

Modernizing Desktop Apps on Windows 10 with .NET Core 3.0



Miguel Angel Castejón Domínguez

Modernizing Desktop Apps on Windows 10 with .NET Core 3

PUBLISHED BY

Microsoft Press and Microsoft DevDiv

Divisions of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2019 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced in any form or by any means without the written permission of the publisher.

This book is available for free in the form of an electronic book (eBook) available through multiple channels at Microsoft such as <http://dot.net/architecture>

If you have questions related to this book, email at dotnet-architecture-ebooks-feedback@service.microsoft.com

This book is provided “as-is” and expresses the author’s views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at <http://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Co-Author:

Olia Gavrysh, Program Manager, .NET team, Microsoft

Miguel Angel Castejón Dominguez, Innovation Architect, Kabel

Participants and reviewers:

Miguel Ramos, Sr. Program Manager, Windows Developer Platform team, Microsoft

Adam Braden, Principal Program Manager, Windows Developer Platform team, Microsoft

Maira Wenzel, Sr. Content Developer, Microsoft

Andy De Gorge, Sr. Content Developer, Microsoft

Ricardo Minguez Pablos, Sr. Program Manager, Azure IoT team, Microsoft

Nish Anil, Sr. Program Manager, .NET team, Microsoft

Beth Massi, Sr. Product Marketing Manager, Microsoft

Marta Fuentes Lara, Kabel

Raúl Fernández de Córdoba, Kabel

Antonio Manuel Fernández Cantos, Kabel

Scott Hunter, Partner Director PM, .NET team, Microsoft

Contents

Why Modern Desktop Applications?	1
Introduction.....	1
Desktop applications nowadays.....	2
A tale of two platforms.....	5
Paths to modernization	7
What this guide does not cover	8
Who should use this guide	8
How to use this guide	8
Sample apps.....	9
Send us your feedback.....	9
What's new with .NET Core 3.0 for Desktop?	10
From .NET Framework to .NET Core. The motivation behind .NET Core	10
Introduction to .NET Core.....	11
.NET Framework vs .NET Core.....	13
.NET Standard versus PCL.....	15
.NET Core 3.0 new Desktop features	15
Benefits of Open Source	16
Migrating Modern Desktop Applications	18
Configuration files	18
Accessing Databases.....	20
Consuming services	21
Consuming a COM Object	22
More things to consider.....	22
Windows 10 Migration	24
Introduction.....	24
WinRT APIs.....	24
How to add WinRT APIs to your desktop project	27
XAML Islands.....	33
Migrating an example desktop application to .NET Core 3.0	38

Introduction.....	38
Migration Process Overview	38
Migrating a Windows Forms application.....	42
Migrating a WPF Application	48
Deploying Modern Desktop Applications	50
Introduction.....	50
The modern application lifecycle.....	50
MSIX: The next generation of deployment.....	51
How to create a MSIX package from an existing Win32 desktop application	54
Auto Updates in MSIX.....	60

Why Modern Desktop Applications?

Introduction

A story of one company

Back in early 2000's one multinational company started developing a distributed desktop solution to exchange information between different branches of the company and execute optimized operations on centralized units. They have chosen a brand-new framework called Windows Forms for their application development. Over years the project evolved into a mature well tested and time proven application with hundreds of thousands of lines of code. Time passed and .NET Framework 2.0 is no longer the hot new technology. The developers who are working on this application are facing a dilemma. They would like to use the latest stack of technologies in their development and have their application look and "feel" modern. At the same time, they don't want to throw away the great product they have built over 15 years and rewrite the entire application from scratch.

Your story

You might find yourself in the same boat, where you have mature Windows Forms (also known as WinForms) or WPF (Windows Presentation Foundation) applications that have proved their reliability over the years. You probably want to keep using these applications for many more years. At the same time, since those applications were written some time ago, they might be missing capabilities like modern look, performance, integration with new devices and platform features and so on, which gives them a feel of "old tech". There is another problem that might concern you as a developer. While working on the older .NET Framework versions and maintaining applications that were written a while ago, you might feel like you aren't learning new technologies and missing on building modern technical skills. If that is your story – this book is for you!

About this guide

This guide is about strategies you can adopt to move your existing desktop applications through the path of modernization and incorporate the latest runtime, language and platform features. You will discover that there is no unique recipe as each application is different, and so are your requirements and preferences. The good news is that there are common approaches you can apply to add new features and capabilities to your applications. Some of them will not even require major modifications of your code. In this book, we will reveal how all those features work behind the scenes and explain the mechanics of their implementations. Moreover, you will find some common scenarios for modernizing existing desktop applications depicted in detail so you can find inspiration for evolving your projects.

Microsoft approach to modernizing existing applications is to give you the flexibility to create your own customized path. All the modernization strategies described in this book are mostly independent. You can choose ones that are relevant for your application and skip others that aren't important for you. In other words, you will be able to mix and match the strategies to best address your application needs.

Desktop applications nowadays

Before the rise of the Internet, desktop applications were the main approach to build software systems. You could choose any programming language like Cobol, Fortran, Visual Basic, C++, etc., but you ended up creating some sort of desktop application, from small tools to complex distributed architectures.

Then, Internet technologies started shocking the development world and winning over more and more engineers with advantages like easy deployment and simplified distribution processes. The fact that once Web application was deployed to production all users got automatic updates made a huge impact on the software agility.

However, the Internet infrastructure, underlying protocols and standards like HTTP and HTML were not design for building complex applications. In fact, the major development effort back then was aiming just one goal: to give Web applications same capabilities that desktop applications have, such as fast data input, state management and so on.

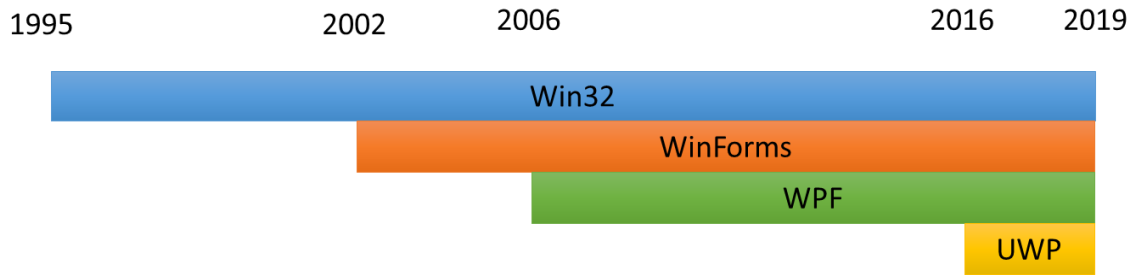
Even though Web and Mobile applications have grown at an incredible pace, for certain tasks desktop applications still hold the number one place in terms of efficiency and performance. That explains why there are millions of developers who are building their projects with WPF and WinForms and the amount of those applications is constantly growing.

Here are some reasons for choosing desktop applications in your development:

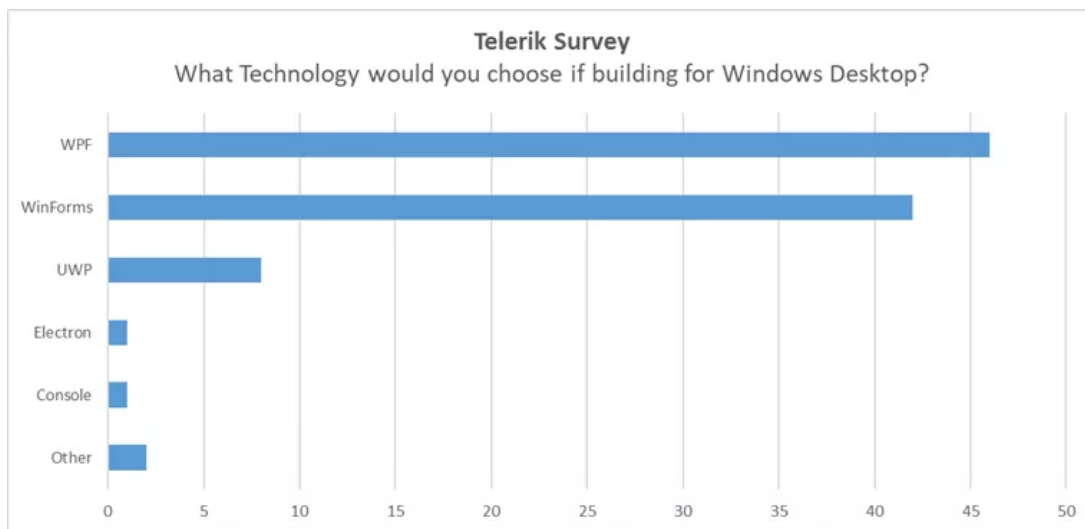
- Desktop apps have better interaction with user's PC.
- The performance of desktop applications for complex calculations is much higher than performance of web applications.
- Running custom logic on the client side is possible but much harder with a web application.
- Using multithreading is easier and more efficient in a desktop application.
- The learning curve for designing User Interfaces is smother and in case of WinForms is completely intuitive with drag-and-drop experience of the Windows Forms designer.
- It is easy to start coding and testing your algorithms without the need to set up a server infrastructure or to care about connectivity problems, firewalls and browser compatibility.
- Debugging is powerful as compared to web debugging.
- Access to hardware devices like camera, Bluetooth or card readers is easy.
- And since the technology have been around for a long time there is a huge human expertise and knowledge database available on developing desktop applications.

So, as you can see, developing for desktop is great for many reasons. The technology is mature and time tested, the development cycle is fast, the debugging is powerful and arguably, desktop apps have less complexity and easier to get started with.

Microsoft offered a variety of UI desktop technologies through the years from Win32 introduced in 1995 and to UWP released in 2016.



The most popular technologies for building Windows Desktop according to a survey published by Telerik on April 2016 are Windows Forms, WPF and UWP.



You can develop in any of them using C# and VB.NET, but let's take a closer look.

Windows Forms

First released in 2002, Windows Forms is a managed framework and is the oldest, most used, desktop technology built on the Windows GDI engine. It offers a very smooth drag-and-drop experience for developing user interfaces in Visual Studio. At the same time, Windows Forms relies on the Visual Studio Designer as the main way you develop your UI, so creating visual components from code is not trivial.

- Mature technology with lots of code samples and documentation.
- Very powerful and productive designer. Not so convenient to design UI "from code".
- Easy and intuitive to learn, thanks to the designer's drag and drop experience.
- Supported on any Windows version.
- Supported on .NET Core 3.0.

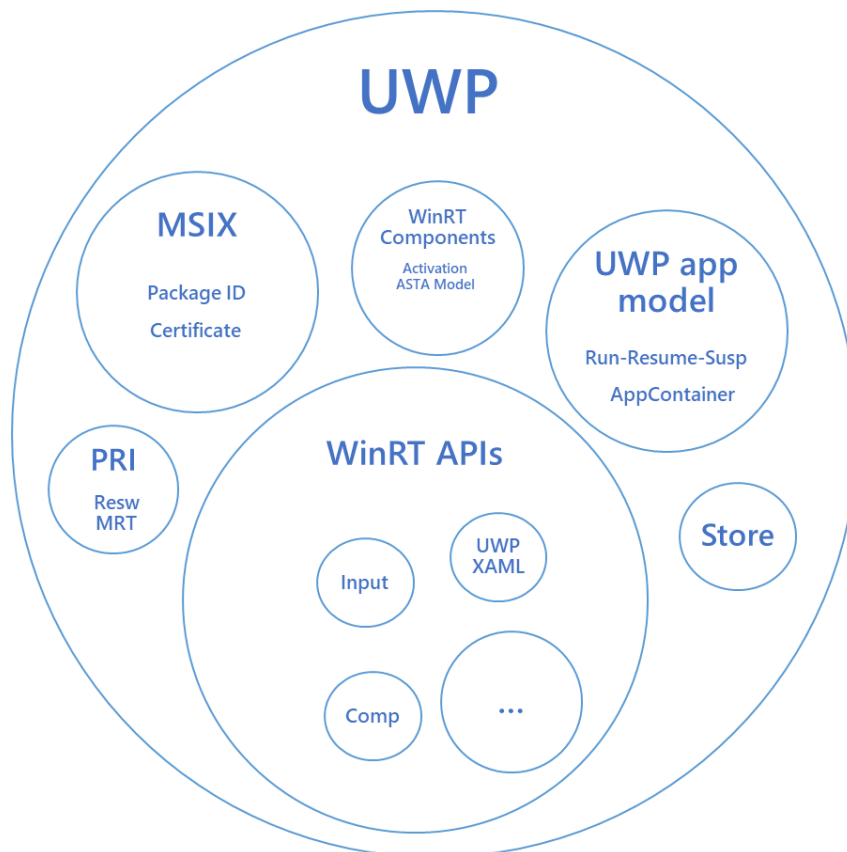
WPF

Based on the XAML language specification, WPF favors a clear separation between UI and code. XAML offers such capabilities like templating, styling, and binding, which is suited for building big applications. Like Windows Forms, it is a managed framework, but the design is modular and reusable.

- Mature technology.
- Designer is available, but developers usually prefer to create the design from code using declarative XAML.
- The learning curve is steeper than Windows Forms.
- Supported on any Windows version.
- Supported on .NET Core 3.0.

UWP

The Universal Windows Platforms is not only a presentation framework like WPF and Windows Forms, but it is also a platform itself. This platform has its own API set (the Windows Runtime API), a new deployment system (MSIX), a modern application lifecycle model (for low battery consumption), a new Resource Management System (based on PRI files), among other things. The platform was created to support all kind of input systems (like ink, touch, gamepad, mouse, keyboard, gaze, etc.) in all form-factors with performance and low battery consumption in mind. For these reasons the Shell of the Windows 10 OS uses parts of the UWP platform.



UWP contains a presentation framework that is XAML based, like WPF, but it has some important differences such as:

- Applications are executed in app containers. App containers are a sandbox mechanism which control what resources an UWP app can access or not.
- Supported only on Windows 10.
- Apps can be deployed through Microsoft Store for easier deployment besides sideload.
- Designed as part to the Windows Runtime API.
- Contains an extensive set of built-in rich controls and additional controls in the Microsoft UI Library NuGet packages (WinUI library) updated every few months.

A tale of two platforms

In the last 20 years, while UI desktop technologies were growing and following the path from Windows Forms to UWP, the hardware was also evolving from heavy weight PC units with small CRT monitors to HDPI monitors and lightweight tablets and phones with different data input techniques like Touch and Ink. This resulted in creating two different concepts: a Desktop Application and a Modern Application. A Modern Application is one that considers different device form factors, various input and output methods, and leverages modern desktop features while running on a sandboxed execution model. The (traditional) Desktop Application, on the other hand, is an application that needs a solid UI with high density of controls that is best operated with a mouse and a keyboard.

We can describe the differences between the two concepts in the following table:

	Modern Application	Desktop Application
Security	Contained execution & Great Fundamentals. Design from the ground up to respect user privacy, manage battery life and focus to keep the device safe.	User & Admin level of security. You have native access to the Registry and Hard Drive folders.
Deployment	Installation and updates are managed by the platform.	MSI, Custom installers & Updates. Traditionally a source of headaches for developers and IT managers.
Distribution	Trusted Distribution & Signed Packages. Distribution is performed from a trusted source and never from the Web.	Web, SCCM & Custom distribution. No control over what is installed, affects the whole machine.

UI	Modern UI. Different input mechanisms, ink, touch, gamepad, keyboard, mouse, etc.	Windows Forms, WPF, MFC. Designed for the mouse and keyboard for a very dense UI and to get the most productivity from the desktop.
Data	Cloud First Data with Insights. Source of truth in the cloud. Insights to know what happens with your app and how it is performing.	Local Data. Traditional desktop applications usually need some local data.
Design	Designed for reuse. Reuse in mind between different platforms, front end and back end, running assets in many places as possible.	Designed for Windows Desktop only

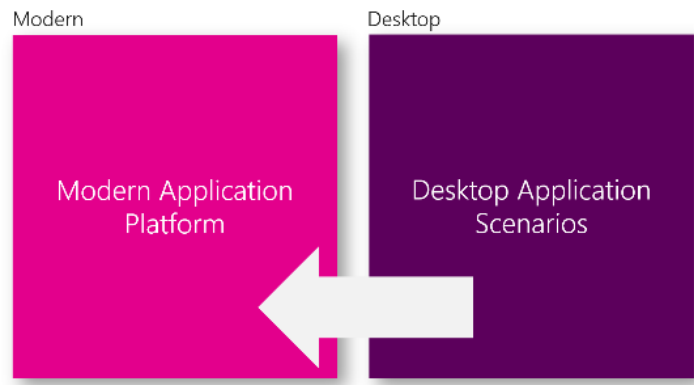
As a part of the commitment to provide developers with the best tools to build applications, Microsoft has performed a great effort to bring these concepts, or we can even say platforms, closer together to empower developers with the best of both worlds. To do that, Microsoft has performed a bidirectional effort between the two platforms.

Bring both platforms closer together

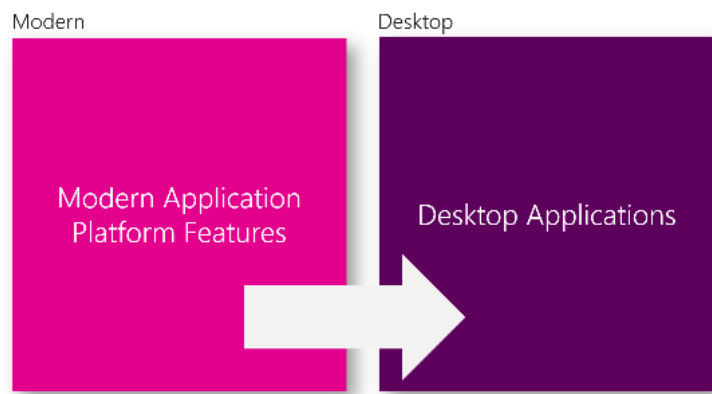


1. Move Desktop Application scenarios into Modern Application platform. The traditional desktop development is still very popular because it addresses certain scenarios really well. It

makes sense to take these common desktop scenarios and bring them into the modern desktop platform to make the platform fully capable.



2. Move Modern Application features into Desktop Applications. For existing desktop apps that need a way to leverage modern capabilities without rewriting from scratch, features from the Modern Application platform are pushed into the Desktop Application.



In this book we will focus on the second part and show how you can modernize your existing desktop applications.

Paths to modernization

The structure of this guide reflects three different axes to accomplish modernization: Modern Features, Deployment and Installation.

Modern Features

Say you have a working Windows Forms application that a sales representative of your company uses to fill in a customer order. A new requirement comes in to enable the customer to sign-in the order using a tablet pen. Inking is native in today's operating systems and technologies but was not available when the app was developed.

This path will show you how you can leverage modern desktop features into your existing desktop development.

Deployment

Modern development cycles have stressed out to provide agility on how new versions of applications are deployed to every single user. Since Windows Forms and WPF applications are based on a particular version of the .NET Framework that must be present on the machine, they cannot take advantage of new .NET Framework version features without the intervention of the IT people with the risk of having side effects for other apps running on the same machine. This has limited the innovation pace for developers forcing them to stay on outdated versions of the .NET Framework.

Now, with the launch of .NET Core 3.0, you can leverage a radically new approach of deploying multiple versions of .NET Core side by side and specifying which version of .NET Core each application should target. This way, you can use newest features in one application while being confident you aren't going to break any other applications.

Installation

Desktop applications always rely on some sort of installation process before the user can start using them. This brought into the game a set of technologies, from MSI and ClickOnce to custom installers or even XCOPY deployment. Any of these methods deals with delicate problems because applications need a way to access shared resources on the machine. Sometimes installation needs to access the Registry to insert or update new Key Values, sometimes to update shared DLLs referenced by the main application. This causes a continuous headache for users, creating this perception that once you install some application, your computer will never be the same, even if you uninstall it afterwards.

In this book, we will introduce a new way of installing applications with MSIX that solves the problem described above. You will learn how you can easily set up a packaging, installation and updates for your application.

What this guide does not cover

This guide covers a specific subset of scenarios focused on lift and shift scenarios, outlining the way to gain the benefits of modernizing without the effort of rewriting code.

This guide is not about developing modern applications with .NET Core or implementing desktop scenarios into the Modern Application Platform. It focuses on how you can leverage your investments on desktop applications while you keep them updated with some of the latest technologies for desktop development.

Who should use this guide

We wrote this guide for developers and solution architects who want to modernize existing Windows Forms and WPF Desktop Applications to leverage the benefits of .NET Core 3.0 and MSIX installation.

You also might find this guide useful if you are a technical decision maker, such as an enterprise architect or a development lead/director who just wants an overview of the benefits that you can get by updating existing Desktop Apps.

How to use this guide

This guide addresses the “why”—why you might want to modernize your existing applications, and the specific benefits you get from using .NET Core 3.0 and MSIX to modernize your desktop apps. The

content of the guide is designed for architects and technical decision makers who want an overview, but who don't need to focus on implementation and technical, step-by-step details.

Along the different chapters, sample implementation code snippets and screenshot are provided, with chapter 5 devoted to a showcase a complete migration process for sample applications.

Sample apps

To highlight the necessary steps to perform a modernization we will be using a sample application called eShopModernizing. This application has two flavors, Windows Forms and WPF and we will show a step by step process on how to perform the modernization on both of them to .NET Core.

Also, on the GitHub repo for this book, you will find the results of the process, which you can consult with if you decide to follow the step-by-step tutorial.

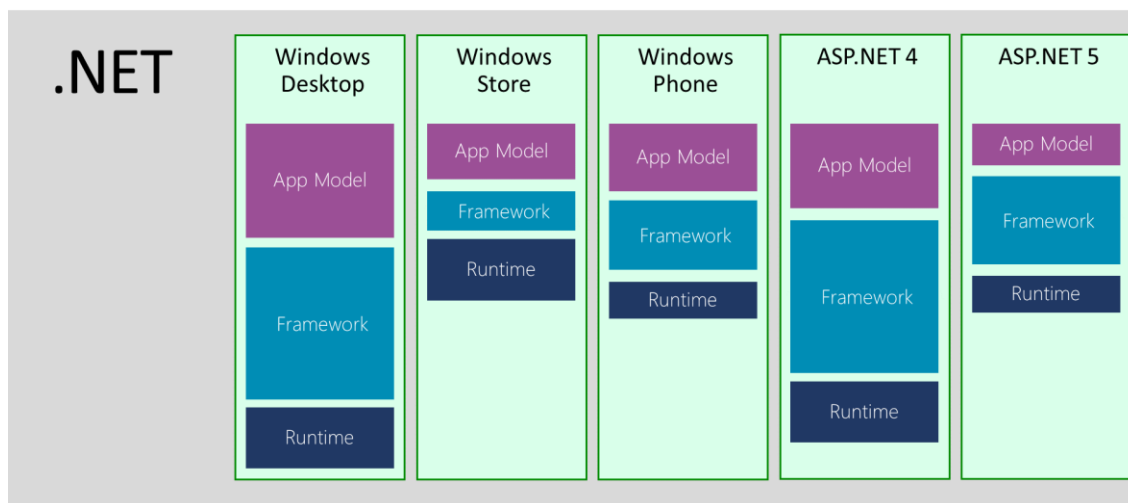
Send us your feedback

We wrote this guide to help you understand your options for improving and modernizing existing .NET desktop applications. The guide and related sample applications are evolving. We welcome your feedback! If you have comments about how this guide might be more helpful, please send them to dotnet-architecture-ebooks-feedback@service.microsoft.com.

What's new with .NET Core 3.0 for Desktop?

From .NET Framework to .NET Core. The motivation behind .NET Core

Since its inception in 2002, .NET Framework has evolved through the years to support many technologies like Windows Desktop, ASP.NET, Entity Framework, Windows Store and many others. All of them are very different in nature. Therefore, Microsoft was approaching this evolution by taking parts of the .NET Framework and creating a different application stack for each technology. That way, development capabilities could be customized for the needs of the specific stack, which maximized the potential of every platform. Of course, that led to fragmentation on the versions of the .NET Framework maintained by different independent teams. All these stacks have a common structure, containing an App Model, a Framework and a Runtime, but they differ in the implementation of each of these parts.



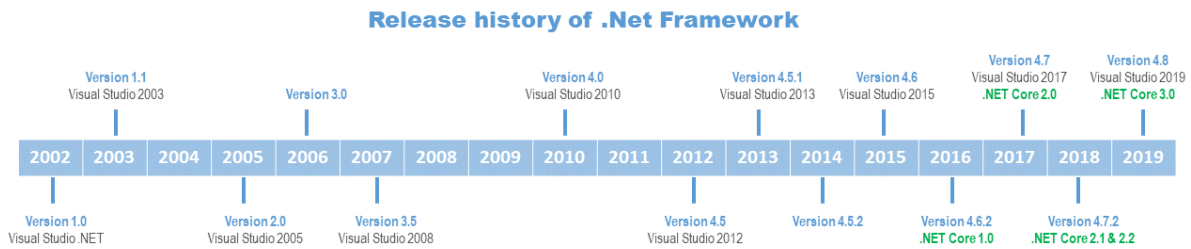
If you are targeting only one of these platforms, you can use this model. However, in many cases you might need more than one target platform in the same solution. For example, your application may have a desktop admin part, a customer-facing web site that shares the backend logic running on a server, and even a mobile client. In this case, you need a unified coding experience that can span all this .NET verticals.

By the time Windows 8 was released, the concept of the Portable Class Libraries (PCL) was born. Originally, the .NET Framework was designed around the assumption that it is always deployed as a single unit, so factoring was not a concern. To face the problem of code sharing between verticals, the driving force was on how to refactor the framework. The idea of contracts is to provide a well factored API surface area. Contracts are simply assemblies that you compile against and are design with proper factoring in mind taking care of the dependencies between them.

This leads to a reasoning about the API differences between verticals at the assembly level, as opposed to the individual API level that we had before. This aspect enabled a class library experience that can target multiple verticals, also known as portable class libraries.

With PCL the experience of development is unified across verticals base on the API shape and the most pressing need to create libraries running on different verticals is addressed. But there is still a

great challenge: APIs are only portable when the implementation is moved forward across all the verticals, and still verticals have independent implementations.



A better approach is to unify the implementations across verticals by providing a well factored implementation instead of a well factored view. It is a lot simpler to ask each team owning a specific component to think about how their APIs work across all verticals than trying to retroactively provide a consistent API stack on top. This is where .NET Standard comes in, see details on next section.

Another big challenge has to do with how the .NET Framework is deployed. The .NET Framework is a machine-wide framework. Any changes made to it affect all applications taking a dependency on it. Although this has many advantages like reducing disk space and centralized access to services, it presents some pitfalls.

To start with, it is difficult for application developers to take a dependency on a recently released framework. You have either to take a dependency on the latest OS or provide an application installer that will install the .NET Framework along with the application. If you are a web developer, you might not even have this option as the IT department establishes the server supported version.

Even if you are willing to go through the trouble of providing an installer in order to chain in the .NET Framework setup, you may find that upgrading the .NET Framework can break other applications.

Despite the efforts to provide backward compatible versions of the framework, there are compatible changes that can break applications. For example, adding an interface to an existing type that can change how this type is serialized and cause breaking problems depending on the existing code. Because .NET Framework installed base is huge, fighting against these breaking scenarios slows down the pace of innovations inside the .NET Framework.

To solve all these issues Microsoft has developed .NET Core to approach the evolution of the .NET Platform.

Introduction to .NET Core

The .NET Core is the evolution of Microsoft's .NET technology into a modular, cross platform, open source and cloud ready platform, which runs on Windows, Mac, and Linux with plans to run also on ARM based architectures like Android and IoT.

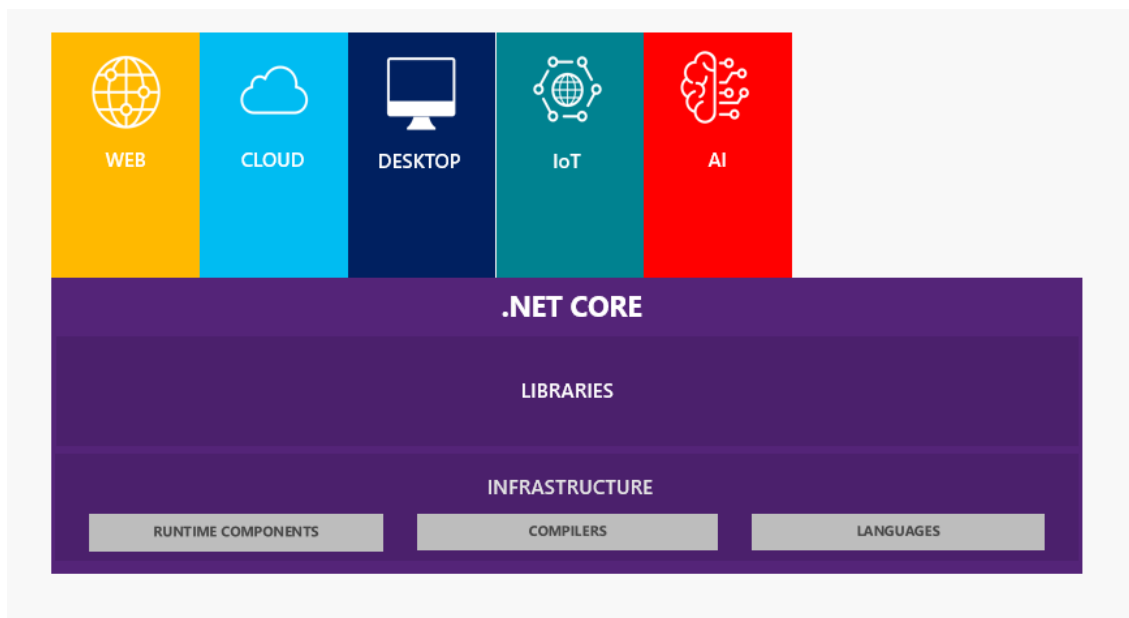
The purpose of .NET Core is to provide a unified platform for all types of applications, which includes Windows, cross platform and mobile applications. .NET Standard library enables this by providing shared base API's, which every application model needs, and excluding any application model specific API's.

This framework gives applications many benefits in terms of efficiency and performance, simplifying the packaging and deployment in the different supported platforms.

The benefits of .NET Core come from these three characteristics:

- **Cross-platform:** It allows application execution on different platforms, Windows, macOS and Linux.
- **Open source:** .NET Core platform is open source and available through GitHub, fostering transparency and community contributions.
- **Strongly supported:** Microsoft officially supports .NET Core.

In .NET Core 3.0, there is planned support for developing the following application types:



The goal for this framework is quite impressive: to target every type of .NET development present and future including Desktop, Web, Cloud, Mobile, Gaming, IoT and AI. Microsoft plans to complete this vision with .NET 5 at the end of 2020. Note that the "Core" name was removed to reinforce its uniqueness in the .NET World.

.NET – A unified platform



.NET Framework vs .NET Core

Therefore, you now understand the relevance of .NET Core inside the Microsoft strategy for .NET and might be wondering what happens with .NET Framework and asking questions like: do you have to abandon it? Is it going to disappear? What are my choices to modernize the applications I have on .NET Framework?

The good news is Microsoft plans to support both .NET Framework and .NET Core. In fact, the announcement of .NET Core 3.0 was followed by exciting features in .NET Framework regarding modernization, so let's take a brief look at both paths.

Listening to the .NET Framework developer's community suggestions Microsoft have addressed three main scenarios for **.NET Core**:

- **Side-by-side versions of .NET supporting Windows Forms and WPF:** This solves the problem of side effects when updating the machine's framework version. With .NET Core multiple versions can be installed on the same machine and for each application you can specify, which version of .NET Core you'd like it to use. Even more, now you can develop and run Windows Forms and WPF on top of .NET Core.
- **Embed .NET directly into an application:** You can now ship the .NET Core as part of your application package. This enables you to take advantage of the latest version, features, and APIs without having to wait for a specific version to be installed on the machine.
- **Take advantage of .NET Core features:** .NET Core is the fast-moving, open source version of .NET. Its side-by-side nature enables fast introduction of new innovative APIs and BCL (Base Class Libraries) improvements without the risk of breaking compatibility. Now Windows Forms and WPF applications on Windows can take advantage of the latest .NET Core features, which also includes more fundamental fixes for runtime performance, high-DPI support, and so on.

Regarding **.NET Framework 4.8**, Microsoft have addressed three scenarios:

- **Modern browser and media controls:** Today, .NET desktop applications use Internet Explorer and Windows Media Player for showing HTML and playing media files. Since these

legacy controls don't show the latest HTML or play the latest media files, we are adding new controls that take advantage of Microsoft Edge and newer media players to support the latest standards.

- **Access to UWP Controls:** UWP contains new controls that take advantage of the latest Windows features and touch displays. You will not have to rewrite your applications to use these new features and controls. Now they are available to Windows Forms and WPF so that you can take advantage of these new features in your existing code.
- **High DPI improvements:** The resolution of displays is increasing to 4K and 8K resolutions. We want to make sure your existing Windows Forms and WPF applications can look great on these new displays. That's why we are adding improvements for HDPI

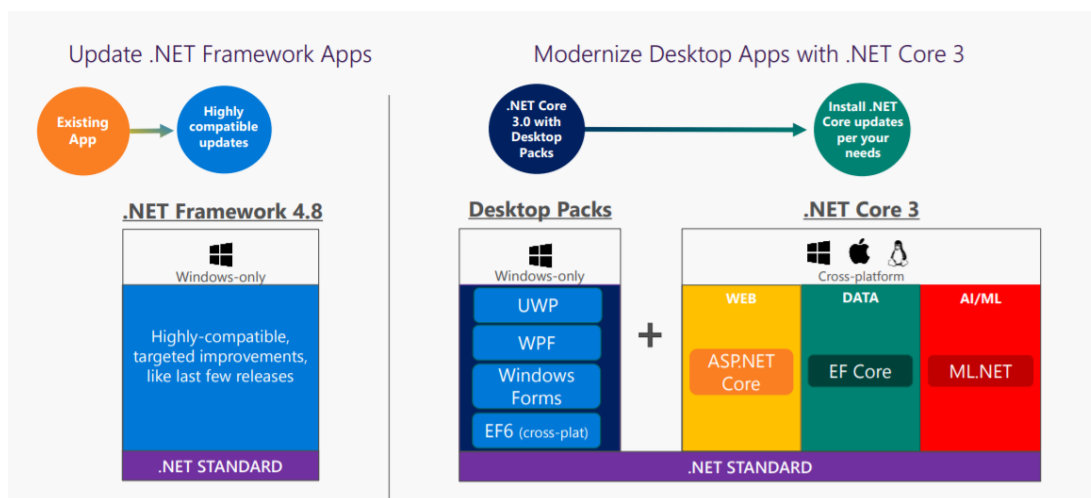
Since .NET Framework is working on millions of machines, Microsoft will continue to support it but will not be adding new features.

.NET Core is the open source, cross-platform, and fast-moving version of .NET. Because of its side-by-side nature, it can take changes without the fear of breaking any application. This means that .NET Core will get new APIs and language features over time that .NET Framework cannot.

An essential part of this roadmap is to ease developers to move applications to .NET Core. .NET Core 3.0 takes a huge step by adding WPF, Windows Forms and Entity Framework 6 support, and we will keep porting APIs and features to help close the gap and make migration easier for those who chose to do so.

So, if you have existing .NET Framework applications, you should not feel pressured to move to .NET Core. .NET Framework will be fully supported and will always be a part of Windows. However, if in the future you want to use the newest language features and APIs, you'll need to move your applications to .NET Core.

For your brand-new desktop applications, we recommend starting directly on .NET Core. It is lightweight, runs side by side, high performance and cross platform and fits perfectly on containers and microservices architectures.



.NET Standard versus PCL

The .NET Standard is a formal specification of .NET APIs that are intended to be available on all .NET implementations. The motivation behind the .NET Standard is establishing greater uniformity in the .NET ecosystem. .NET Standard is a specification of .NET APIs that make up a uniform set of contracts to compile your code against. These contracts are implemented in each .NET flavor, thus enabling portability across different .NET implementations.

The .NET Standard enables the following key scenarios:

- Defines uniform set of base class libraries APIs for all .NET implementations to implement, independent of the workload.
- Enables developers to produce portable libraries that are usable across .NET implementations, using this same set of APIs.
- Reduces or even eliminates conditional compilation of shared source due to .NET APIs, only for OS APIs.

.NET Standard is the evolution of PCLs and here are the main differences between .NET Standard and PCLs:

- .NET Standard is a set of curated APIs, picked by Microsoft, PCLs are not.
- The APIs that a PCL contains is dependent on the platforms that you choose to target when you create a PCL. This makes a PCL only sharable for the specific targets that you choose.
- .NET Standard is platform-agnostic, it can run anywhere, on Windows, Mac, Linux and so on.
- PCLs can also run cross-platform, but they have a more limited reach. PCLs can only target a limited set of platforms.

.NET CORE 3.0 NEW DESKTOP FEATURES

Support for Windows Forms and WPF

Windows Forms and WPF are part of .NET Core 3.0. Both presentation frameworks are only for the Windows OS and they are not cross-platform. You can think of WPF as a rich layer over DirectX and Windows Forms as a thinner layer over GDI+. WPF and Windows Forms do a great job of exposing and exercising much of the desktop application functionality in Windows. With .NET Core 3.0. Windows Forms and WPF become available for .NET Core as well as for .NET Framework. Now you can start your new desktop applications targeting .NET Core and migrate your existing ones from .NET Framework to .NET Core.

A new version of .NET Standard, version 2.1, was released at the same time. As expected, all new .NET Standard APIs are part of .NET Core 3.0.

You can build desktop applications with C#, F#, and VB with .NET Core 3.0.

Also, important to notice that both Windows Forms and WPF implementations for .NET Core 3 are being open sourced.

XAML Islands

XAML Islands is a set of components for developers to use the new Windows 10 controls (UWP XAML controls) in their current WPF, Windows Forms, and native Win32 apps (like MFC). You can have your "islands" of UWP XAML controls wherever you want inside your Win32 apps.

These XAML Islands are possible because Windows 10 1903 update introduces a set of APIs that allows hosting UWP XAML content in Win32 windows using windows handlers (HWnds). Notice that only apps running on Windows 10 1903 and above can use XAML Islands.

To facilitate to create XAML Islands for Windows Forms and WPF developers, the Windows Community Toolkit introduces a set of .NET wrappers in several NuGet packages. Those wrappers are the Wrapped Controls and Hosting Controls:

- The WebView, WebViewCompatible, [InkCanvas](#), [MediaPlayerElement](#), and [MapControl](#) wrapped controls wrap some UWP XAML controls into Windows Forms or WPF controls, hiding UWP concepts for those developers.
- The WindowsXamlHost control for Windows Forms and WPF allows others not-wrapped UWP XAML controls and custom controls can be loaded into a XAML Island.

Access to all Windows 10 APIs

Windows 10 has a great amount of API available for developers to work with. This APIs give access to a wide variety of functionality like Authentication, Bluetooth, Appointments & Contacts. Now these APIs are exposed through .NET Core and give Windows developers the chance to create powerful desktops apps leveraging the capabilities present on Windows 10.

Side-by-side support and self-contained EXEs

The .NET Core deployment model is one of the biggest benefits that Windows desktop developers will experience with .NET Core 3.0. The ability to globally install .NET Core provides much of the same central installation and servicing benefits of .NET Framework, while not requiring in-place updates.

When a new .NET Core version is released, you can update each app on a machine as needed without any concern of affecting other applications. New .NET Core versions are installed in their own directories and exist “side-by-side” with each other.

If you need to deploy with isolation, you can deploy .NET Core with your application. .NET Core will bundle your app with the .NET Core runtime as in a single executable.

These deployment options were request by developers for quite a long time but were difficult to achieve using .NET Framework. The modular architecture used by .NET Core makes these flexible deployment options possible.

Performance

Since its inception, targeting the Web and Cloud workloads, .NET Core has had performance plugged into its DNA. Server-side code must be performant enough to fulfill high concurrency scenarios and .NET Core scores today as the best performance web platform in the market.

You can now take advantage of these performance improvements when you use it to build your next generation of desktop applications.

Benefits of Open Source

Just a few words about .NET Core being open source. Building a cross-platform stack is something very complex that needs the interaction of specialized teams on each of the targeted platforms. This effort needs a lot of collaboration from inside and outside of Microsoft. By making it open source and thus

open to public collaboration, you get the ultimate agile development style in place, raising the quality bar since issues are detected by a huge and active community of developers.

This is a key success factor of .NET Core that will continue to speed up the roadmap mentioned above: To be the single .NET platform that any developer will ever need to build any application.

Migrating Modern Desktop Applications

In this chapter, we are exploring the most common issues and challenges you can face when migrating an existing application from .NET Framework to .NET Core.

A complex desktop application does not work in isolation and needs some kind of interaction with subsystems that may reside on the local machine or on a remote server. It probably connects to a database as persistence storage either local or remotely. With the rise of Internet and service-oriented architectures, it's extremely common to have your application connected to some sort of service residing on a remote server or in the cloud. You may need to access the machine's file system to implement some functionality. Alternatively, maybe you are using a piece of functionality that resides inside of a COM object outside your application. A common scenario is, for example, you are integrating Office assemblies into your app.

Besides, there are differences in the surface of the APIs exposed by .NET Framework and .NET Core and some parts of what is existing on .NET Framework is no longer available on .NET Core, so it's important for you to know and take into account the API differences when planning a migration.

Configuration files

Configuration files offer the possibility to store sets of properties that are read at runtime, affecting the behavior of the application, like where to locate a database or how many times to execute a loop. This comes in handy when for example the same app code runs on a development environment with a certain set of configuration values and in production with a different one.

Configuration on .NET Framework

If you have a working .NET Framework desktop application, chances are you have an app.config file accessed through the AppSettings class from System.Configuration namespace.

Within the .NET Framework infrastructure there is a hierarchy of configuration files that inherit properties from its parents. You can find a machine.config that defines a bunch of properties and configuration sections that can be used or overridden in any descendant configuration file.

Configuration on .NET Core

In the .NET Core world, there is no machine.config. Although, you can continue to use the old fashioned System.Configuration namespace, you may consider a switch to the modern Microsoft.Extensions.Configuration namespace which offers a good amount of enhancements.

The configuration API supports the concept of a configuration provider that defines the source of data to be used to load the configuration. There are a variety of built-in providers like in-memory .NET objects, INI files, JSON files, XML files, command-line arguments, environment variables and encrypted user store, or you can build your own.

The new configuration allows a list of name-value pairs, which can be grouped into a multi-level hierarchy. Any stored value maps to a string, and there is built-in binding support that allows you to deserialize settings into a custom POCO object.

The ConfigurationBuilder object lets you add as many configuration providers you may need for your application, using a precedence rule to resolve preference. So, the last provider you add in your code will override the others. This is a great feature for managing different environments for execution since you can define different configurations for development, testing and production environments, and managed them on a single function inside your code.

Migrating configuration files

You can continue using the old app.config XML file for configuration, or update to the new .NET Core configuration model.

To perform a migration from an old-style app.config to a new configuration file, you should choose between XML or JSON. In the case of XML, the conversion is straight forward since the content is the same, just rename "app.config" to "app.xml". Then you need to update your file to <appSettings> as your root xml node instead of <configuration>.

If you want to use a JSON format, and you don't want to migrate by hand, there is a tool called **dotnet-config2json** available for .NET Core that will output a JSON configuration file from an old app.config XML file.

You may also encounter some issues when using configuration sections that were defined in the machine.config. For example, consider this configuration:

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="General" value="4" />
    </switches>
    <trace autoflush="true" indentsize="2">
      <listeners>
        <add name="myListener" type="System.Diagnostics.TextWriterTraceListener, System,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
initializeData="MyListener.log" traceOutputOptions="ProcessId, LogicalOperationStack, Timestamp,
ThreadId, Callstack, DateTime" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

If you use this configuration with .NET Core, you will get an exception:

```
Unrecognized configuration section system.diagnostics
```

This is because the system.diagnostics section, and the assembly responsible for handling it was defined in the machine.config file no longer exists.

To fix the issue, copy the section definition from your old machine.config to your new configuration file:

```
<configSections>
  <section name="system.diagnostics" type="System.Diagnostics.SystemDiagnosticsSection,
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
</configSections>
```

Accessing databases

Almost every desktop application needs a database. For desktop, client-server architectures with a direct connection between the desktop app and the database engine are common. These databases can be local or remote depending on the need to share information between different users.

There are a lot of technologies and frameworks available to the developer to connect, query, and update, a database.

The most common examples of databases for Windows Desktop applications are Microsoft Access and Microsoft SQL Server. If you have more than 20 years of experience programming for the desktop, ODBC, OLEDB, RDO, ADO and ADO.NET, LINQ and Entity Framework will sound familiar.

ODBC

You can continue to use ODBC on .NET Core since the System.Data.Odbc library is compatible with .NET Standard 2.0.

OLE DB

OLE DB has been a great way to access various data sources in a uniform manner, but it was based on COM, which is a Windows-only technology. It is also unsupported in SQL Server versions 2014 and later. For those reasons, OLE DB will not be supported by .NET Core.

ADO.NET

You can still use ADO.NET from your existing desktop code on .NET Core. You will just need to update some NuGet packages.

EF Core vs EF 6

There are two versions of Entity Framework (EF): Entity Framework 6 and Entity Framework Core.

The latest version of Entity Framework for .NET Framework is 6.3. With the launch of .NET Core, Microsoft released a new data access stack based on Entity Framework called Entity Framework Core.

You can use EF 6.3 and EF Core from both .NET Framework and .NET Core, so which should you choose?

EF 6.3 will be the first version of EF 6 that can run on .NET Core and work cross-platform. In fact, the main goal of this release is to facilitate migrating existing applications that use EF 6 to .NET Core 3.0.

EF Core was designed to provide a developer experience similar to EF6. Most of the top-level APIs remain the same, so EF Core will feel familiar to developers who have used EF6.

Although compatible, there are differences on the implementation you should check before making a decision. (<https://docs.microsoft.com/ef/efcore-and-ef6/>)

The recommendation is to use EF Core if:

- The app needs the capabilities of .NET Core.
- EF Core supports all the features that the app requires

Consider using EF6 if both of the following conditions are true:

- The app will run on Windows and the .NET Framework 4.0 or later.
- EF6 supports all the features that the app requires.

Relational databases

SQL Server

SQL Server has been one of the databases of choice if you were developing for the desktop some years ago. With the use of `System.Data.SqlClient` in .NET Framework, you could access versions of SQL Server, which encapsulates database-specific protocols.

In .NET Core, you can find a new `SqlClient` fully compatible with the one existing in the .NET Framework but located in the `Microsoft.Data.SqlClient` library. You just have to add a dependency for this package and do some renaming for the namespaces and everything should work as expected.

Microsoft Access

Microsoft Access and its Jet Database Engine has been used for years when the sophisticated and more scalable SQL Server was not needed. You can still connect to Microsoft Access using the `System.Data.Odbc` library.

Consuming services

With the rise of service-oriented architectures, desktop applications begin to evolve from a client-server model to the three-tier approach. In the client-server approach a direct database connection is established from the client holding the business logic usually inside a single EXE file. On the other hand, the three-tier approach establishes an intermediate service layer implementing business logic and database access allowing for better security, scalability and reusability. Instead of working directly with datasets of data, the layer approach relies in a set of services implementing contracts and types objects to implement data transfer.

If you have a desktop application using a WCF service and you want to migrate it to .NET Core, there are some things to consider.

The first thing is how to resolve the configuration to access the service. Because the configuration is different on .NET Core, you will need to make some updates in your configuration file. Second, you will need to regenerate the service client with the new tools present on Visual Studio 2019.

If you find that after migration there are libraries you need that aren't present on .NET Core, you can add a reference to `Microsoft.Windows.Compatibility` and see if the missing functions are there.

If you are using the `WebRequest` class to perform Web Service calls you may encounter some differences on .NET Core. The recommendation is to use the `System.Net.Http.HttpClient` instead.

Consuming a COM Object

Currently there is no way to add a reference to a COM object from Visual Studio 2019 in .NET Core, so you have to manually modify the .csproj file for the project.

You need to insert a COMReference structure inside the Project file like:

```
<ItemGroup>

  <COMReference Include="MSHTML">

    <Guid>{3050F1C5-98B5-11CF-BB82-00AA00BDCE0B}</Guid>

    <VersionMajor>4</VersionMajor>

    <VersionMinor>0</VersionMinor>

    <Lcid>0</Lcid>

    <WrapperTool>primary</WrapperTool>

    <Isolated>>false</Isolated>

  </COMReference>

</ItemGroup>
```

More things to consider

Several technologies available to .NET Framework libraries aren't available for use with .NET Core. If your code relies on some of these technologies, consider the alternative approaches outlined below.

The Windows Compatibility Pack provides access to APIs that were previously available only for .NET Framework. It can be used from both .NET Core as well as .NET Standard.

For more information on API compatibility, the CoreFX team maintains a list of behavioral changes/compatibility breaks and deprecated/legacy APIs at GitHub.

(<https://github.com/dotnet/corefx/wiki/ApiCompat>)

AppDomains

Application domains (AppDomains) isolate apps from one another. AppDomains require runtime support and are generally quite expensive. Creating additional app domains is not supported on .NET Core. For code isolation, we recommend separate processes or using containers as an alternative. For the dynamic loading of assemblies, we recommend the new `AssemblyLoadContext` class.

To make code migration from .NET Framework easier, .NET Core exposes some of the AppDomain API surface. Some of the APIs function normally (for example, `AppDomain.UnhandledException`), some members do nothing (for example, `SetCachePath`), and some of them throw `PlatformNotSupportedException` (for example, `CreateDomain`).

Remoting

.NET Remoting was used for cross-AppDomain communication, which is no longer supported. Also, Remoting requires runtime support, which is expensive to maintain. For these reasons, .NET Remoting is not supported on .NET Core.

For communication across processes you should consider inter-process communication (IPC) mechanisms as an alternative to Remoting, such as the `System.IO.Pipes` or the `MemoryMappedFile` class.

Across machines, use a network-based solution as an alternative. Preferably, use a low-overhead plain text protocol, such as HTTP. The Kestrel web server, the web server used by ASP.NET Core, is an option here.

Code Access Security

Sandboxing, which relies on the runtime or the framework to constrain which resources a managed application or library uses is not supported on .NET Core.

Use security boundaries provided by the operating system, such as virtualization, containers, or user accounts, for running processes with the minimum set of privileges.

Security Transparency

Similar to CAS (Code Access Security), Security Transparency separates sandboxed code from security critical code in a declarative fashion but is no longer supported as a security boundary.

Use security boundaries provided by the operating system, such as virtualization, containers, or user accounts for running processes with the least set of privileges.

Windows 10 Migration

Introduction

Consider the following situation: You have a working desktop application that was developed for Windows 7. It is using WPF technology, available at that time, and it's working fine. However, when you run it on Windows 10, it looks dated. It is like when you look at a futuristic movie like Matrix and you see Neo using the Nokia 8110 device. The film feels great after 20 years, but it would benefit from a device modernization.

With the release of Windows 10, Microsoft introduced many innovations to support scenarios like tablets and touch devices to provide the best experience for users. You can log in with your face using Windows Hello. You can use a pen to draw or handwritten text that is automatically recognized and digitalized. You can even run locally customized AI models built on the cloud using WinML.

All these features are enabled for Windows developers through Windows Runtime libraries and you can take advantage of all of these features in your existing desktop apps because all these libraries are exposed to both the .NET Framework and .NET Core. You can even modernize your UI with the use of XAML Islands and improve the visuals and enhance the behavior of your apps.

One important thing to note here is that you don't need to abandon .NET Framework technology to follow this modernization path. You can safely stay on .NET Framework and have all the benefits of Windows 10 without the pressure to migrate to .NET Core. So, you get both the power and the flexibility to choose your modernization path.

WinRT APIs

Windows Runtime (WinRT) APIs are object-oriented, well-structured application programming interfaces (APIs) that give Windows 10 developers access to everything the operating system has to offer. Through WinRT APIs you can integrate functionalities like Push Notifications, Device APIs, Microsoft Ink, and WinML, and others, on your desktop apps.

In general, WinRT APIs can be called from a classic desktop app. However, two main areas present an exception to this rule:

- The APIs that require a package identity
- APIs that requires visualization like XAML or Composition.

UWP Packages

Application Package Identity

UWP apps have a deployment system where the OS manages the installation and uninstallation of application. That requires the installation to be declarative, meaning that no user code is executed during install, instead everything the app wants to integrate with the system (protocols, file types, extensions, etc.) is declared in the application manifest. At deployment time, the deployment pipeline configures those integration points. The only way for the OS to manage all this functionality and keep track of it, is for each 'package' to have an identity, a unique identifier for the application.

Some WinRT APIs require this package identity to work. However, classic desktop apps like native C++ or .NET apps, use different deployment systems that do not require a package identity. If you want to use these WinRT APIs in your desktop application, you need to provide a package identity.

One way to provide an identity is to build an additional packaging project. Inside the packaging project, you point to the original source code project and specify the Identity information. If you install the package and run the installed app, it will automatically get an identity, enabling your code to call all WinRT APIs requiring Identity.

```
<?xml version="1.0" encoding="utf-8"?>

<Package xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"

    xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"

    <Identity Name="YOUR-APP-GUID "

        Publisher="CN=YOUR COMPANY"

        Version="1.x.x.x" />

</Package>
```

You can check which APIs need a package identity by inspecting if the type that contains the API is marked with the *DualApiPartition* attribute. If it is present, you can call it from a traditional desktop app without an identity. If not present, you must provide an identity using the process described above.

<https://docs.microsoft.com/windows/desktop/apiindex/uwp-apis-callable-from-a-classic-desktop-app>

Benefits of packaging

Besides giving you access to these APIs, you get some additional benefits by creating a Windows App package for your desktop application including:

- **Streamlined deployment.** Apps have a great deployment experience ensuring that users can confidently install an application and update it. If a user chooses to uninstall the app, it is removed completely with no trace left.
- **Automatic updates and licensing.** Your application can participate in the Microsoft Store's built-in licensing and automatic update facilities. Automatic update is a highly reliable and efficient mechanism; the changed parts of files are downloaded during an update.
- **Increased reach and simplified monetization.** If you choose to distribute your application through the Microsoft Store you reach millions of Windows 10 users.
- **Add UWP features.** You can add UWP features to your app's package at your own pace.

Prepare for packaging

Before proceeding to package your desktop application, there are some points you must address before starting the process. Your application must respect any of the Microsoft Store rules and policies and run in the UWP application model. For example, it must run on the .NET Framework 4.6.2 or later. Any writes to the HKEY_CURRENT_USER registry hive and the AppData folder will be virtualized to a user-specific app-specific location.

The design goal for packaging is to separate the application state from system state while maintaining compatibility with other apps. Windows 10 accomplishes this by placing the application inside a Universal Windows Platform (UWP) package. It detects and redirects some changes to the file system and registry at runtime to fulfill the promise of a trusted and clean install and uninstall behavior of an application provided by packaging.

Packages that you create for your desktop application are desktop-only, full-trust applications, and aren't sandboxed, although there is lightweight virtualization applied to the app for writes to HKCU & AppData. This allows them to interact with other apps the same way classic desktop applications do.

- Installation

App packages are installed under `C:\Program Files\WindowsApps\package_name`, with the executable titled `app_name.exe`. Each package folder contains a manifest (named `AppxManifest.xml`) that contains a special XML namespace for packaged apps. Inside that manifest file is an `<EntryPoint>` element, which references the full-trust app. When that application is launched, it does not run inside an app container, but instead it runs as the user as it normally would.

After deployment, package files are marked read-only and heavily locked down by the operating system. Windows prevents apps from launching if these files are tampered with.

- File system

The OS supports different levels of file system operations for packaged desktop applications, depending on the folder location.

When trying to access the user's AppData folder the system creates a private per-user, per-app location behind the scenes. This creates the illusion that the packaged application is editing the real AppData when it is actually modifying a private copy. By redirecting writes this way, the system can track all file modifications made by the app. It can then clean all those files when uninstalling reducing system "rot" and providing a better application removal experience for the user.

- Registry

App packages contain a `registry.dat` file, which serves as the logical equivalent of `HKLM\Software` in the real registry. At runtime, this virtual registry merges the contents of this hive into the native system hive to provide a singular view of both.

All writes are kept during package upgrade and only deleted when the application is uninstalled.

- Uninstallation

When the user uninstalls a package, all files and folders located under `C:\Program Files\WindowsApps\package_name` are removed, as well as any redirected writes to AppData or the registry that were captured during the process.

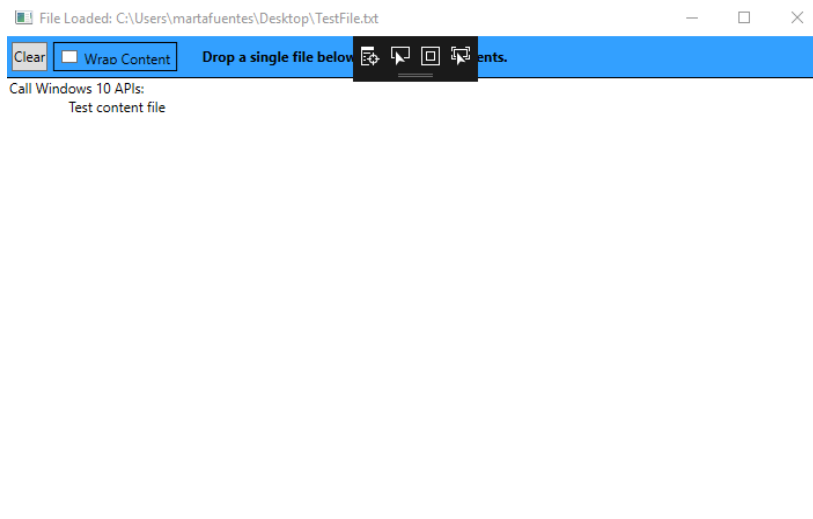
For details about how a packaged application handles installation, file access, registry and uninstallation please read <https://docs.microsoft.com/windows/msix/desktop/desktop-to-uwp-behind-the-scenes>

You can get a complete list of things to check on <https://docs.microsoft.com/windows/msix/desktop/desktop-to-uwp-prepare>

How to add WinRT APIs to your desktop project

In this section, you can find a walkthrough about how to integrate Toast Notifications on an existing WPF application. Although it is simple from the code perspective, it helps illustrate the whole process. Notifications are one of the many available WinRT APIs available that you can use in .NET app. In this case, the API requires a Package Identity. This process is more straightforward if the APIs doesn't require Package Identity.

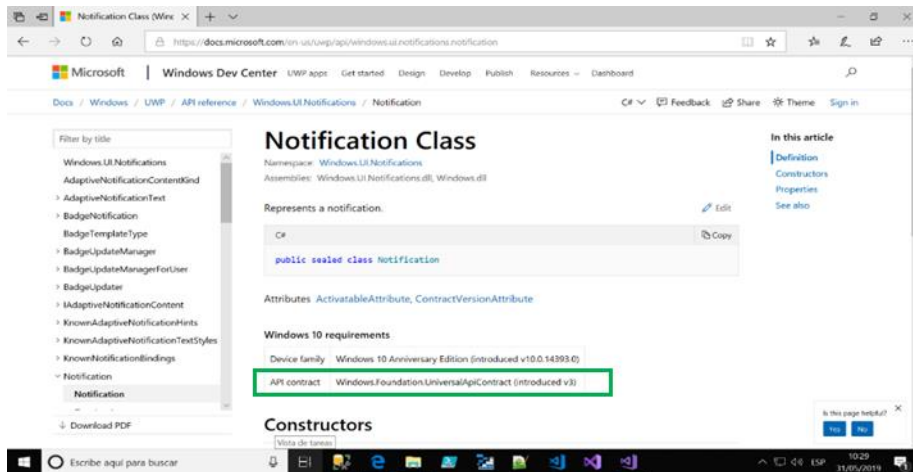
Let's take an existing WPF sample app that reads files and shows its contents on the screen. The goal will be to display a Toast Notification when the application starts.



First, you should check in the below link whether the Windows 10 API that you will use requires a Package Identity:

<https://docs.microsoft.com/windows/apps/desktop/modernize/desktop-to-uwp-supported-api>

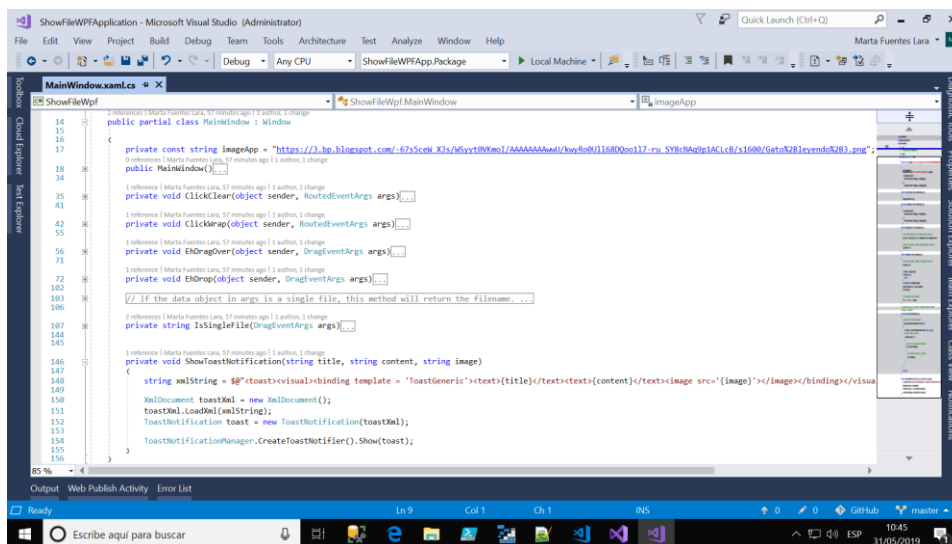
Our sample will use the Windows.UI.Notifications API that requires a packaged identity:



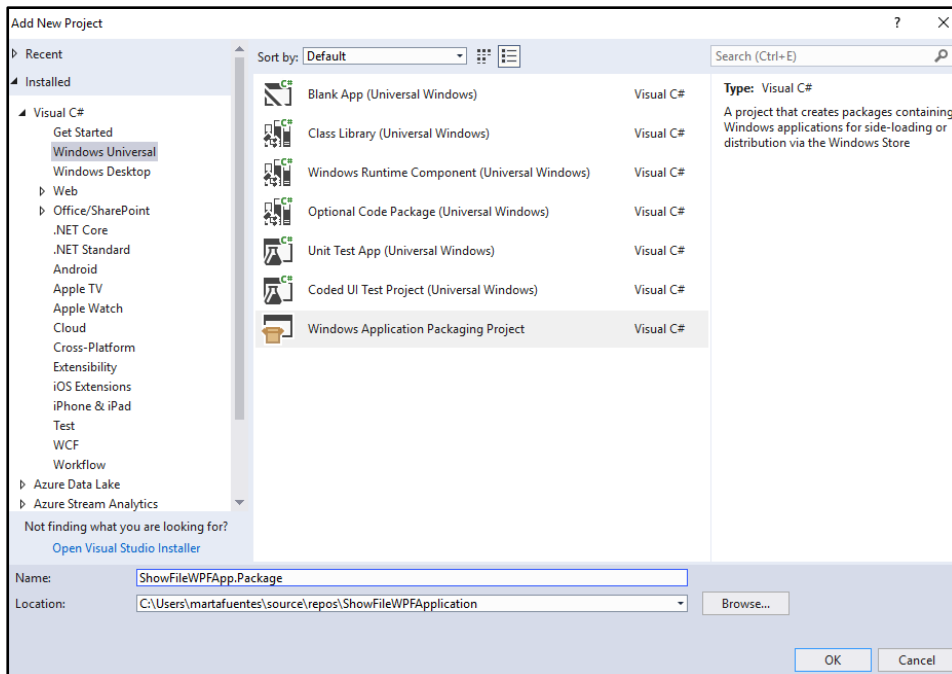
To enable access to the Windows Runtime API you just need to add a reference to the Microsoft.Windows.SDK.Contracts NuGet package and this will do the magic behind the scenes (see details in this link: <https://blogs.windows.com/windowsdeveloper/2019/04/30/calling-windows-10-apis-from-a-desktop-application-just-got-easier/#OQKKZIOZu4p7IC0q.97>)

You are now prepared to start adding some code.

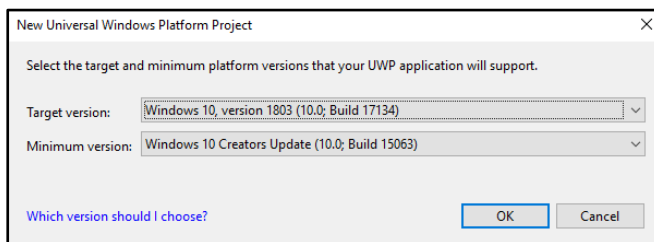
You need to create a ShowToastNotification method that will be called on application startup. It just builds a toast notification from an XML pattern:



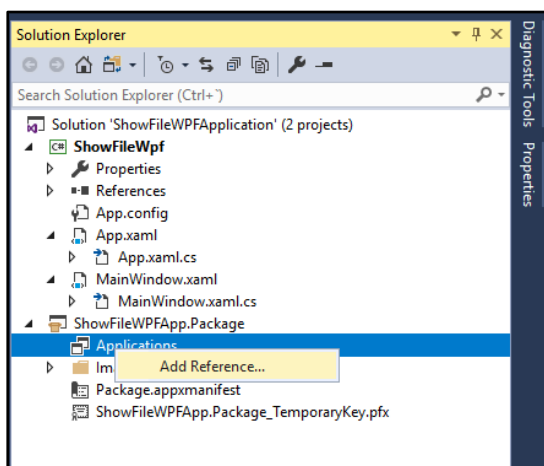
Although the project builds, there are errors because the Notifications API requires a Package Identity, and we didn't provide it. Adding a Windows Packaging Project in the Solution will fix it:

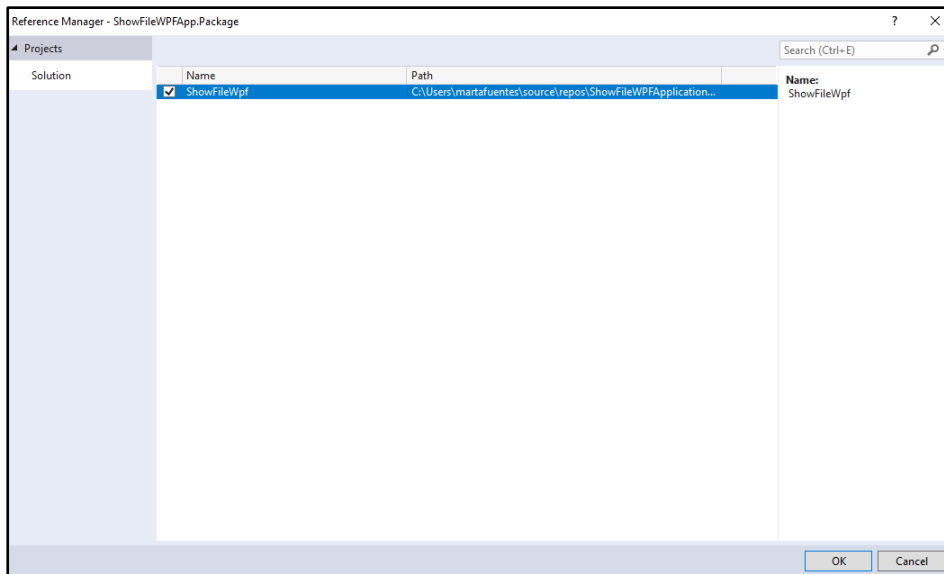


You should select the minimum Windows version you want to support and the version you are targeting. Not all the WinRT APIs are supported in all Windows 10 versions. Each Windows 10 update adds new APIs that are only available from this version; down-level support is not available.

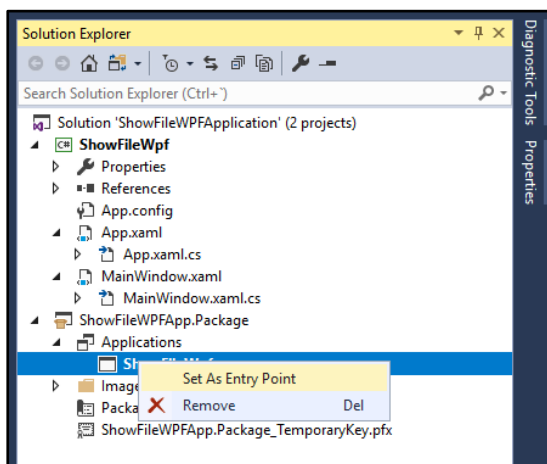


Next step is to add the WPF application to the Windows Packaging Project by adding a project reference:

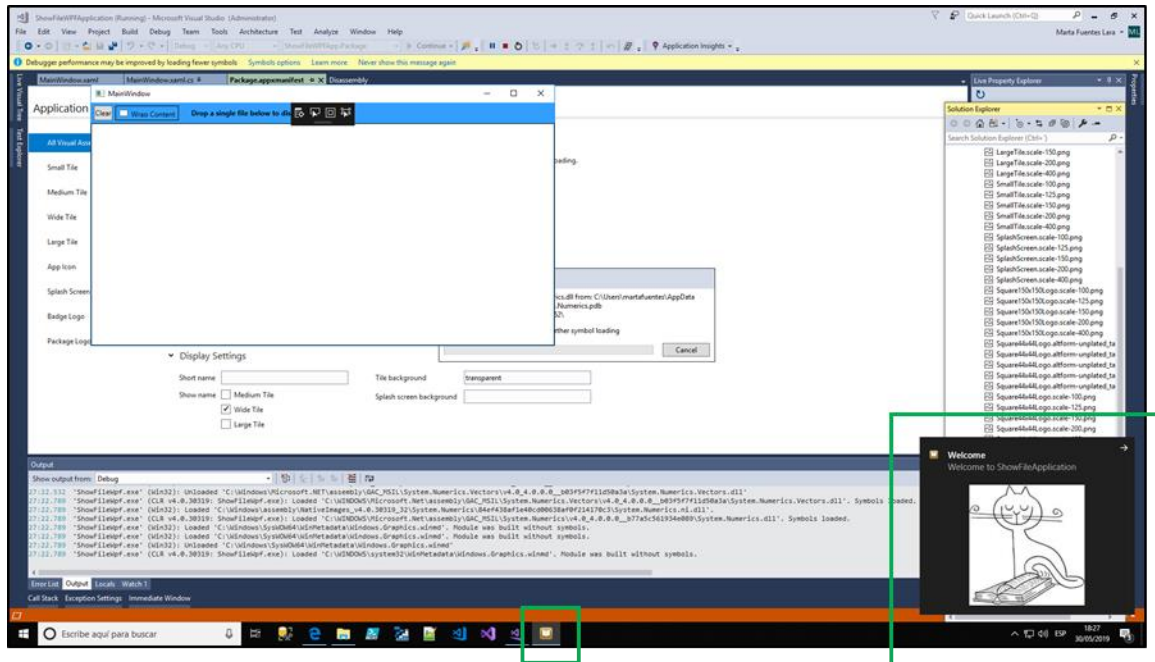




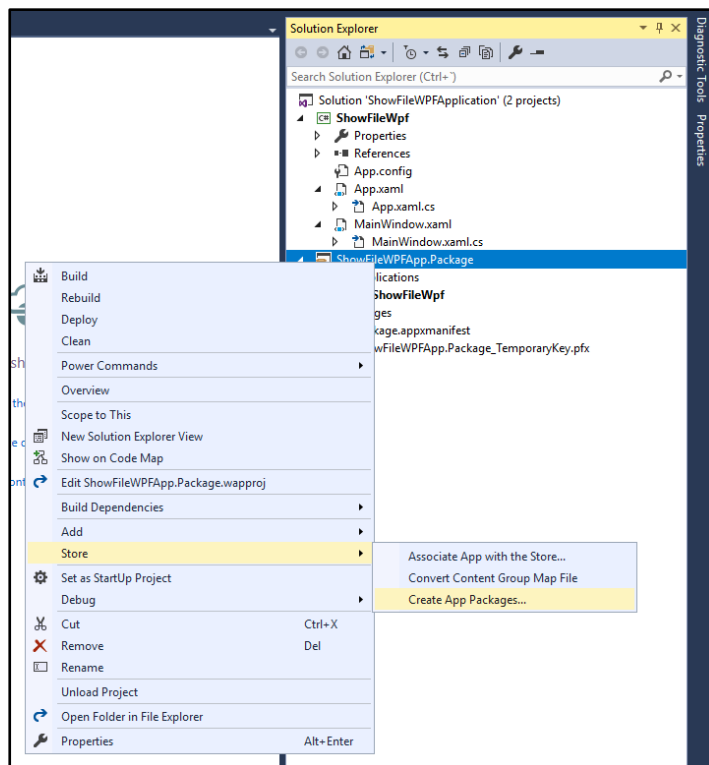
A Windows Packaging Project can package several apps so you should set which one is the Entry Point:



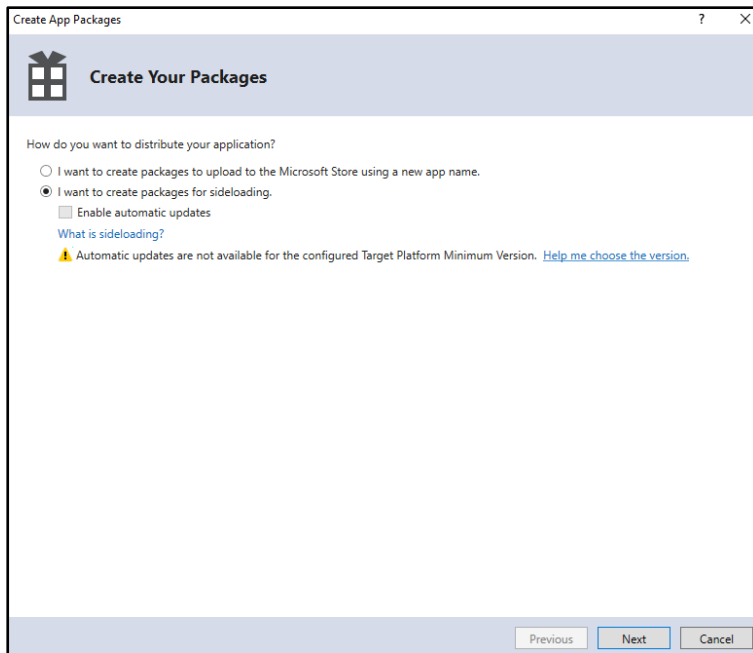
Next step is to set the WPF Project as the startup Project in the solution configuration. You can press F5 to compile and build and see the results.



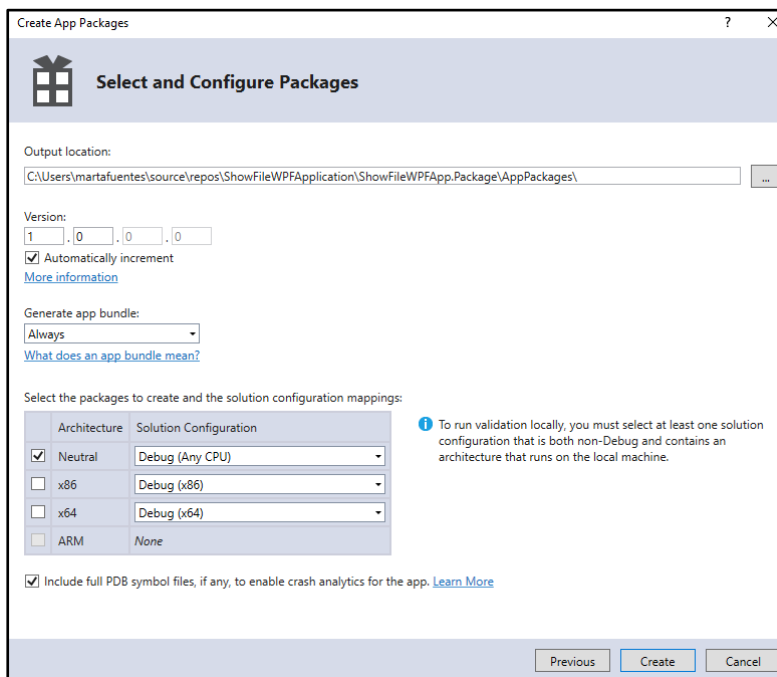
Let's generate the package so you can install your app. Right click on **Store > Create App Packages**



Select the sideloading option to deploy the app from your machine:



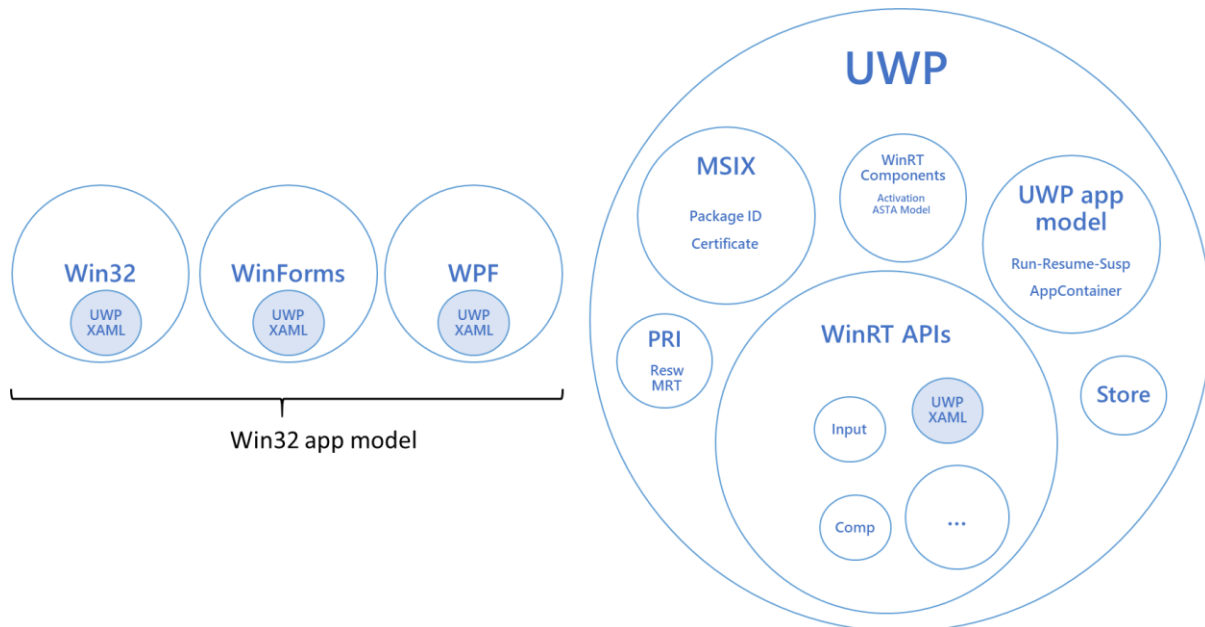
Select the application architecture of your app:



Finally, create the package clicking on **Create**.

XAML Islands

XAML Islands are a set of components that enable Windows desktop developers to use UWP XAML controls on their existing Win32 applications, including Windows Forms and WPF.



You can image your Win32 app with your standard controls and among them an “island” of UWP UI containing controls from the modern world. The concept is similar as having an iFrame inside a web page that shows content from a different page.

Besides adding functionality from the Windows 10 APIs, you can add pieces of UWP XAML inside of your app using XAML Islands.

Windows 10 1903 update introduces a set of APIs that allows hosting UWP XAML content in Win32 windows. Only apps running on Windows 10 1903 and above can use XAML Islands.

The road to XAML Islands

The road to XAML Islands started in 2012 when Microsoft introduced the WinRT APIs as a framework to modernize the Win32 apps (Windows Forms, WPF, and native Win32 apps). However, the new UI controls inside WinRT were available for new applications but not for existing ones.

In 2015, along with Windows 10, UWP was born. The Universal Windows Platform (UWP) allows you to create apps that work across Windows devices like Xbox, Mobile, and Desktop. One year later, Microsoft announced Desktop Bridge (formerly known as Project Centennial). Desktop Bridge is a set of tools that allowed developers to bring their existing Win32 apps to the Microsoft Store. More capabilities were added in 2017, allowing developers to [enhance](#) their Win32 apps leveraging some of the new Windows 10 APIs, like live tiles and notifications on the action center. But still, no new UI controls.

At Build 2018, Microsoft announced a way for developers to use the new Windows 10 XAML controls into their current win32 apps, without fully migrating their apps to UWP. It was branded as UWP XAML Islands.

How it works

The Windows 10 1903 update introduces several XAML hosting APIs. Two of them are [WindowsXamlManager](#) and [DesktopWindowXamlSource](#).

The WindowsXamlManager class handles the UWP XAML Framework. Its method InitializeForCurrentThread loads the UWP XAML Framework inside the current thread of the Win32 app.

The DesktopWindowXamlSource is the instance of your XAML Island content. It has the Content property which you are responsible for instantiating and setting. The DesktopWindowXamlSource renders and gets its input from an HWND. It needs to know to which other HWND it will attach the XAML Island's one, and you are responsible for sizing and positioning the parent's HWND.

WPF or Windows Forms developers don't usually deal with HWND inside their code, so it may be hard to understand and handle HWND pointers and the underlying wiring stuff to communicate Win32 and UWP worlds.

The XAML Islands .NET Wrappers

The Windows Community Toolkit has a set the XAML Islands .NET wrappers for WPF or Windows Forms that make easier to use XAML Islands. These wrappers manage the HWNDs, the focus management, among other things. Windows Forms and WPF developers should use these wrappers.

The Windows Community Toolkit offers two types of controls: Wrapped Controls and Hosting Controls

Wrapped Controls

These wrapped controls are UWP controls wrapped into Windows Forms or WPF controls, hiding UWP concepts from the developer. These controls are:

- WebView and WebViewCompatible
- InkCanvas and InkToolbar
- MediaPlayerElement
- MapControl

Add the [Microsoft.Toolkit.Wpf.UI.Controls](#) package to your project, include the reference to the namespace and start using them.

```
<Window
...
xmlns:uwpControls="clr-
namespace:Microsoft.Toolkit.Wpf.UI.Controls;assembly=Microsoft.Toolkit.Wpf.UI.Controls">

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*/>
    </Grid.RowDefinitions>
```

```
<uwpControls:InkToolbar TargetInkCanvas="{x:Reference Name=inkCanvas}"/>
<uwpControls:InkCanvas Grid.Row="1" x:Name="inkCanvas" />
</Grid>
```

Hosting Controls

Most of the built-in controls, as well as 3rd party controls, can be integrated into Windows Forms and WPF as a “XAML Island” of fully functional UI. The `WindowsXamlHost` control for WPF and Windows Forms allows doing this.

For example, to use the `WindowsXamlHost` control in WPF, you just need to add the [Microsoft.Toolkit.Wpf.UI.XamlHost](#) package provided by Windows Community Toolkit. Once you have placed the `WindowsXamlHost` into your UI code, you just need to specify which UWP type you want to load and you can choose to use a wrapped control like a `Button` or a more complex one composed by several different controls.

Here is a sample code snippet that shows how to add a UWP `Button`:

```
<Window
...
xmlns:xamlhost="clr-
namespace:Microsoft.Toolkit.Wpf.UI.XamlHost;assembly=Microsoft.Toolkit.Wpf.UI.XamlHost">

<xamlhost:WindowsXamlHost x:Name="myUwpButton"
InitialTypeName="Windows.UI.Xaml.Controls.Button" />
```

It is better to have a single, bigger, XAML Island containing a custom composite control than to have several islands each with one control.

One of the core features of XAML is the binding markup extension and it works between your Win32 code and the island. For instance, you can bind a Win32 `Textbox` with a UWP `Textbox`. One important thing to consider is that these bindings are one-way bindings, from UWP to Win32, so if you update the `Textbox` inside the XAML Island the Win32 `Textbox` will be updated, but not the other way around.

This link contains a walkthrough about how to use XAML Islands:

<https://docs.microsoft.com/windows/apps/desktop/modernize/host-standard-control-with-xaml-islands>

Adding UWP XAML custom controls

A XAML custom control is a control (or user control) created by you or by 3rd parties (including WinUI 2.x controls). To host a custom UWP control in a Windows Forms or WPF app, you'll need to use the [WindowsXamlHost](#) in your app. Additionally, you must create a UWP app project that defines a `XamlApplication` object.

Your WPF (or Windows Forms) project must have access to an instance of the `Microsoft.Toolkit.Win32.UI.XamlHost.XamlApplication` class provided by the Windows Community Toolkit. This object acts as a root metadata provider for loading metadata for custom UWP XAML types in assemblies in the current directory of your application. The recommended way to do this is to add a Blank App (Universal Windows) project to the same solution as your WPF (or Windows Forms) project and revise the default App class in this project.

The custom UWP XAML control can be included on this UWP app or in a independent UWP Class Library project that you reference in the same solution as your WPF (or Windows Forms) project.

You can check a details step by step process description here:

<https://docs.microsoft.com/windows/apps/desktop/modernize/host-custom-control-with-xaml-islands>

The Windows UI Library (WinUI 2)

Besides the built-in Windows 10 controls that come with Windows, additional controls are included in the Windows UI Library (**WinUI 2**). WinUI 2 provides official native Microsoft UI controls and features for Windows UWP apps and these controls can be used inside of XAML Islands.

WinUI 2 is open source and you can find information at: <https://github.com/microsoft/microsoft-ui-xaml>

The following article demonstrates how to host a UWP XAML control from the WinUI 2 library: <https://docs.microsoft.com/windows/apps/desktop/modernize/host-custom-control-with-xaml-islands>

Do you need XAML Islands

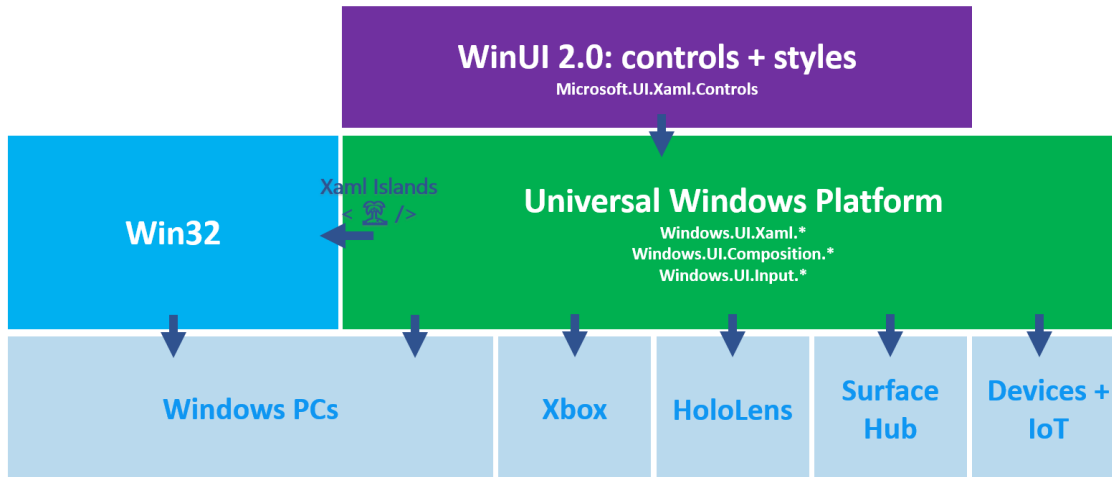
XAML Islands are intended for existing Win32 Apps that wants to improve their user experience by leveraging new UWP controls and behaviors without a full rewrite of the app. Until XAML Islands was introduced, you could only leverage Windows 10 APIs.

If you are developing a new Windows App, a UWP App is probably the right approach.

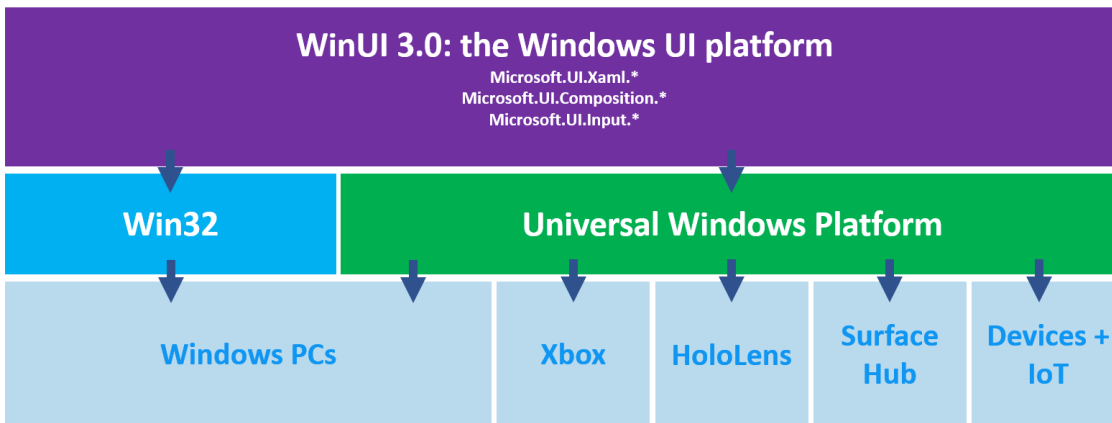
The road ahead XAML Islands: WinUI 3.0

Since Windows 8, the Windows UI platform, including the [XAML UI](#) framework, [visual composition](#) layer, and [input processing](#), have all been shipped as an integral part of Windows. This means that to benefit from the latest improvements on UI technologies. you must upgrade to the latest version of the UI, slowing down development of your apps. To decouple these two evolution cycles and foster innovation, Microsoft is actively working on the WinUI project.

Starting with WinUI 2 in 2018, Microsoft started shipping some new XAML UI controls and features as separate [NuGet](#) packages that build on top of the UWP SDK.



WinUI 3 is under active development and will greatly expand the scope of WinUI to include the full UI platform, which will now be fully decoupled from the UWP SDK:



This means that the XAML framework will now be developed on GitHub and ship out of band as NuGet packages. This includes the XAML Islands APIs that will support loading the WinUI 3 platform on Win32 apps.

The existing UWP XAML APIs that ship as part of the OS will no longer receive new feature updates. They will still receive security updates and critical fixes according to the Windows 10 support lifecycle.

The Universal Windows Platform contains more than just the XAML framework (e.g. application and security model, media pipeline, Xbox and Windows 10 shell integrations, broad device support) and will continue to evolve. All new XAML features will just be developed and shipped as part of WinUI instead of the OS.

WinUI 3 in desktop app and WinUI XAML Islands

As you can see, WinUI 3 is the evolution of UWP XAML and it works naturally within the UWP app model and all its requirements (MSIX packaged ID, sandbox, CoreWindow, etc.). To use just WinUI 3 in a Win32 app model, the WinUI content should be hosted by another UI Framework (Windows Forms, WPF, etc.) using **WinUI XAML Islands**. This is the right path if you want to evolve your app and mix

technologies. However, if you want to replace your entire old UI with WinUI, your app shouldn't load UI Frameworks for just hosting WinUI.

WinUI 3 will address this critical feedback adding **WinUI in desktop apps**. This will mean that Win32 apps can use WinUI 3 as standalone UI Framework, without the need to load Windows Forms or WPF.

WinUI 3 will let developers easily mix and match the right combination of:

- App model: UWP, Win32
- Platform: .NET Core 3 or Native
- Language: .NET (C#, Visual Basic), standard C++
- Packaging: [MSIX](#), AppX for the Microsoft Store, unpackaged
- Interop: use WinUI 3 to extend existing WPF, WinForms and MFC apps using WinUI XAML Islands.

For more information, see <https://github.com/microsoft/microsoft-ui-xaml>

Migrating an example desktop application to .NET Core 3.0

Introduction

In this chapter, we present practical guidelines to help you perform a migration of your existing application from .NET Framework to .NET Core.

We present a well-structured process you can follow and the most important things to consider on each step.

We then document a step-by-step migration process for a sample desktop application, both with Windows Forms and Windows Presentation Foundation versions.

Migration Process Overview

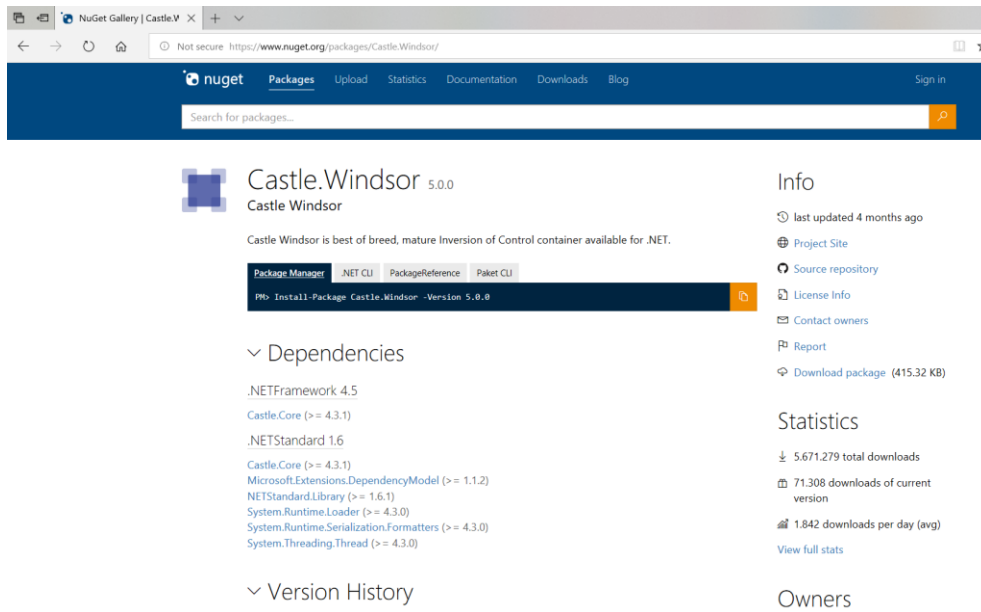
The migration process can be articulated in four sequential steps:

1. **Preparation:** You must understand the dependencies of the project to have an idea of what is ahead. In this step, you take the current project into a state that simplifies the startup point for the migration.
2. **Migrate Project File:** .NET Core projects use the new SDK style projects. You need to create a new project file with this format or update the one you must use the SDK style.
3. **Fix Code and Build:** Get the code compiling in .NET Core, addressing API differences between .NET Framework and .NET Core or in third party packages used by your app. Additionally, regenerate any generated code like WCF client-code.
4. **Run and Test:** There are differences that don't show up until run-time, so you need to test your app to be sure everything works as expected.

Preparation

Check out every dependency compatibility in .NET Core

Once you have migrated the package references you must check each reference for compatibility. If you go to nuget.org you can explore the dependencies of each NuGet package your application is using. If it has .NET Standard dependencies, then it's going to work on .NET Core because it depends on .NET Standard. Here you can see a screenshot showing the dependencies for the Castle.Windsor Package:



To check out package compatibility, you can use the tool <http://fuget.org> that offers a more detailed information about versions and dependencies.

Maybe the versions of the packages referenced by the project are older versions that don't support .NET Core but you can find newer versions that do support it. Therefore, updating packages to newer versions is in general a good recommendation although you must consider that this can introduce some breaking changes forcing you to update your code to keep it compiling well.

What happens if you don't find a compatible version? What if you just don't want to update the version of a package because of these breaking changes? Don't worry because it is possible to depend on .NET Framework packages from a .NET Core application. You must test this extensively because it can cause runtime errors if the external package calls an API that is not available on .NET Core. This is great news when you are using an old package that is not going to be updated and you just can retarget to work on the .NET Core.

Check for API compatibility

Since APIs surfaces in .NET Framework and .NET are similar but not identical, you must check which of those APIs are available on .NET Core and which are not. You can use the .NET Portability Analyzer tools to surface APIs used that aren't present on .NET Core. It looks at the binary level of your app and extracts all the APIs that are called showing the ones that aren't available on your target framework, .NET Core 3.0 in our case.

You can find this tool at:

<https://docs.microsoft.com/dotnet/standard/analyzers/portability-analyzer>

The most interesting result from this tool refers to differences in your own code and not in external packages that you cannot change. Remember you should have updated most of these packages to make them work with .NET Core.

Migrate Project File

Create the new .NET Core project

In most of the cases you will want to update your existing project to the new .NET Core format, but be aware that you can also create a new project while maintaining the old one. The main drawback from updating the old project is that you lose designer support, which may be important to you. If you want to keep using the designer, you must create a new .NET Core project in parallel with the old one and share assets. When it is time to modify UI elements in the designer you can switch to the old project to do that, and since assets are linked there will be updated in the .NET Core project.

.NET Core SDK style project are a lot simpler than .NET Framework and apart from the mentioned PackageReference stuff, you will not need to do much more. SDK style adds all necessary files underneath project file location like .cs and .xaml files without the need to explicitly include them in the .csproj.

Assembly.info considerations

For .NET Core projects AssemblyInfo.cs file gets automatically generated, so if you happen to have this file already in your project, you will get errors. To fix this you can either delete your existing AssemblyInfo.cs file or keep it and disable auto generation by adding this entry to the .NET Core project file:

```
<GenerateAssemblyInfo>false</GenerateAssemblyInfo>
```

Resources

Embedded resources are included automatically but resources are not, so you need to migrate those to the new .csproj file.

Packages References

In a .NET Framework application, all references to external packages are declared in the packages.config file. In .NET Core, there is no longer the need to use the packages.config file. Instead, the new PackageReference format inside the project file is used to pull in NuGet dependencies.

You must upgrade from one format to another. You can let Visual Studio do the work for you by right-clicking on the packages.config file and selecting the "Migrate packages.config to PackageReference" option. Or you can do it manually by taking the dependencies contained in the packages.config file and pasting them into the project file with the PackageReference attribute.

Fix Code and Build

Microsoft.Windows.Compatibility

In case your applications depend on APIs that aren't available on .NET Core like Registry, ACLs, WCF you have to include a reference to the Microsoft.Windows.Compatibility package to add these Windows-specific APIs. They work on .NET Core but aren't included as they aren't cross-platform.

There is a tool called API Analyzer (<https://docs.microsoft.com/dotnet/standard/analyzers/api-analyzer>) that help you identify not compatible APIs as long as you develop your code.

Use #if defines

If you need different execution paths when targeting .NET Framework and .NET Core, you should use #if directives to keep the same code base for both targets.

Technologies not available on .NET Core

Some technologies aren't available on .NET Core like:

- AppDomains
- Remoting
- Code Access Security
- WCF Server
- Windows Workflow

Therefore, it is time to look for a replacement for those if you use them.

Regenerate autogenerated clients

If your application uses some autogenerated code like, for example a WCF client, you may need to regenerate this code to target .NET Core. Sometimes you can find some missing references since it may not be included as part of the basic .NET Core assemblies. Using a tool like .NET API Catalog (<https://apisof.net/>) you can easily locate the assembly the missing reference lives in and add it from NuGet.

Rolling back package versions

As a rule, we have stated that you should update every single package to be compatible with .NET Core. However, you may find that targeting a compatible version of an assembly just does not pay off. If the cost of change is not acceptable, you can consider rolling back package versions, keeping the ones you used on .NET Framework. Although they may not target .NET Core, they should work unless they call an unsupported API.

Run and Test

Once you have your application building with no errors, you can start the last step of migration by testing all aspects of the app.

In this final step, you may find a variety of issues depending on the complexity of your application and the dependencies and APIs you are using.

For example, in case you use configuration files, such as `app.config`, you may find some errors at run-time like Configuration Sections not present. Using `Microsoft.Extensions.Configuration` should fix the problem.

Another reason for errors is the use of `BeginInvoke` and `EndInvoke`; these aren't supported on .NET Core. The reason they aren't supported on .NET Core is that they have a dependency on `Remoting`, which does not exist on .NET Core. To solve this issue, try to use `Async` when available or `Task.Run`.

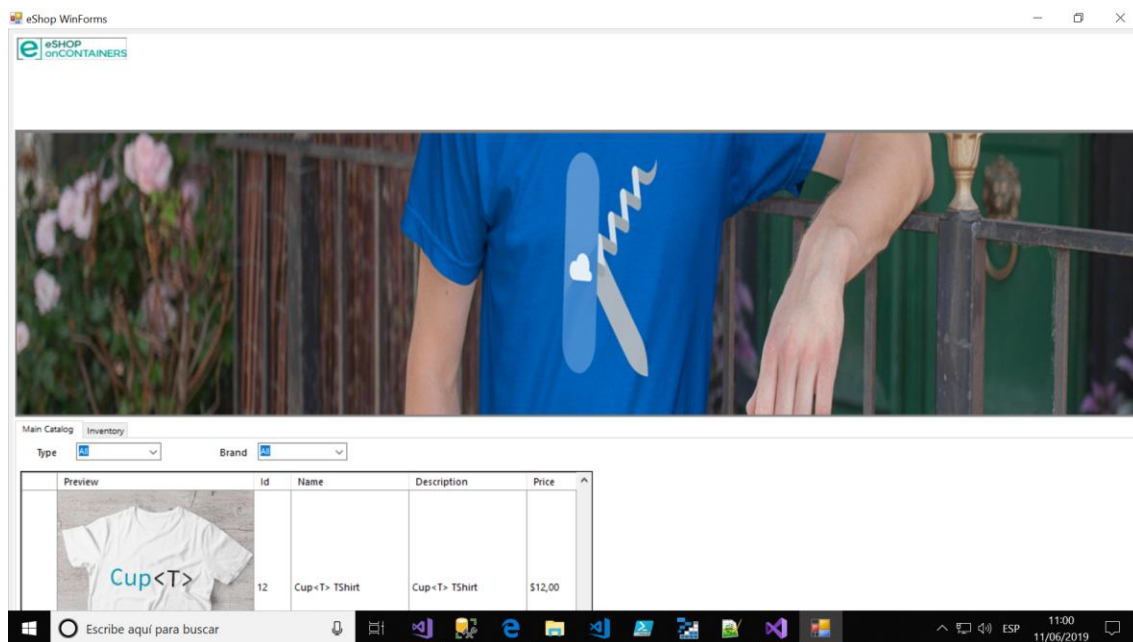
You can use compatibility analyzers to let you identify APIs and code patterns that can potentially cause problems at run-time with .NET Core. You can go to <http://github.com/dotnet/platform-compat> and use the Roslyn analyzers.

Migrating a Windows Forms application

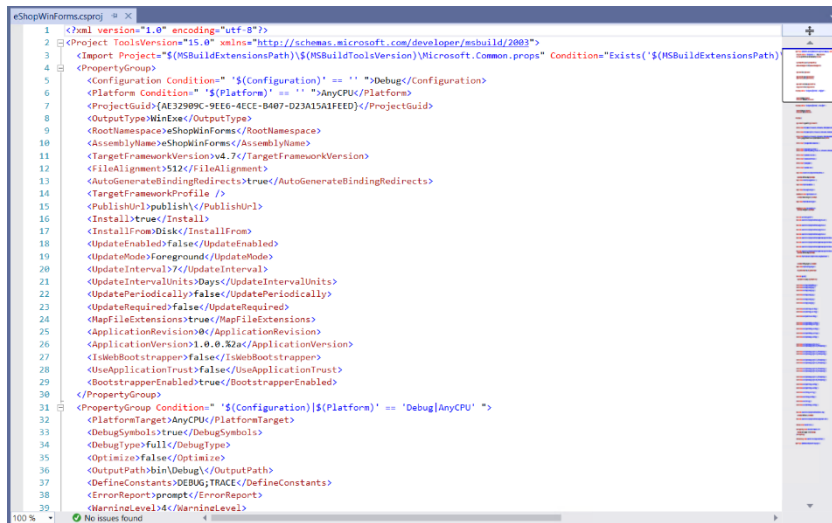
To showcase a complete migration process of a Windows Forms application, we've chosen to migrate the eShop sample application available at <https://github.com/dotnet-architecture/eShopModernizing>.

This application shows a product catalog and allows the user to navigate, filter, and search for products. From an architecture point of view the app relies on an external WCF service that serves as a façade to a back-end database.

You can see the main application window in the following picture:



If we open the `.csproj` file we can something like this, the .NET Framework project file:



As mentioned before, .NET Core project has a more compact style and we need to migrate the project structure to the new .NET Core SDK style.

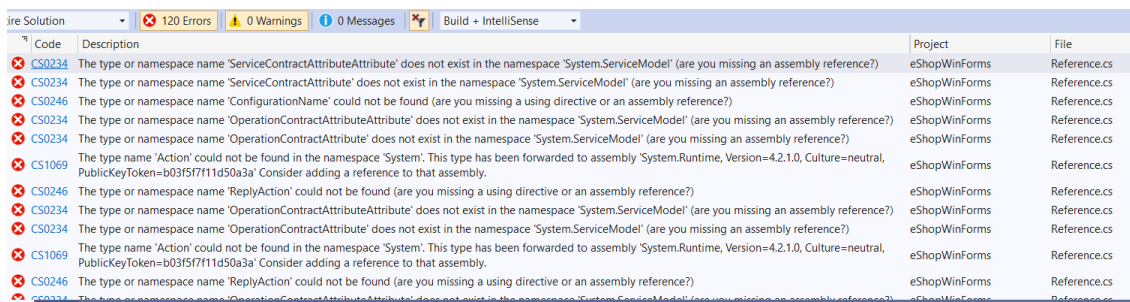


Select the Windows Forms project in the solution explorer and **right click -> Unload Project -> Edit**

Now we can update the .csproj file. We will delete all the content and replace it with:

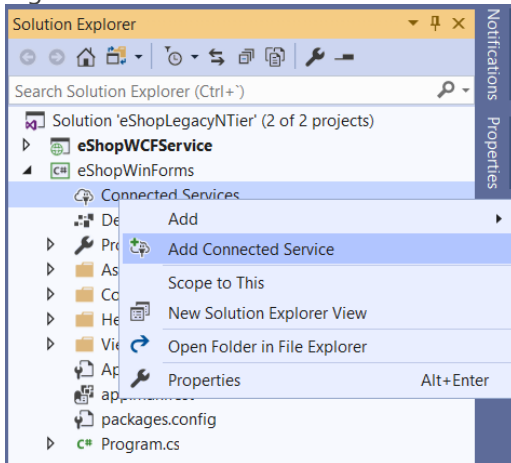
We are done updating the project file, save and reload it. Now the project is targeting the .NET Core.

If we compile the project at this point, we find some errors related to the WCF client reference. Since this is autogenerated code we must regenerate it to target .NET Core.

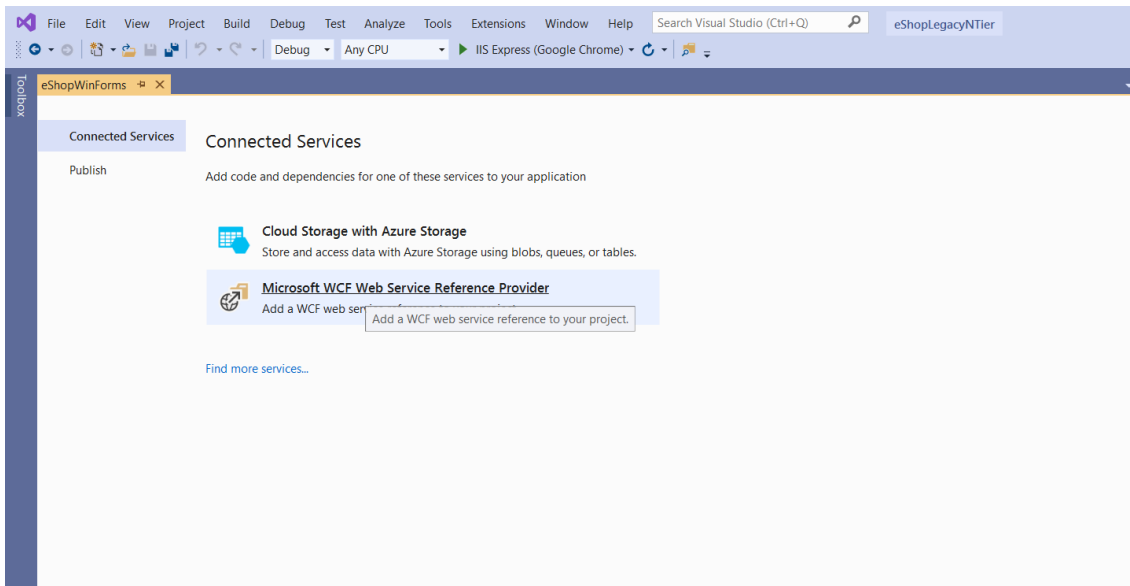


We can delete the Reference.cs file and add generate a new Service Client.

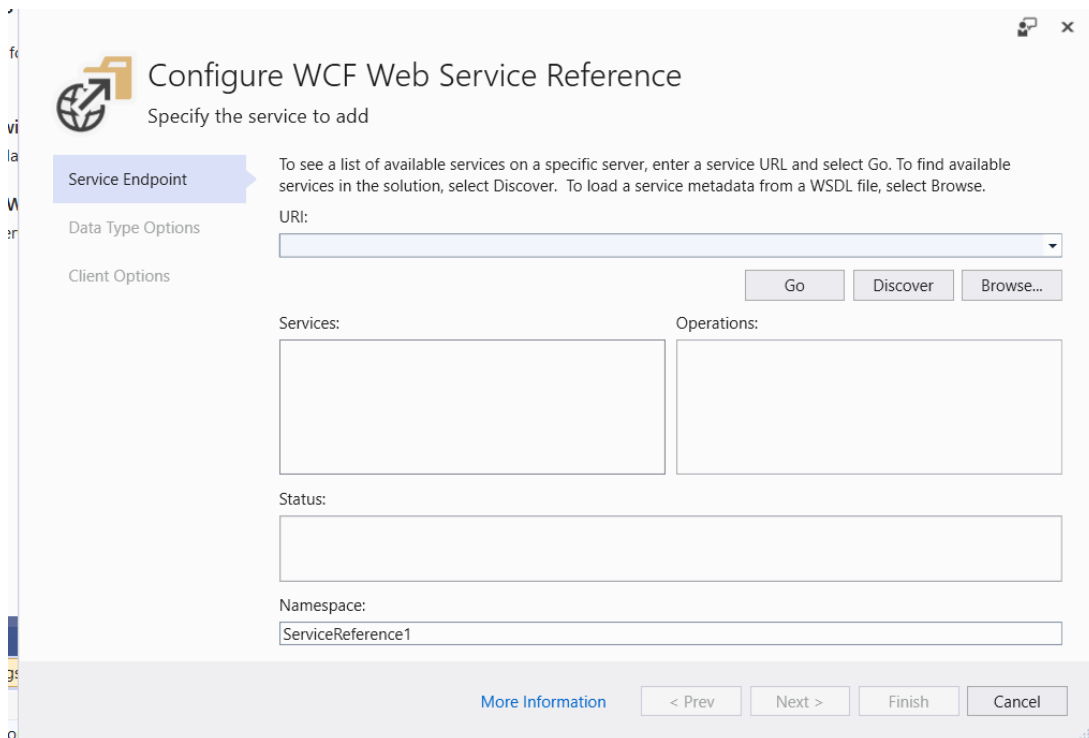
Right-click over **Connected Services** and select the **"Add Connected Service"** option.



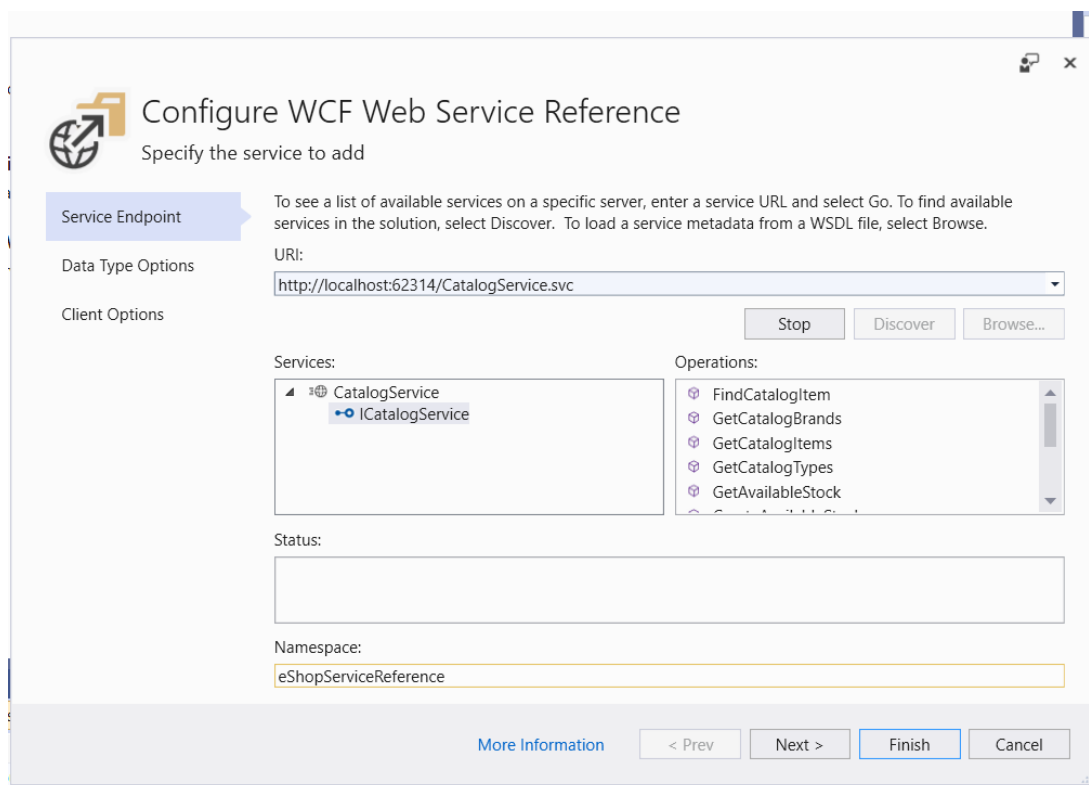
This will open the Connected Services window where we select the Microsoft WCF Web Service option.



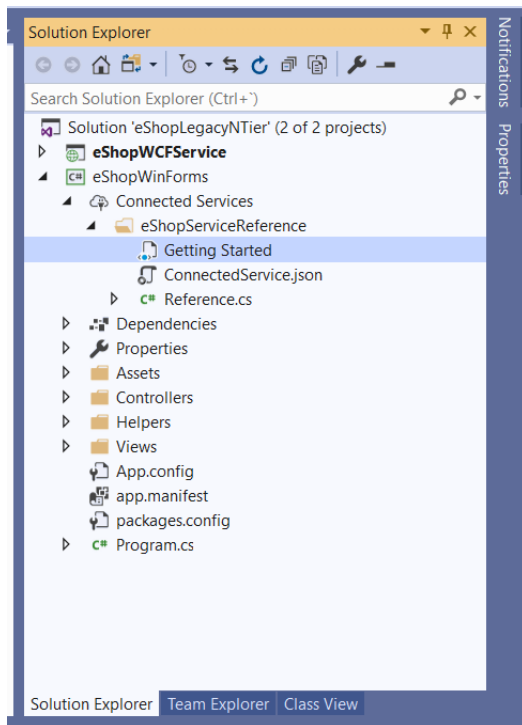
If we have the WCF Service in the same solution, we can choose to select the Discover option instead of specifying a service URL.



Once the service is located the tools reflects the API contract implemented by the service. We change the name of the Namespace to be eShopServiceReference:



We click on **Finish** and after a while, we will see the generated code.



We can see three generated documents:

1. Getting Started: just a link to GitHub to provide some info on WCF
2. ConnectedService.json: configuration parameters to connect to the service.
3. Reference.cs: this is the actual WCF client code.

If we compile again, we see many errors coming from .cs files inside the Helper folder. This folder was present in the .NET Framework version but not included in the old .csproj. But with the new SDK project style every code file present underneath the project file location is included by default, meaning that the new .NET Core project tries to compile the files inside the Helper folder. Since it is not needed, we can safely delete it.

We compile again but when we execute the application, we see no images of products. The problem is that now the path to the files has changed slightly and we need to add another level of depth in the path, updating from:

```
string image_name = Enviroment.CurrentDirectory + "../../../../Assets\\Images\\Catalog\\" +  
catalogItems.Picturefilename;
```

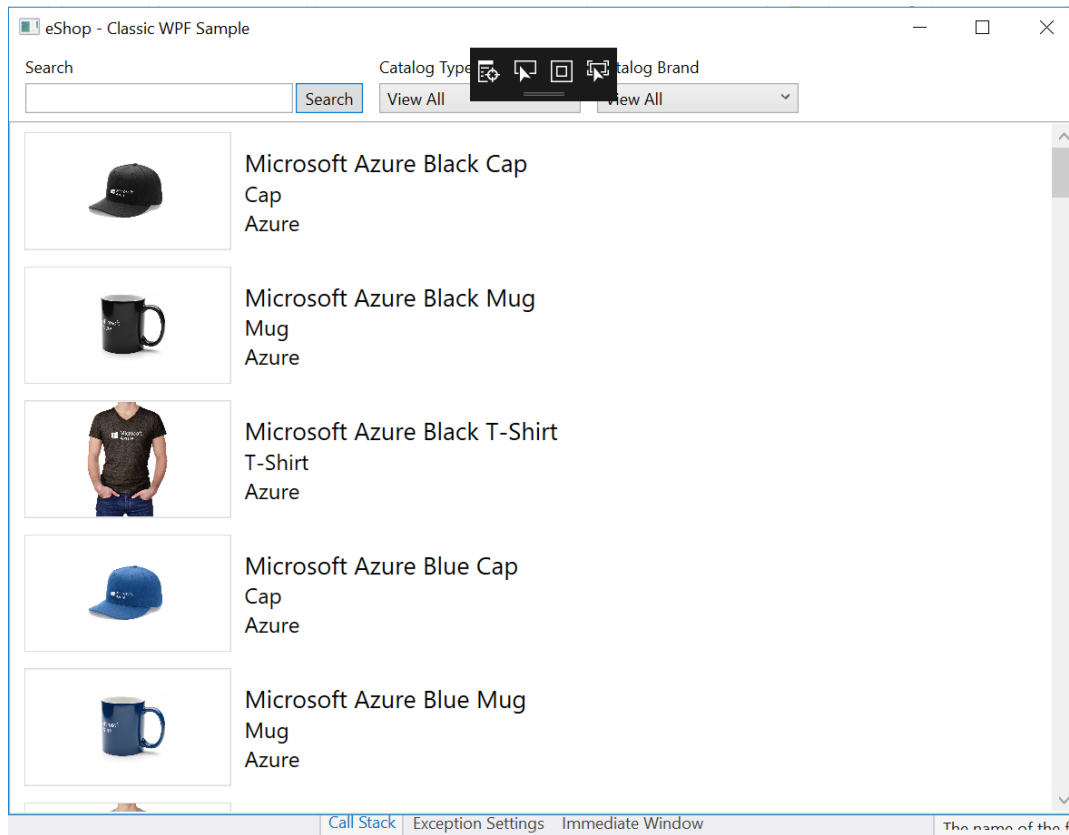
to

```
string image_name = Enviroment.CurrentDirectory + "../../../../Assets\\Images\\Catalog\\" +  
catalogItems.Picturefilename;
```

After this change, we check that the application launches and performs as expected on .NET Core.

Migrating a WPF Application

In this case, we will migrate Shop.ClassicWPF application. Here is a screenshot of the app before migration.



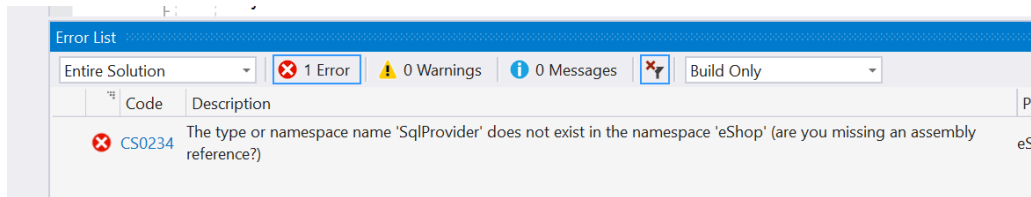
This application uses a local SQL Server Express database to hold the product catalog information. This database is accessed directly from the WPF application.

We must first update the .csproj file to the new SDK style used by .NET Core. To do this we follow the same steps described in the Windows Forms migration, we unload the project, open the .csproj file, update its contents, and finally reload the project.

In this case, we delete all the content of the .csproj file and replace it with:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWPF>true</UseWPF>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  </PropertyGroup>
</Project>
```

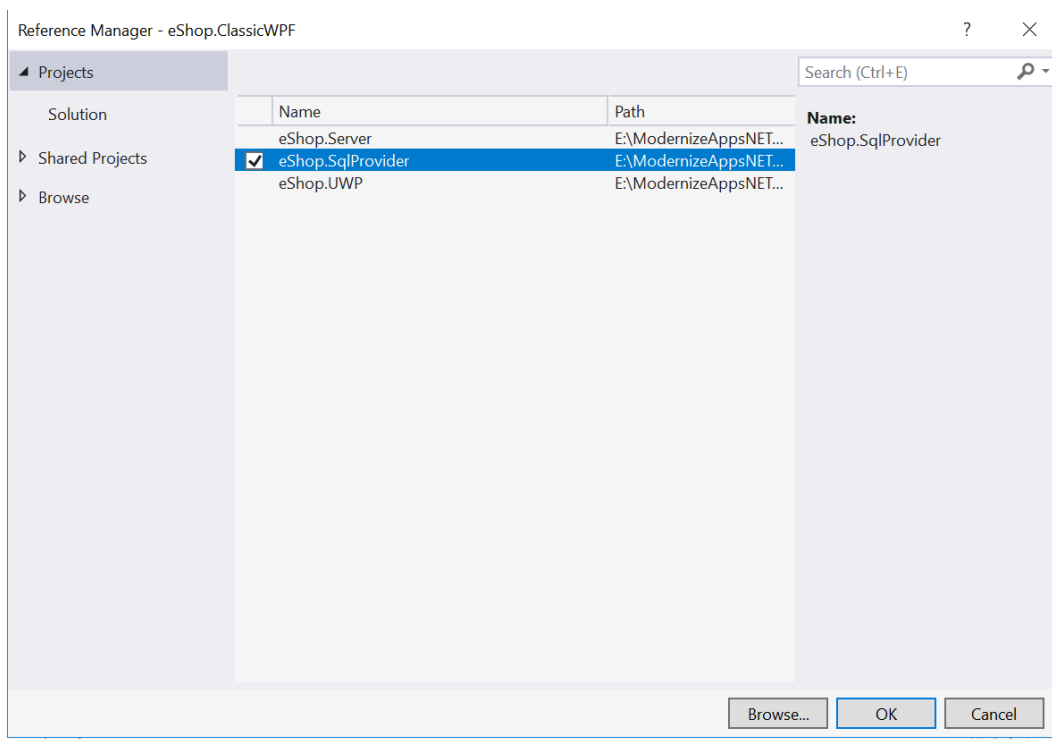
If we reload the project and compile, we get the following error:



Since we have deleted all the .csproj contents we have lost a project reference present in the old project. We just need to add this line to .csproj to include it:

```
<ItemGroup>
  <ProjectReference Include="..\eShop.SqlProvider\eShop.SqlProvider.csproj" />
</ItemGroup>
```

Alternatively, we can let Visual Studio help us by clicking on Add Reference option and select the project from the solution:



Once we do this the application compiles and executes as expected on .NET Core.

Deploying Modern Desktop Applications

Introduction

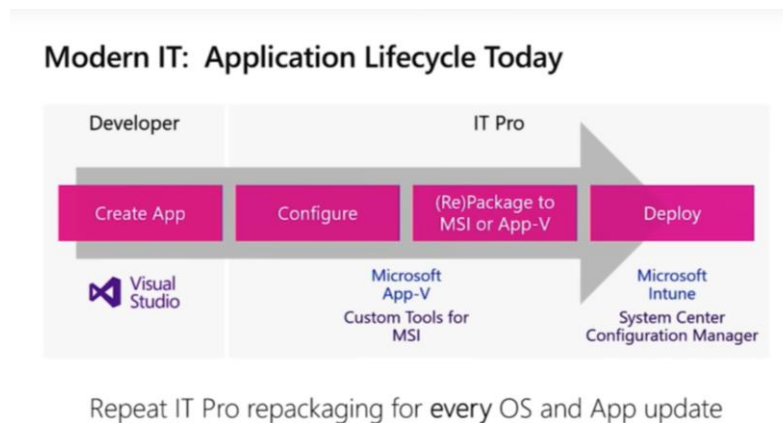
Being a desktop developer, you probably know that developing an application goes further than just writing your code. You also should think about how your application is going to be packaged and deployed to the users' machines. Sometimes this task falls under the umbrella of the IT department. Nevertheless, it is important to know about the latest solutions available out there, since packaging is fundamental for the enterprise IT with lots of money invested there. Modernization in this area can help your company to reduce the time to market between a business requirement is defined and the moment you deliver the feature to your client.

In this chapter, we talk about MSIX the brand-new technology from Microsoft that tries to capture the best of previous technologies to provide a solid foundation for the packaging technology of the future.

The modern application lifecycle

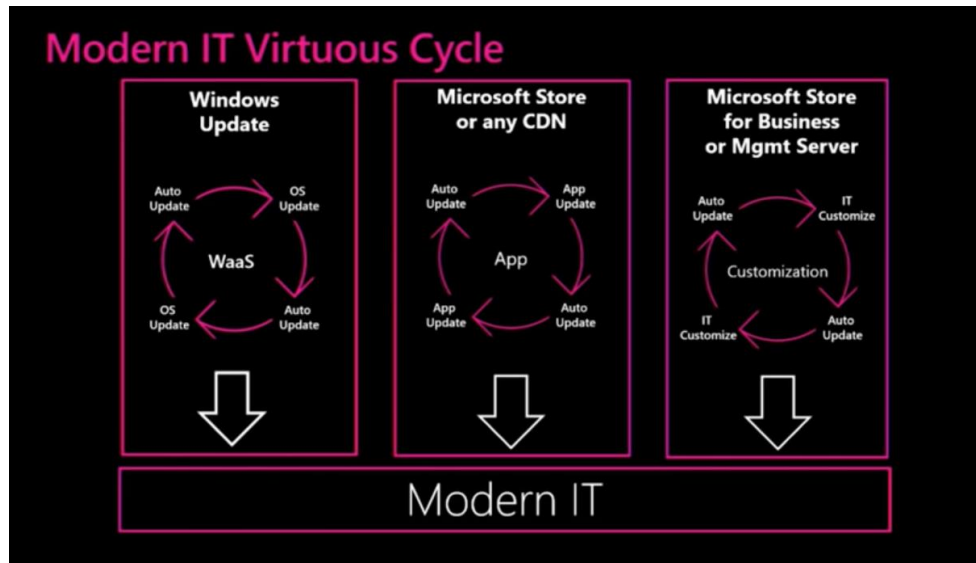
Today, developers write and build the code for an app and then pass the generated assets to the IT pros who reconfigure the app and repackage it typically in an MSI or more recent App-V packaging format, which is deployed through different channels and tools. One of the main problems with this approach is commonly known as “packaging paralysis”. The problem is that this cycle repeats every time there is an app update or an OS update.

We can see the process reflected on this picture:



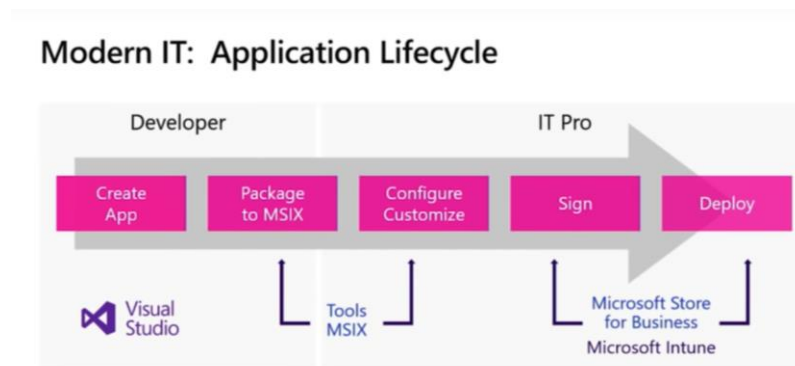
Companies need a way to break this packaging cycle with three independent cycles:

- OS updates
- Application updates
- Customization



This means we should be able to update the underlying OS without having to repackage our apps. We should also be able to use customizations from IT without the need to repackage the original developer package.

This radical change leads us to the new and modern IT lifecycle depicted in the following picture:



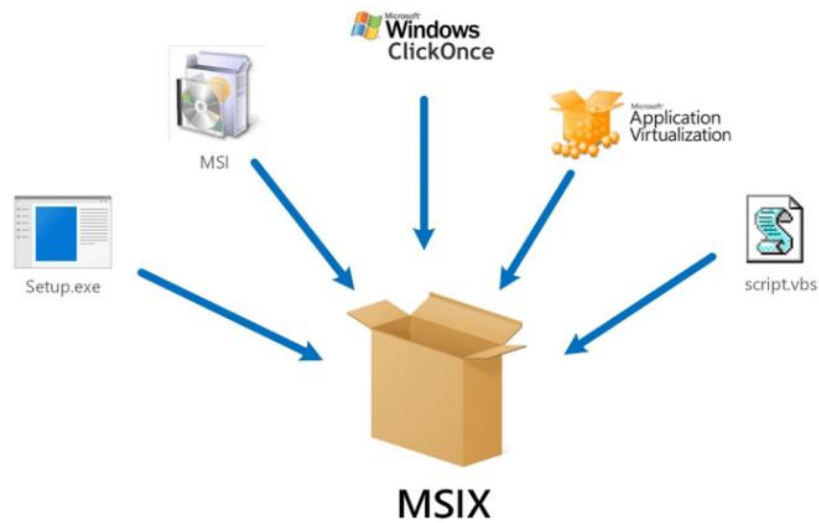
Developers create the app and generate an MSIX package that IT pros can consume and configure without the need of repackaging.

MSIX: The next generation of deployment

Before MSIX, we had several packaging technologies available like setup wizards, MSI, ClickOnce, App-V and scripting. Each of those have their strengths and Microsoft has decided to pick the best feature of all to build MSIX. Therefore, MSIX is built on the foundations of these existing technologies:

- App-V => Containerization
- ClickOnce => Auto updating
- MSI => Easy to distribute

With MSIX, we get one installer technology with all these features.



Benefits of MSIX

Never regret installing an App

MSIX provides a predictable, reliable, and safe deployment of your app. The declarative method contained in the package manifest lets the OS keep track of every asset your application needs. It also provides a true clean uninstall with no side effects.

Disk space optimization

MSIX is optimized to reduce the footprint the application has on the user's machine. It creates a single instance storage of your files, meaning that if you have two different packages with the same DLL, it is not installed twice. The platform takes care of that because it knows all the files that are installed by an app thanks to its declarative nature. It also allows you to have different versions of a DLL working side by side.

With the use of Resource packages, you can easily create multilingual apps and the OS takes care of installing the ones that are used.

Network optimization

MSIX detects the differences on the files at the byte-level enabling a feature called Differential Updates. What this means is that only the updated bytes are downloaded on application updates.

Differential updates



With Streaming installation, the user can start working on your application quickly while other parts of the app are downloaded on the background. This feature contributes to a very engaging experience for your users.

Through optional packages feature you achieve componentization on your app deployment so you can download them when needed.

Simple packaging and deployment

The AppManifest declares the versioning, device targeting, and an identify every application in a standard way. It also provides a way to sign your assets providing a solid security foundation.

OS Managed

The OS handles install, update, and removal of an application. Applications are installed per user but downloaded only once, minimizing the disk footprint.

Windows provides integrity for the app

With the use of digital signatures, you can guarantee that you don't install an application from an untrusted source. MSIX also prevents tampering because it keeps a record of file hashes and detects when the file has been modified after installation.

Works for the entire App catalog

One of the coolest things about MSIX is that it works for the entire application catalog, Windows Forms, WPF, MFC/ATL, Delphi, ClickOnce, the Microsoft Store, even xCopy deployments, you can use the same MSIX package.

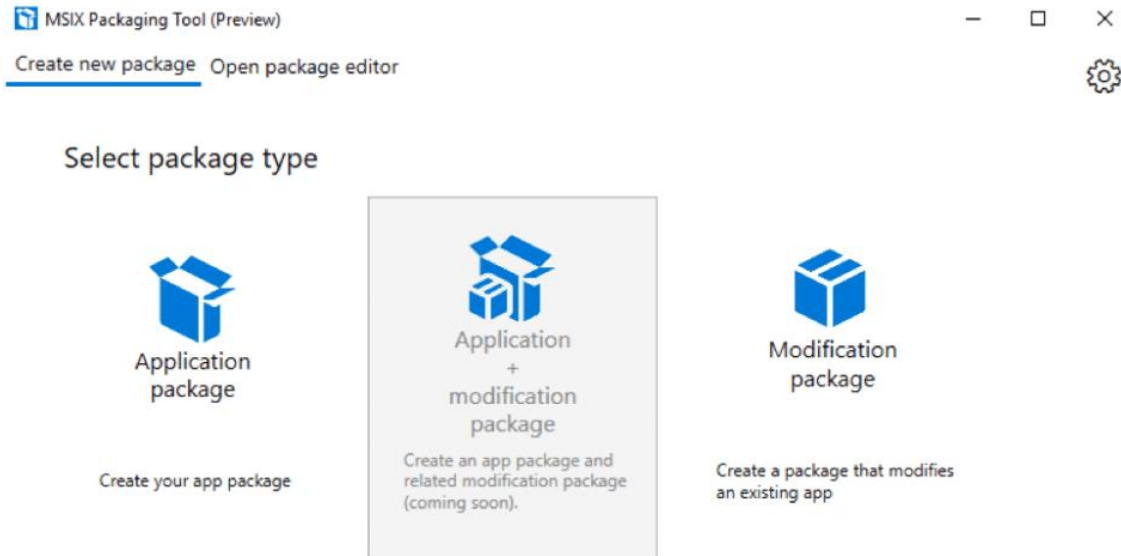
Tools

Windows Application Packaging Project

You can use the **Windows Application Packaging Project** project in Visual Studio to generate a package for your desktop app. Then, you can publish that package to the Microsoft Store or sideload it onto one or more PCs.

MSIX Packaging Tool

The MSIX Packaging Tool enables you to repackage your existing Win32 applications in the MSIX format. It offers both an interactive UI and a command line interface for packaging your app and gives you the ability to convert an application without having the source code.



Package Support Framework

The Package Support Framework is an open source kit that helps you apply fixes to your existing win32 application when you don't have access to the source code, so that it can run in an MSIX container. The Package Support Framework helps your application follow the best practices of the modern runtime environment.

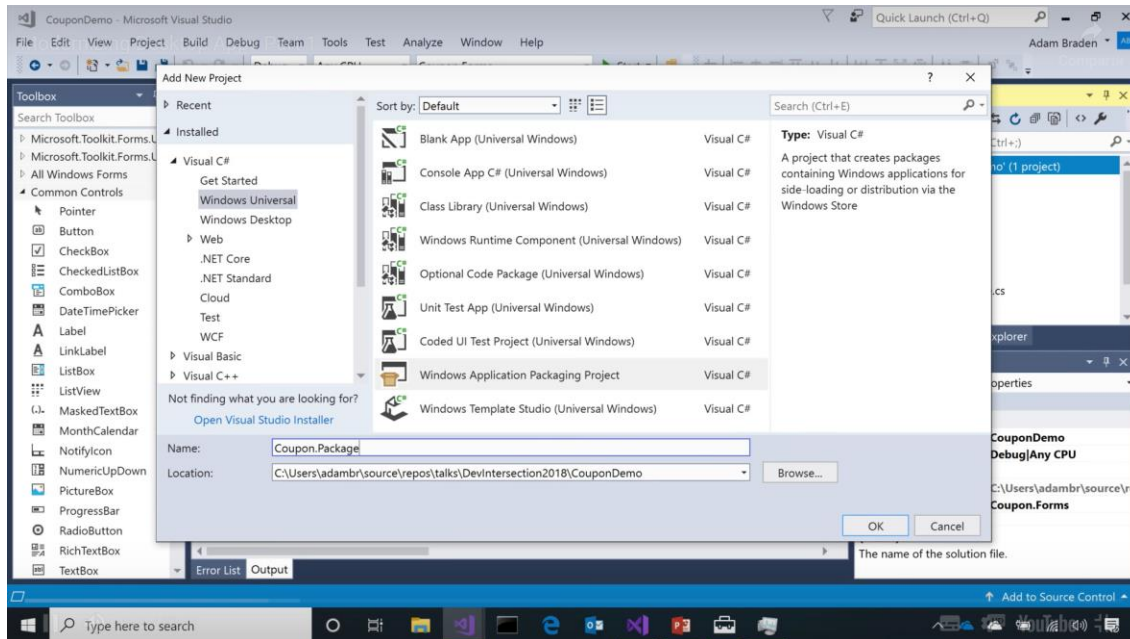
App Installer

App Installer allows for Windows 10 apps to be installed by double clicking the app package. This means that users don't need to use PowerShell or other developer tools to deploy Windows 10 apps. The App Installer can also install an application from the web along with optional packages.

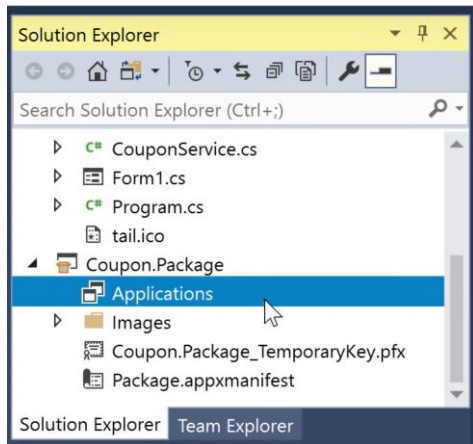
How to create a MSIX package from an existing Win32 desktop application

Let's see the steps you must take to create a MSIX package for an existing Win32 application, a Windows Forms app in this case.

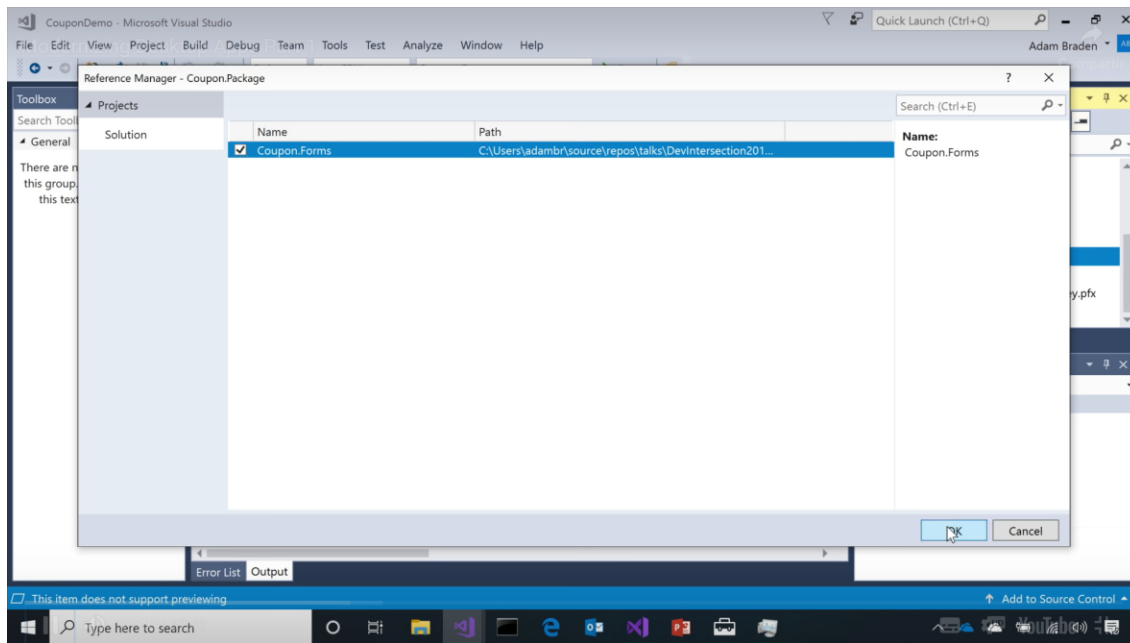
To start, add a new project to your solution, select the Windows Application Packaging Project and give it a name.



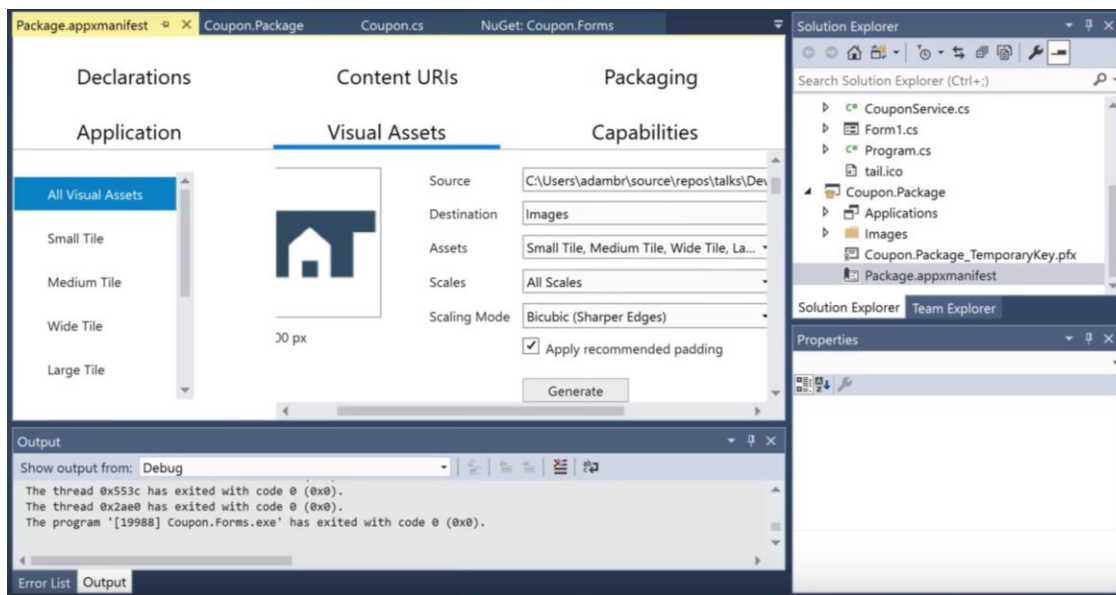
You will see the structure of the packaging project and note a special folder called Applications. Inside this folder is where you specify which are the applications you want to include in the package, you can include more than one.



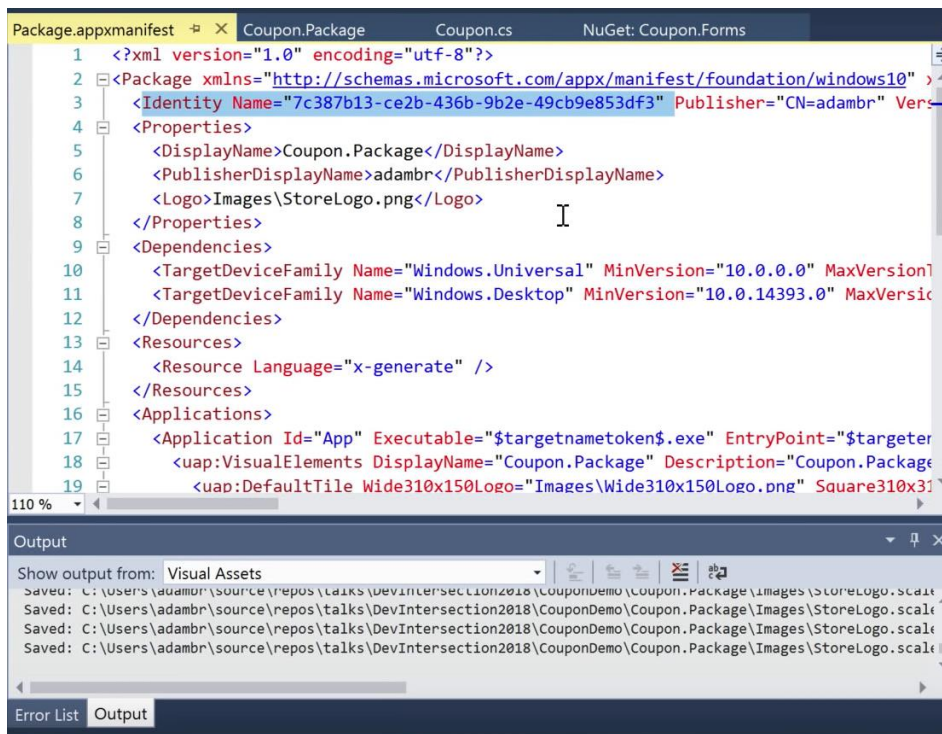
Right-click on the **Applications** folder and select the Windows Forms project we want to pack that is present in the Visual Studio solution.



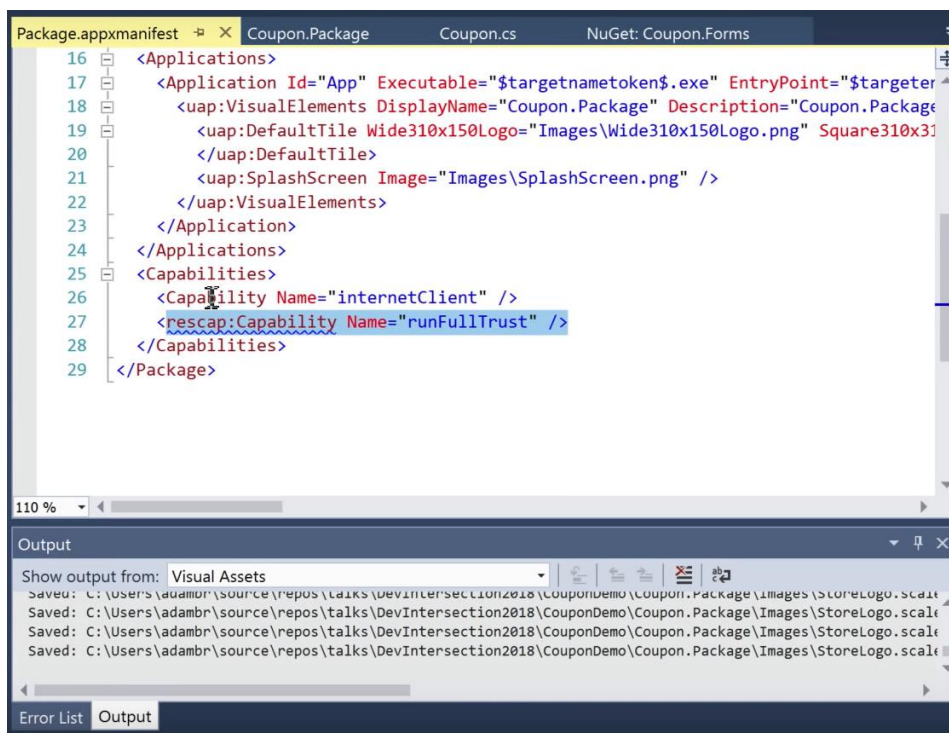
At this point, you can compile and generate the package but let us look at a couple of things. To have a better user experience, Visual Studio can generate all the visual assets a modern application needs to handle icons and tiles for the tile bar and start menu. By opening the Package.appxmanifest file in Visual Studio you can access the Manifest Designer and generate all the visual assets from a given image present on your project just by clicking **Create**.



If you open the code for the Package.appxmanifest you can see a couple of interesting things.



First is the Identity node. This is where your packaged application sets its identity to let the OS manage it.



Second is the Capabilities node. You can find all the requirements the application needs, paying special attention to the `<rescap:Capability Name="runFullTrust" />` element which tells the OS to run the app in full trust mode, as is common with a Win32 application.

Set the packaging project as the startup project for the solution and hit **Run**. This is going to compile the Windows Forms application, create an MSIX package, deploy the package, install it locally on the development machine, and launch the app.



With this, you have the fully integrated Windows 10 install and uninstall experience that MSIX provides.

But, how do you deploy the MSIX package to another machine? Right-click on the packaging project, select the Store menu, and then click on Create App Packages option.

Then, choose between creating a package to upload to the store, or, as is common with most modernization scenarios, choose "I want to create packages for sideloading".

Create App Packages

Select and Configure Packages

Output location:
 ...

Version:
 . . .

☒ Automatically increment
[More information](#)

Generate app bundle:

[What does an app bundle mean?](#)

Select the packages to create and the solution configuration mappings:

	Architecture	Solution Configuration
<input checked="" type="checkbox"/>	Neutral	Debug (Any CPU)
<input type="checkbox"/>	x86	Debug (x86)
<input type="checkbox"/>	x64	Debug (x64)
<input type="checkbox"/>	ARM	None

i To run validation locally, you must select at least one solution configuration that is both non-Debug and contains an architecture that runs on the local machine.

☒ Include full PDB symbol files, if any, to enable crash analytics for the app. [Learn More](#)

Previous Next Cancel

There you can select the different architectures you want to target. You can include as many as you want into the same MSIX package.

The final step is to declare where you want to deploy the final installation assets.

Create App Packages

Configure Update Settings

Installation URL: http(s) or UNC path or file share:

How can I publish my application?
Specify how often the application should check for updates:

☒ Check everytime the application runs

☐ Check every: Day(s)

Previous Create Cancel

You can choose to use a web server or a shared UNC path on your enterprise file servers. Pay attention to the settings you can specify to determine how you want to update your application. We will be discussing application updates in the next section.

For a detail step by step guide please refer to:

<https://docs.microsoft.com/windows/msix/desktop/desktop-to-uwp-packaging-dot-net>

Auto Updates in MSIX

The Windows Store has a great updating mechanism using Windows Update. In most enterprise scenarios, you don't use the Store to distribute your desktop apps, so you need a similar way to configure updates for your application and distribute them to your users.

Using a combination of Windows 10 features and MSIX packages, you can provide a great update experience for your users. In fact, the user does not need to be technical to benefit from a seamless application update experience.

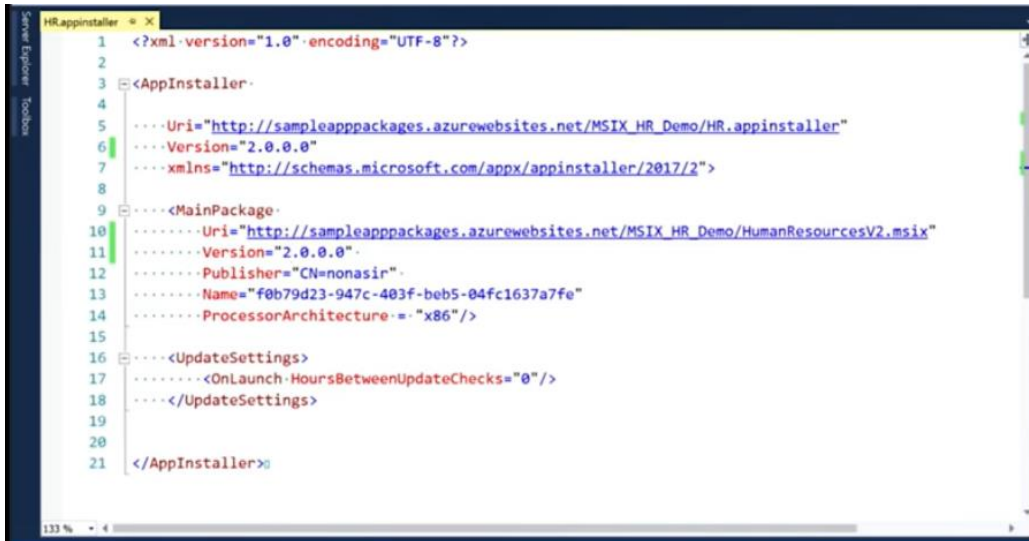
You can configure your updates to interact with the user in two different ways:

- User prompted updates: The OS shows a nice auto generated UI to notify the user the application is about to install. It builds this UI based on the properties you specify on your installation files.
- Silent updates in the background. With this option, your users don't need to be aware of the updates.

You can also configure when you want to perform updates, each time the application is run, or on a regular interval. Thanks to the side-loading features of Windows 10, you can even get these updates while the application is running.

When you use this type of deployment, a special file is created named *.appinstaller*. This simple file contains the following sections:

- The location of the *.appinstaller* file.
- The application's main MSIX package properties.
- The update behavior.

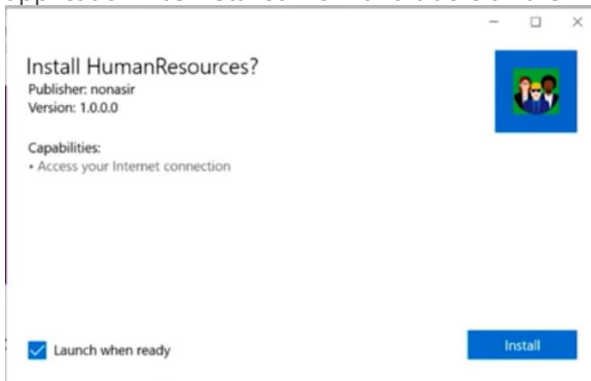


```
1 <?xml:version="1.0" encoding="UTF-8"?>
2
3 <AppInstaller>
4
5   <Uri="http://sampleapppackages.azurewebsites.net/MSIX_HR_Demo/HR.appinstaller"
6   <Version="2.0.0.0"
7   <xm:ns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
8
9   <MainPackage>
10    <Uri="http://sampleapppackages.azurewebsites.net/MSIX_HR_Demo/HumanResourcesV2.msix"
11    <Version="2.0.0.0"
12    <Publisher="CN=nonasir"
13    <Name="f0b79d23-947c-403f-beb5-04fc1637a7fe"
14    <ProcessorArchitecture="x86"/>
15
16    <UpdateSettings>
17      <OnLaunch>
18        <HoursBetweenUpdateChecks="0"/>
19      </OnLaunch>
20    </UpdateSettings>
21  </AppInstaller>
```

In combination with this file, Microsoft has designed a special URL protocol to launch the installation process from a link:

<ms-appinstaller:?source=http://mywebservice.azureedge.net/MyApp.msix> Install app package

This protocol works on all browsers and launches the installation process with a good user experience on Windows 10. Since the OS manages the installation process, it is aware of the location the application was installed from and tracks all the files affected by the process.



The process to distribute a new version of your app is simple. After you have generated the new MSIX package, move it to the deployment server. Next, edit the *.appinstaller* file to reflect this changes, mainly the version and the path to the new MSIX file. The next time the user launches the application the system is going to detect the change and download the files for the new version in the

background. When this is done, installation will execute on new application launch transparently for your user.