ios/get-started/installation/windows/connecting-to-mac/

Android development on Windows

- Visual Studio Installer installs all the needed tools
- Can be done entirely on Windows

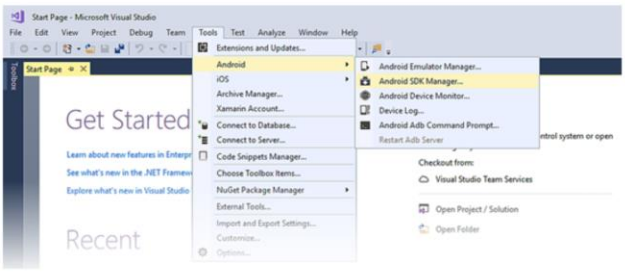


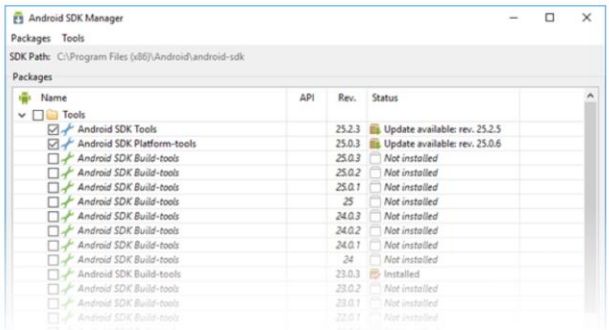
PXL IT

https://docs.microsoft.com/xamarin/android/get-started/installation/windows

Android SDK Manager

- Manage Android SDK components
  - By default Visual Studio installs the Google Android SDK Manager





https://docs.microsoft.com/xamarin/android/get-started/installation/windows

Emulators

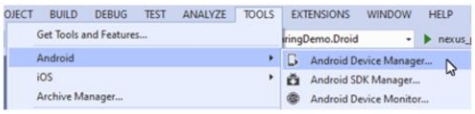
- Emulate a physical device
  - Used when no physical device is available
- 3 components:
  - Google Android Emulator: creates a virtualized device running on the developer's workstation
  - Emulator Image: template (specification) of the hardware and operating system (e.g. Nexus 5x hardware running Android 9.0)
  - Android Virtual Device (AVD): an emulated Android device created from an emulator image

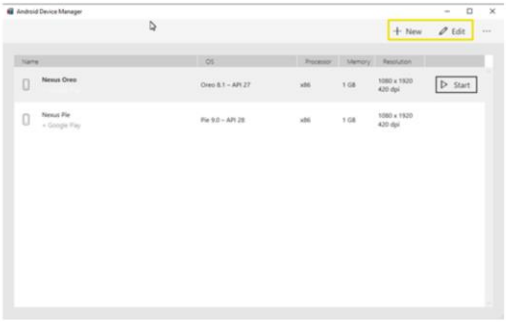
=> The Android Emulator starts a specific AVD based on a certain Emulator Image

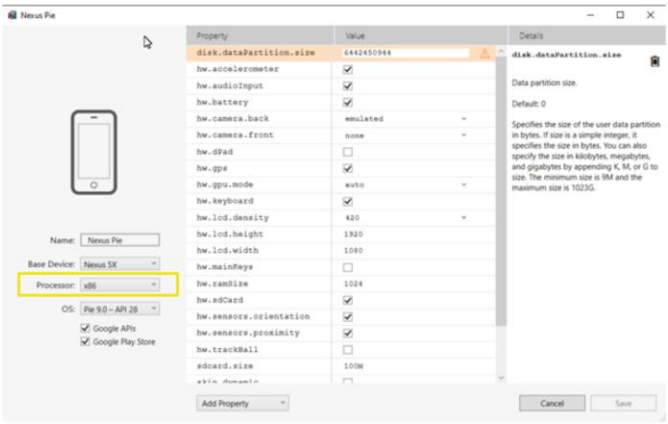
PXL IT

https://docs.microsoft.com/xamarin/android/get-started/installation/windows

Emulators


OJECT BUILD DEBUG TEST ANALYZE TOOLS EXTENSIONS WINDOW HELP
Get Tools and Features...
Android
iOS
Archive Manager...
Android Device Manager...
Android SDK Manager...
Android Device Monitor...


Android Device Manager
New
Edit
Start

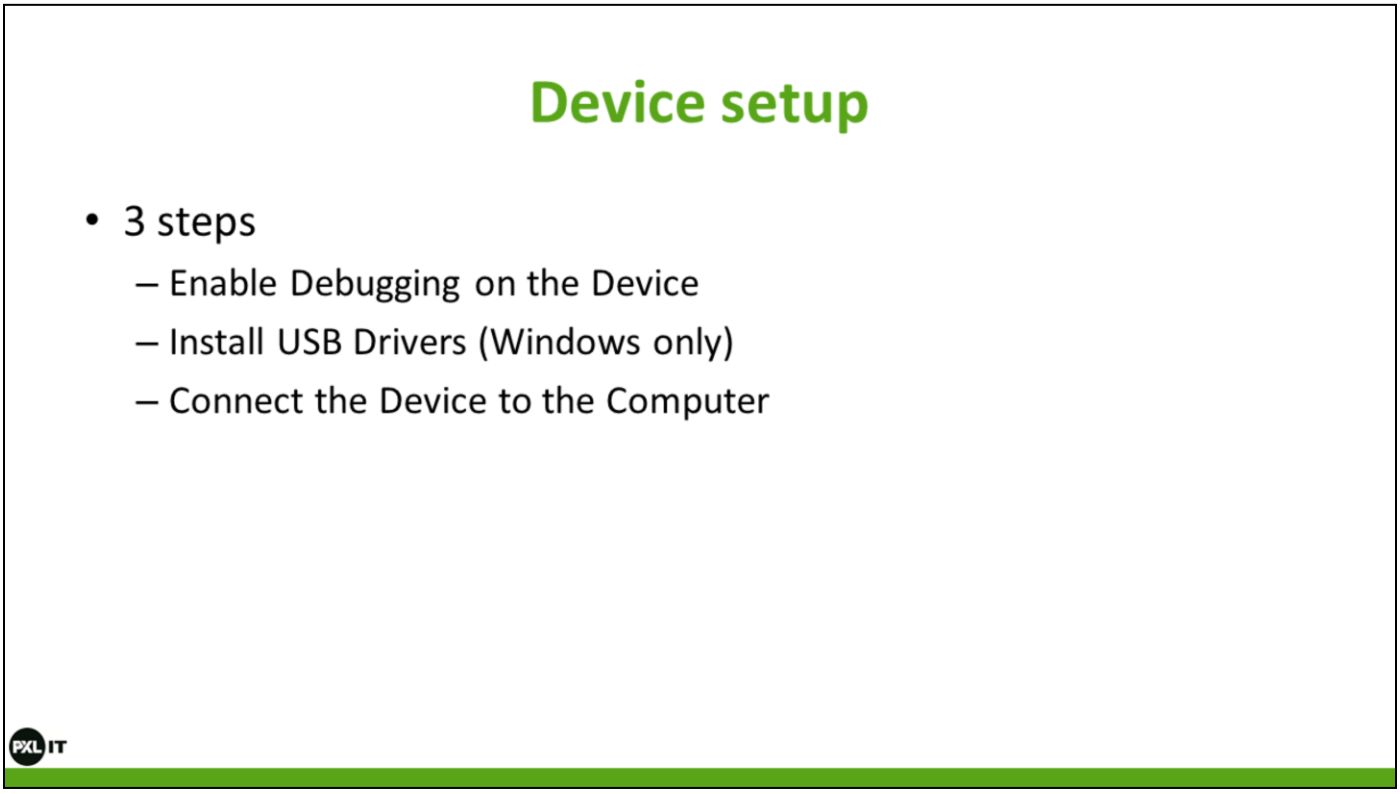

Nexus Pie
Property
Value
Details
Name: Nexus Pie
Base Device: Nexus 5X
Processor: x86
OS: Pie 9.0 – API 28
Google APIs
Google Play Store
Add Property
Cancel
Save

Hardware acceleration for emulators

- Emulators are notoriously slow
- You can improve performance by using X86 emulator images in conjunction with the virtualization features of your computer
  - Hyper-V: virtualization feature of Windows (recommended if available)
  - HAXM: virtualization engine for computers running Intel CPU's (use if Hyper-V is not available)
- Configurations instructions can be found in the Xamarin documentation

PXL IT

https://docs.microsoft.com/xamarin/android/get-started/installation/android-emulator/hardware-acceleration



Enable Debugging on the Device - By default, it will not be possible to debug applications on a Android device.

Install USB Drivers - This step is not necessary for OS X computers. Windows computers may require the installation of USB drivers.

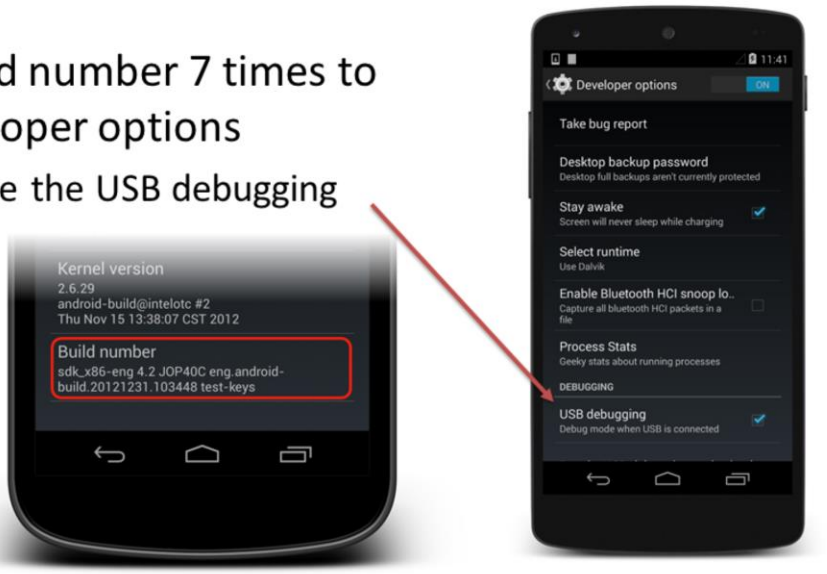
Connect the Device to the Computer - The final step involves connecting the device to the computer by either USB or WiFi.

References:

https://docs.microsoft.com/en-us/xamarin/android/get-started/installation/set-up-device-for-development

Enable Debugging

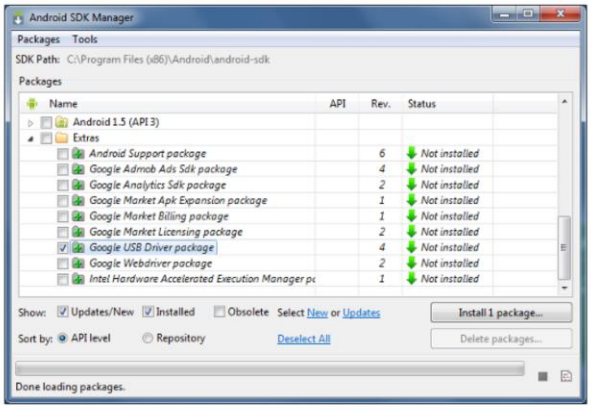
- Tap the Build number 7 times to reveal developer options
  - Then enable the USB debugging option



PXL IT

https://docs.microsoft.com/en-us/xamarin/android/get-started/installation/set-up-device-for-development

Get the USB drivers



References:

https://docs.microsoft.com/en-us/xamarin/android/get-started/installation/set-up-device-for-development

https://developer.android.com/studio/run/oem-usb.html#Drivers

Developing for Android

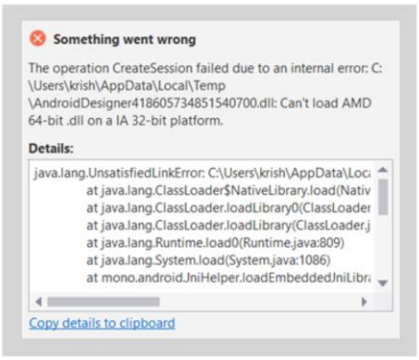
TRY IT YOURSELF - PHONEWORD

Students try to follow the android quickstart:

https://docs.microsoft.com/en-us/xamarin/android/get-started/hello-android/hello-android-quickstart

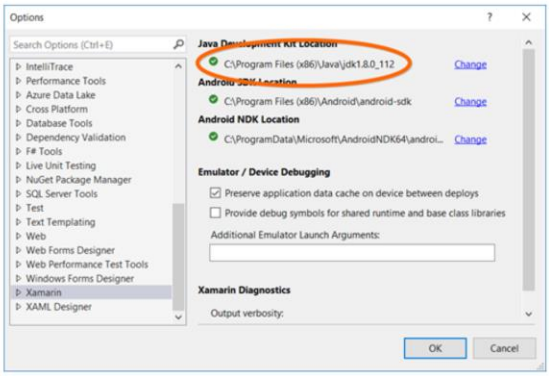
Gotcha: Xamarin needs 64-bit Java

- Android Designer fails to load

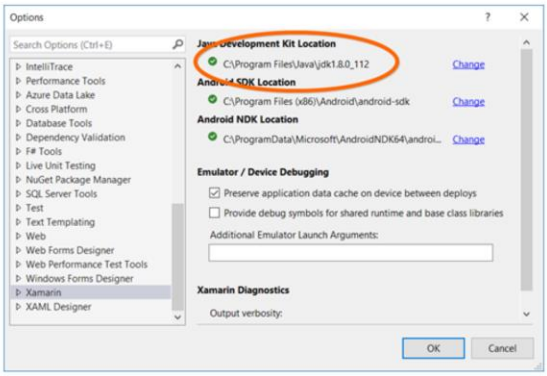


Solution: Point to a 64-bit Java (JDK) installation

- Tools > Options



32 bit JDK: wrong



64 bit JDK: correct

PXL IT

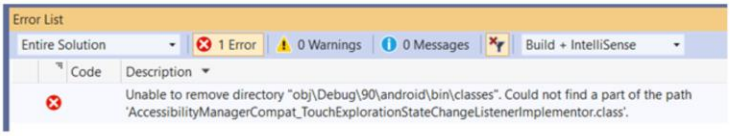
50

32-bit JDK is default installed in C:\Program Files (x86)\Java

64-bit JDK is default installed in C:\Program Files\Java

Gotcha: Could not find a part of the path

- Problem:
  - Android project does not compile. The error mentions a path that cannot be found.



- Solution:
  - The physical path of the solution is too long. Put the solution higher up in the directory hierarchy.