Microsoft

Modernizing Desktop Apps on Windows 10 with .NET Core 3.1

Olia Gavrysh

Miguel Angel Castejón Domínguez

Version 1.0

PUBLISHED BY

Microsoft Developer Division, .NET, and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Co-Authors:

Olia Gavrysh, Program Manager, .NET team, Microsoft

Miguel Angel Castejón Dominguez, Innovation Architect, Kabel

Participants and reviewers:

Maira Wenzel, Senior Program Manager, .NET team, Microsoft

Andy De Gorge, Senior Content Developer, .NET docs team, Microsoft

Miguel Ramos, Senior Program Manager, Windows Developer Platform team, Microsoft

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

Ricardo Minguez Pablos, Senior Program Manager, Azure IoT team, Microsoft

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

Beth Massi, Senior Product Marketing Manager, Microsoft

Scott Hunter, Partner Director Program Manager, .NET team, Microsoft

Marta Fuentes Lara, Kabel

Raúl Fernández de Córdoba, Kabel

Antonio Manuel Fernández Cantos, Kabel

Introduction

This book 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'll discover that there's 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 won't even require major modifications of your code. In this book, we'll reveal how all those features work behind the scenes and explain the mechanics of their implementations. Moreover, you'll find some common scenarios for modernizing existing desktop applications shown in detail so you can find inspiration for evolving your projects.

Microsoft's 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 can mix and match the strategies to best address your application needs.

Who should use the book

We wrote this book for developers and solution architects who want to modernize existing Windows Forms and WPF desktop applications to leverage the benefits of .NET Core and Windows 10.

You might also find this book useful if you're a technical decision maker, such as an enterprise architect or a development lead or director who wants an overview of the benefits of updating existing desktop applications.

How to use the book

This book addresses the "why"—why you might want to modernize your existing applications, and the specific benefits you get from using NET Core 3.1 and MSIX to modernize your desktop apps. The content of the book 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 screenshots are provided, with chapter 5 devoted to showcase a complete migration process for sample applications.

What this book doesn't cover

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

This book isn't about developing modern applications with .NET Core from scratch or about getting started with Windows Forms and WPF. It focuses on how you can update existing desktop applications with the latest technologies for desktop development.

Samples used in this book

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

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

Send your feedback

This book and related samples are constantly evolving, so your feedback is welcomed! If you have comments about how this book can be improved, use the feedback section at the bottom of any page built on GitHub issues.

Contents

Why modern desktop applications 1

Introduction 1

A story of one company 1

Your story 1

Desktop applications nowadays 1

Windows Forms 3

WPF 4

UWP 4

A tale of two platforms 5

Paths to modernization 6

Modern features 7

Deployment 7

Installation 7

What’s new with .NET Core for Desktop? 8

The motivation behind .NET Core 8

Introduction to .NET Core 9

.NET Framework vs. .NET Core 10

.NET Standard vs. PCL 12

New Desktop features in .NET Core 12

Support for Windows Forms and WPF 12

XAML Islands 13

Access to all Windows 10 APIs 13

Side-by-side support and self-contained EXEs 13

Performance 14

Benefits of open source 14

Migrating Modern Desktop applications 15

Configuration files 15

Configuration on .NET Framework 15

Configuration on .NET Core ... 16

Migrating Configuration files ... 16

Accessing databases ... 17

ODBC ... 18

OLE DB ... 18

ADO.NET ... 18

EF Core vs. EF6 ... 18

Relational databases ... 19

Consuming services ... 19

Consuming a COM Object ... 20

More things to consider ... 20

AppDomains ... 20

Remoting ... 20

Code Access Security (CAS) ... 21

Security Transparency ... 21

Windows 10 migration ... 22

WinRT APIs ... 22

Universal Windows Platform (UWP) packages ... 23

How to add WinRT APIs to your desktop project ... 25

XAML Islands ... 32

The road to XAML Islands ... 33

How it works ... 33

Do you need XAML Islands ... 36

The road ahead XAML Islands: WinUI 3.0 ... 36

Example of migrating to .NET Core 3.1 ... 39

Migration process overview ... 39

Preparation ... 39

Migrate project file ... 41

Fix the code and build ... 42

Run and test ... 43

Migrating a Windows Forms application ... 43

Migrating a WPF application ... 49

Deploying Modern Desktop Applications 51

The modern application lifecycle... 52

MSIX: The next generation of deployment.. 54

Benefits of MSIX 54

Tools.. 56

How to create an MSIX package from an existing Win32 desktop application.. 57

Auto Updates in MSIX.. 64

CHAPTER 1

Why modern desktop applications

Introduction

A story of one company

Back in early 2000s, 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 (also known as WinForms) for their application development. Over the 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'd 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 or Windows Presentation Foundation (WPF) 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's 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!

Desktop applications nowadays

Before the raise of the Internet, desktop applications were the main approach to build software systems. Developers could choose any programming language, such as COBOL, Fortran, VB6, or C++.

But where they developed small tools or complex distributed architectures, they were all desktop applications.

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 a 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 weren't designed 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 and state management.

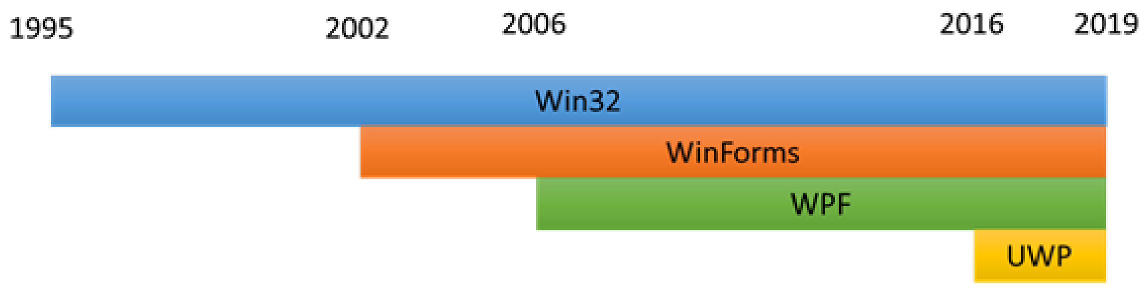
Even though web and mobile applications have grown at an incredible pace, for certain tasks desktop applications still hold the number one place in terms of efficiency and performance. That explains why there are millions of developers who are building their projects with WPF and WinForms and the amount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

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

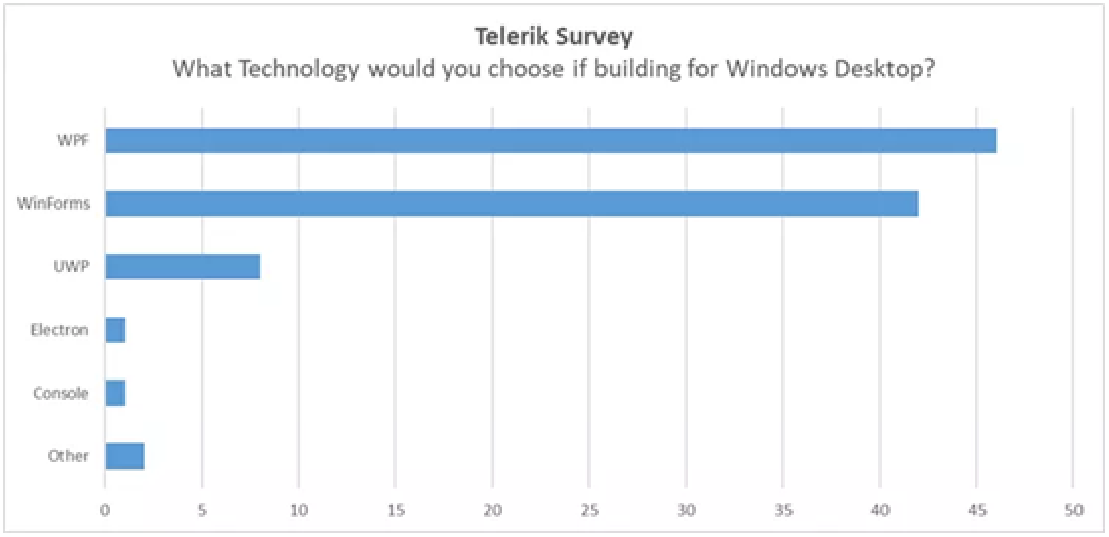
- Desktop apps have better interaction with user's PC.
- The performance of desktop applications for complex calculations is much higher than performance of web applications.
- Running custom logic on the client side is possible but much harder with a web application.
- Using multithreading is easier and more efficient in a desktop application.
- The learning curve for designing user interfaces (UIs) isn't steep. And for WinForms, it's completely intuitive with drag-and-drop experience of the Windows Forms designer.
- It's 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, such as camera, Bluetooth, or card readers, is easy.
- Since the technology has been around for a while, there are many experts and a knowledge base available to develop 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 many UI desktop technologies throughout the years from Win32 introduced in 1995 to Universal Windows Platform (UWP) released in 2016.



According to a survey published by Telerik on April 2016, the most popular technologies for building Windows desktop apps are Windows Forms, WPF, and UWP.



You can develop in any of them using C# and Visual Basic, 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 graphics device interface (GDI) engine. It offers a 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 isn't trivial.

The following list summarizes the main characteristics of Windows Forms:

- Mature technology with lots of code samples and documentation.
- 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 and later versions.

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 large applications. Like Windows Forms, it's a managed framework, but the design is modular and reusable.

Here are the main features of WPF:

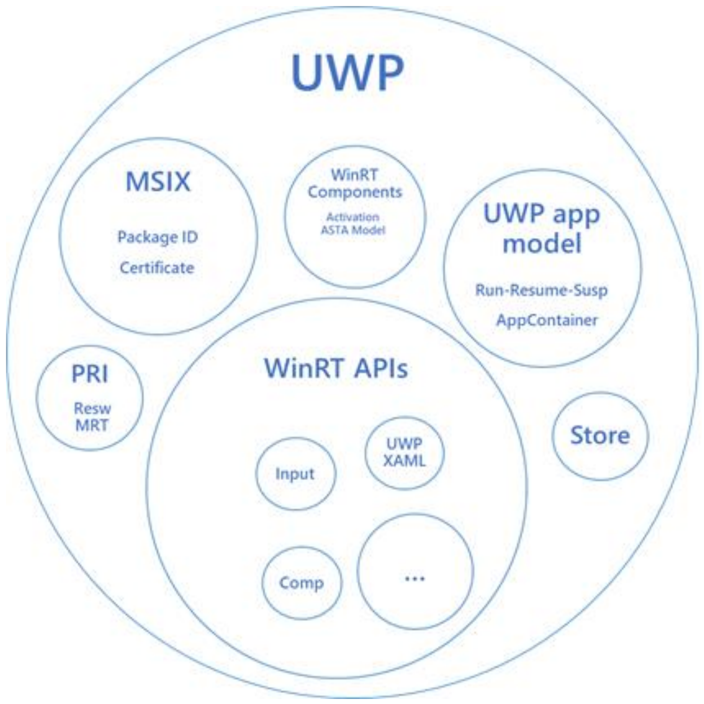
- Mature technology.
- Designer is available, but developers usually prefer to create the design from code using declarative XAML.
- The learning curve is steeper than Windows Forms.
- Supported on any Windows version.
- Supported on .NET Core 3.0 and later versions.

UWP

UWP isn't only a presentation framework like WPF and Windows Forms, but it's 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).

The platform was created to support all kind of input systems (like ink, touch, gamepad, mouse, keyboard, gaze, and so on) 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 control what resources a UWP app can access.
- Supported only on Windows 10.
- Apps can be deployed through Microsoft Store for easier deployment.
- Designed as part of the Windows Runtime API.
- Contains an extensive set of rich built-in controls and additional controls available through the Microsoft UI Library NuGet packages (WinUI library) updated every few months.

A tale of two platforms

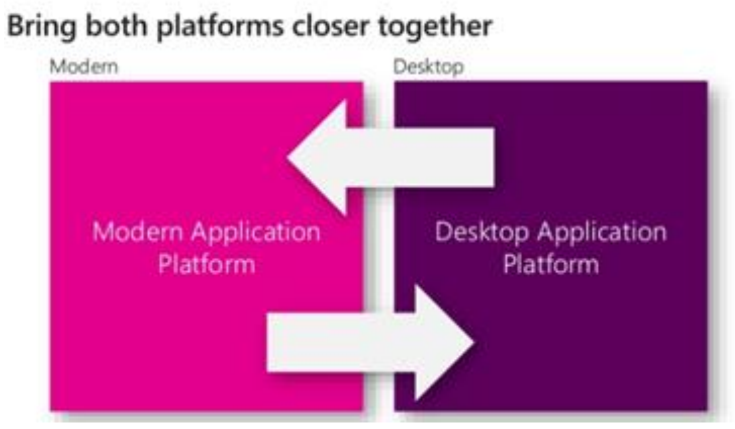
In the last 20 years, while UI desktop technologies were growing and following the path from Windows Forms to UWP, the hardware was also evolving from heavy weight PC units with small CRT monitors to high-DPI monitors and lightweight tablets and phones with different data input techniques like Touch and Ink. These changes 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.

The following table describes the differences between the two concepts:

	Modern Application	**Desktop Application**
Security	Contained execution & Great Fundamentals. Designed from the ground up to respect user's 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 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's performing.	Local Data. Traditional desktop applications usually need some local data.

	Modern Application	Desktop Application
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 put 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
Modern
Desktop
Modern Application Platform
Desktop Application Platform

1. Move Desktop Application scenarios into Modern Application platform. The traditional desktop development is still popular because it addresses certain scenarios well. It makes sense to take these common desktop scenarios and bring them into the modern desktop platform to make the platform fully capable.

![](C:\Users\Miguel\source\repos\dotnet-architecture\ebook-private-resources\modernize-desktop\content\media\why-modern-applications\desktop-to-modern.png)

1. 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.

![](C:\Users\Miguel\source\repos\dotnet-architecture\ebook-private-resources\modernize-desktop\content\media\why-modern-applications\modern-to-desktop.png)

In this book, we'll 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 the order using a tablet pen. Inking is native in today's operating systems and technologies, but it wasn't 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 can't 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. It has limited the innovation pace for developers forcing them to stay on outdated versions of the .NET Framework.

Since the launch of .NET Core 3.0, you can leverage a 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 fact 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'll introduce a new way of installing applications with MSIX that solves the problem described earlier. You'll learn how you can easily set up a packaging, installation, and updates for your application.

What's new with .NET Core for Desktop?

Starting with .NET Core 3.0, .NET Core supports Windows Forms and WPF. So, now you have a choice between .NET Framework and .NET Core for your desktop applications. This chapter will describe what is .NET Core and what are its benefits for desktop applications.

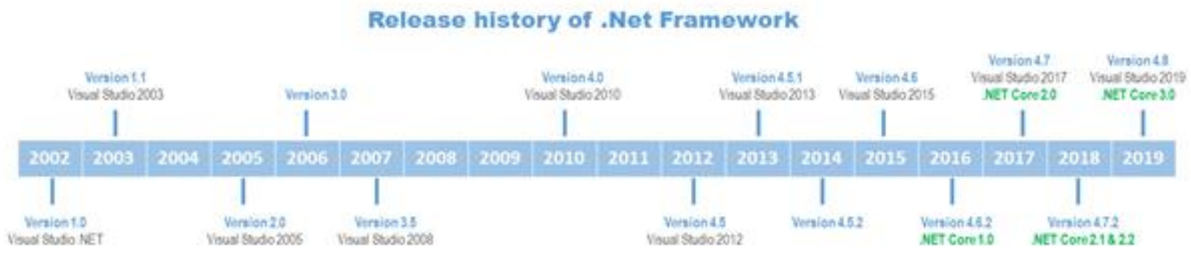
The motivation behind .NET Core

Since its launch in 2002, .NET Framework has evolved through the years to support many technologies like Windows Forms, ASP.NET, Entity Framework, Windows Store, and many others. All of them are 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. That lead 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're 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 back-end 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 Portable Class Libraries (PCLs) was born. Originally, the .NET Framework was designed around the assumption that it would always be deployed as a single unit, so factoring wasn't 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 designed with proper factoring in mind taking care of the dependencies between them.

This leads to 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 based on the API shape. And the most pressing need to create libraries running on different verticals is also addressed. But there's a great challenge: APIs are only portable when the implementation is moved forward across all the verticals.

A better approach is to unify the implementations across verticals by providing a well-factored implementation instead of a well-factored view. It's a lot simpler to ask each team that owns 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 the next section.

Another large 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 deployment model has many advantages, such as reducing disk space and centralized access to services, it presents some pitfalls.

To start with, it's difficult for application developers to take a dependency on a recently released framework. They either have to take a dependency on the latest OS or provide an application installer that installs 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're willing to go through the trouble of providing an installer 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 can change how this type is serialized and cause breaking problems depending on the existing code. Because the 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. It runs on Windows, macOS, and Linux with plans to also run on ARM-based architectures like Android and IoT.

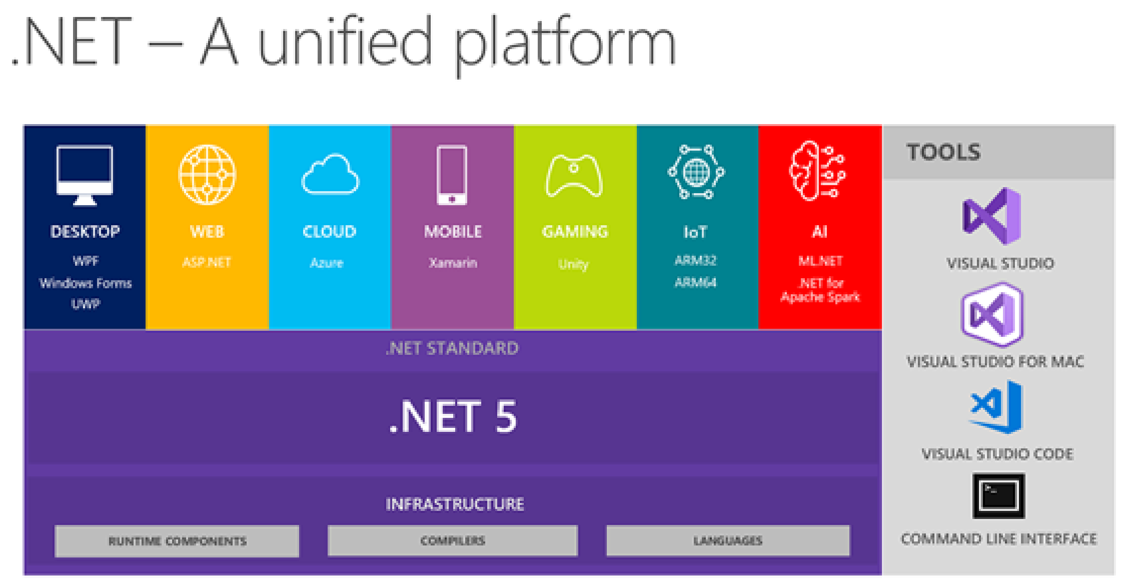
The purpose of .NET Core is to provide a unified platform for all types of applications, which includes Windows, cross-platform, and mobile applications. .NET Standard enables this by providing shared base APIs, which every application model needs, and excluding any application model-specific API.

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.
- **Supported:** Microsoft officially supports .NET Core.

Starting with .NET Core 3.0, besides the existing support for web and cloud, there's also support for desktop, IoT, and AI domains. The goal for this framework is impressive: to target every type of .NET development present and future. Microsoft plans to complete this vision with .NET 5 at the end of 2020. The "Core" name was removed to reinforce its uniqueness in the .NET world.



.NET Framework vs. .NET Core

So now that you understand the relevance of .NET Core inside the Microsoft strategy for .NET, you might be wondering what happens with .NET Framework. You could be 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?

In 2019 the last version of the **.NET Framework - 4.8** was released. It included three major improvements for desktop applications:

- **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, new controls were added that take advantage of Microsoft Edge and newer media players that support the latest standards.
- **Access to UWP controls**: UWP contains new controls that take advantage of the latest Windows features and touch displays. You don't have to rewrite your applications to use these new features and controls, so you can use these new features in your existing WPF or Windows Forms code.
- **High-DPI improvements**: The resolution of displays is increasing to 4K and 8K resolutions. So, .NET Framework 4.8 adds new HDPI improvements to make sure your existing Windows Forms and WPF applications can look great on these new displays.

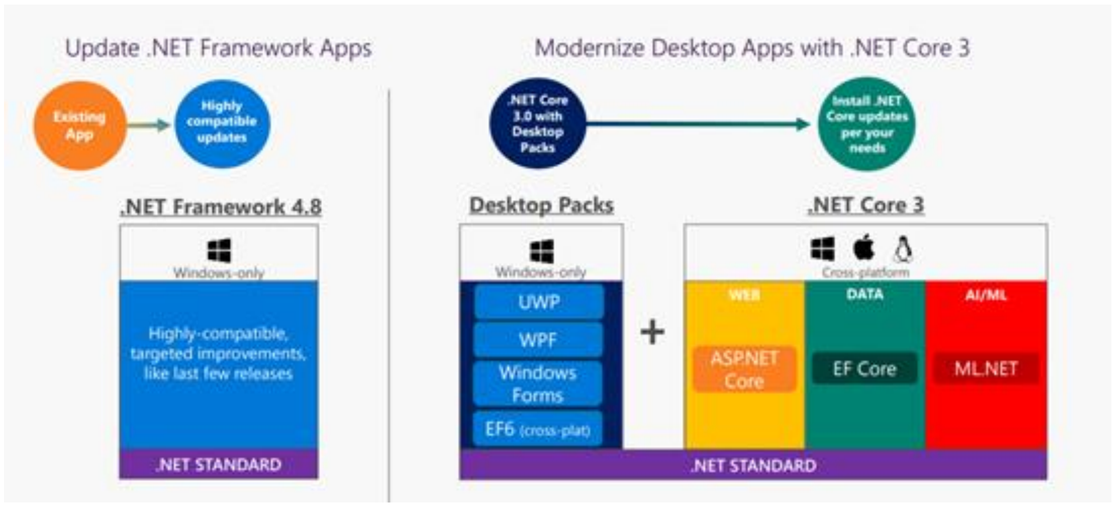
Since .NET Framework is installed on millions of machines, Microsoft will continue to support it but won't add new features.

.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 won't. Also, **.NET Core** already has features that were impossible for .NET Framework, such as:

- **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. Multiple versions of .NET Core can be installed on the same machine and each application specifies which version of .NET Core it should 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 deploy .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 Base Class Libraries (BCL) improvements without the risk of breaking compatibility. Now Windows Forms and WPF applications can take advantage of the latest .NET Core features, which also includes more fundamental fixes for runtime performance, high-DPI support, and so on.

An essential part of the roadmap for Microsoft was to ease developers to move applications to .NET Core and in future to .NET 5. But if you have existing .NET Framework applications, you shouldn't feel pressured to move to .NET Core. .NET Framework will be fully supported and will always be a part of Windows. However, if you want to use the newest language features and APIs in the future, you'll need to move your applications to .NET Core.

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


Update .NET Framework Apps
Existing App
Highly compatible updates
.NET Framework 4.8
Windows-only
Highly-compatible, targeted improvements, like last few releases
.NET STANDARD
Modernize Desktop Apps with .NET Core 3
.NET Core 3.0 with Desktop Packs
Install .NET Core updates per your needs
Desktop Packs
Windows-only
UWP
WPF
Windows Forms
EF6 (cross-plat)
.NET Core 3
Cross-platform
WEB
DATA
AI/ML
ASP.NET Core
EF Core
ML.NET
.NET STANDARD

.NET Standard vs. 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.

.NET Standard is the evolution of PCLs and the following list shows the fundamental differences between .NET Standard and PCLs:

- .NET Standard is a set of curated APIs, picked by Microsoft. PCLs aren't.
- The APIs that a PCL contains are dependent on the platforms that you choose to target when you create it. This makes a PCL only sharable for the specific targets that you choose.
- .NET Standard is platform-agnostic, it can run anywhere, on Windows, macOS, 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.

New Desktop features in .NET Core

Support for Windows Forms and WPF

Windows Forms and WPF are part of .NET Core since version 3.0. Both presentation frameworks are for Windows only, so they aren't 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. So Windows Forms and WPF are available for .NET Core and .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 .NET Core 3.0. As expected, .NET Core 3.x versions support .NET Standard 2.1 and earlier versions.

Also, it's important to notice that both Windows Forms and WPF implementations for .NET Core are open source.

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, version 1903 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 make it easier 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 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. These APIs give access to a wide variety of functionality like Authentication, Bluetooth, Appointments, and 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. 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 requested 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 start, 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 take advantage of these performance improvements when you use .NET Core 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 complex that needs the interaction of specialized teams on each of the targeted platforms. This effort needs much 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 previously mentioned: To be the single .NET platform that any developer will ever need to build any application.

Migrating Modern Desktop applications

In this chapter, we're 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 doesn't work in isolation and needs some kind of interaction with subsystems that may reside on the local machine or on a remote server. It will probably need some kind of database to connect as a persistence storage either locally or remotely. With the raise of Internet and service-oriented architectures, it's 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 file system to implement some functionality. Alternatively, maybe you're using a piece of functionality that resides inside a COM object outside your application, which is a common scenario if, for example, you're integrating Office assemblies in your app.

Besides, there are differences in the API surface that is exposed by .NET Framework and .NET Core, and some features that are available on .NET Framework aren't available on .NET Core. So, it's important for you to know and take them into account when planning a migration.

Configuration files

Configuration files offer the possibility to store sets of properties that are read at run time and affect the behavior of our apps, such as where to locate a database or how many times to execute a loop. The beauty of this technique is that you can modify some aspects of the application without the need to recode and recompile. 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 AppSettingsSection class from the `System.Configuration` namespace.

Within the .NET Framework infrastructure, there's a hierarchy of configuration files that inherit properties from its parents. You can find a *machine.config* file that defines many 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's no *machine.config* file. And even though you can continue to use the old fashioned System.Configuration namespace, you may consider switching to the modern Microsoft.Extensions.Configuration, which offers a good number of enhancements.

The configuration API supports the concept of configuration provider, which defines the data source to be used to load the configuration. There are different kinds of built-in providers, such as:

- In-memory .NET objects
- INI files
- JSON files
- XML files
- Command-line arguments
- Environment variables
- Encrypted user store

Or you can build your own.

The new configuration allows a list of name-value pairs that can be grouped into a multi-level hierarchy. Any stored value maps to a string, and there's built-in binding support that allows you to deserialize settings into a custom plain old CLR object (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 manage them on a single function inside your code.

Migrating Configuration files

You can continue to use your existing app.config XML file. However, you could take this opportunity to migrate your configuration to benefit from the several enhancements made on .NET Core.

To migrate from an old-style *app.config* to a new configuration file, you should choose between an XML format and a JSON format.

If you choose XML, the conversion is straightforward. Since the content is the same, just rename the *app.config* file to a file with XML extension. Then, change the code that references AppSettings to use the `ConfigurationBuilder` class. This change should be easy.

If you want to use a JSON format and you don't want to migrate by hand, there's a tool called dotnet-config2json available on .NET Core that can convert an *app.config* file to a JSON configuration file.

You may also come across some issues when using configuration sections that were defined in the *machine.config* file. For example, consider the following 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 take this configuration to a .NET Core, you'll get an exception:

Unrecognized configuration section system.diagnostics

This exception occurs because that section and the assembly responsible for handling that section was defined in the *machine.config* file, which now doesn't exist.

To easily fix the issue, you can 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 some kind of database. For desktop, it's common to find client-server architectures with a direct connection between the desktop app and the database engine. These databases can be local or remote depending on the need to share information between different users.

From the code perspective, there have been many technologies and frameworks to give the developer the possibility to connect, query, and update a database.

The most common examples of database you can find when talking about Windows Desktop application are Microsoft Access and Microsoft SQL Server. If you have more than 20 years of experience programming for the desktop, names like ODBC, OLEDB, RDO, ADO, ADO.NET, LINQ, and Entity Framework will sound familiar.

ODBC

You can continue to use ODBC on .NET Core since Microsoft is providing the `System.Data.Odbc` library 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, and as such wasn't the best fit for a cross-platform technology such as .NET Core. It's also unsupported in SQL Server versions 2014 and later. For those reasons, OLE DB won't be supported by .NET Core.

ADO.NET

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

EF Core vs. EF6

There are two currently supported versions of Entity Framework (EF), Entity Framework 6 (EF6) and EF Core.

The latest technology released as part of the .NET Framework world is Entity Framework, with 6.4 being the latest version. With the launch of .NET Core, Microsoft also released a new data access stack based on Entity Framework and called Entity Framework Core.

You can use EF 6.4 and EF Core from both .NET Framework and .NET Core. So, what are the decision drivers to help to decide between the two?

EF 6.3 is the first version of EF6 that can run on .NET Core and work cross-platform. In fact, the main goal of this release was to make it easier to migrate existing applications that use EF6 to .NET Core.

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. For more information, see Compare EF Core & EF6.

The recommendation is to use EF Core if:

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

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

- The app will run on Windows and .NET Framework 4.0 or later.
- EF6 supports all of 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` class, fully compatible with the one existing in the .NET Framework but located in the Microsoft.Data.SqlClient library. You just have to add a reference to the Microsoft.Data.SqlClient NuGet package and do some renaming for the namespaces and everything should work as expected.

Microsoft Access

Microsoft Access has been used for years when the sophisticated and more scalable SQL Server wasn't needed. You can still connect to Microsoft Access using the System.Data.Odbc library.

Consuming services

With the raise of service-oriented architectures, desktop applications began to evolve from a client-server model to the three-layer 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-layer 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 as a way 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'll need to make some updates in your configuration file.

Second, you'll need to regenerate the service client with the new tools present on Visual Studio 2019. In this step, you must consider activating the generation of the synchronous operations to make the client compatible with your existing code.

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

If you're using the WebRequest class to perform web service calls, you may find some differences on .NET Core. The recommendation is to use the System.Net.Http.HttpClient instead.

Consuming a COM Object

Currently, there's no way to add a reference to a COM object from Visual Studio 2019 to use with .NET Core. So, you have to manually modify the project file.

Insert a `COMReference` structure inside the project file like in the following example:

```
<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 .NET Core. If your code relies on some of these technologies, consider the alternative approaches outlined in this section.

The Windows Compatibility Pack provides access to APIs that were previously available only for .NET Framework. It can be used on .NET Core and .NET Standard projects.

For more information on API compatibility, you can find documentation about breaking changes and deprecated/legacy APIs at https://docs.microsoft.com/dotnet/core/compatibility/fx-core.

AppDomains

Application domains (AppDomains) isolate apps from one another. AppDomains require runtime support and are expensive. Creating additional app domains isn't supported. 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 isn't supported on .NET Core.

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

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

Code Access Security (CAS)

Sandboxing, which relies on the runtime or the framework to constrain which resources a managed application or library uses or runs, isn't supported on .NET Core.

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

CHAPTER 4

Windows 10 migration

Consider the following situation: You have a working desktop application that was developed in the Windows 7 days. It's using WPF technology available at that time and working fine but it has an outdated UI and behaviors when you run it on Windows 10. It is like when you watch a futuristic movie like Matrix and you see Neo using the Nokia 8110 device. The film works great after 20 years but it would rather benefit from a device modernization.

With the release of Windows 10, Microsoft introduced many innovations to support scenarios like tablets and touch devices and to provide the best experience for users for a Microsoft operating system ever. For example, you can:

- Sign in with your face using Windows Hello.
- Use a pen to draw or handwrite text that is automatically recognized and digitalized.
- Run locally customized AI models built on the cloud using WinML.

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

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 there 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

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, among 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:

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

Universal Windows Platform (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, such as protocols, file types, and extensions, 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 as expected. However, classic desktop apps like native C++ or .NET apps, use different deployment systems that don't require a package identity. If you want to use these WinRT APIs in your desktop application, you need to provide them a package identity.

One way to proceed is to build an additional packaging project. Inside the packaging project, you point to the original source code project and specify the Identity information you want to provide. If you install the package and run the installed app, it will automatically get an identify 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 packaged application identity by inspecting if the type that contains the API is marked with the DualApiPartition attribute. If it is, you can call if from an unpackaged traditional desktop app. Otherwise, you must convert your classic desktop app to a UWP with the help of a packaging project.

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's removed completely with no trace left behind preventing the Windows rot problem.
- **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, because only the changed parts of files are downloaded.
- **Increased reach and simplified monetization**. Maybe not your case but 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 have to 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 has to run on the .NET Framework 4.6.2 or later and writes to the `HKEY_CURRENT_USER` registry hive and the AppData folders will be virtualized to a user-specific app-local 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 goal by placing the application inside a UWP package. It detects and redirects some changes to the file system and registry at run time 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 that aren't sandboxed, although there's lightweight virtualization applied to the app for writes to `HKCU` and `AppData`. This virtualization allows them to interact with other apps the same way classic desktop applications do.

Installation

App packages are installed under *%ProgramFiles%\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 doesn't 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's 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.

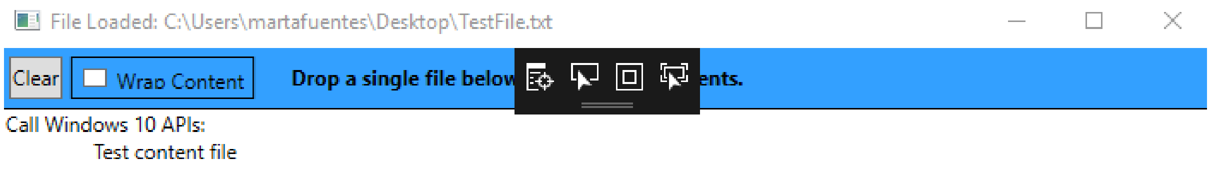
For details about how a packaged application handles installation, file access, registry, and uninstallation, see https://docs.microsoft.com/windows/msix/desktop/desktop-to-uwp-behind-the-scenes.

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 on how to integrate Toast Notifications in an existing WPF application. Although it's 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 don't require Package Identity.

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



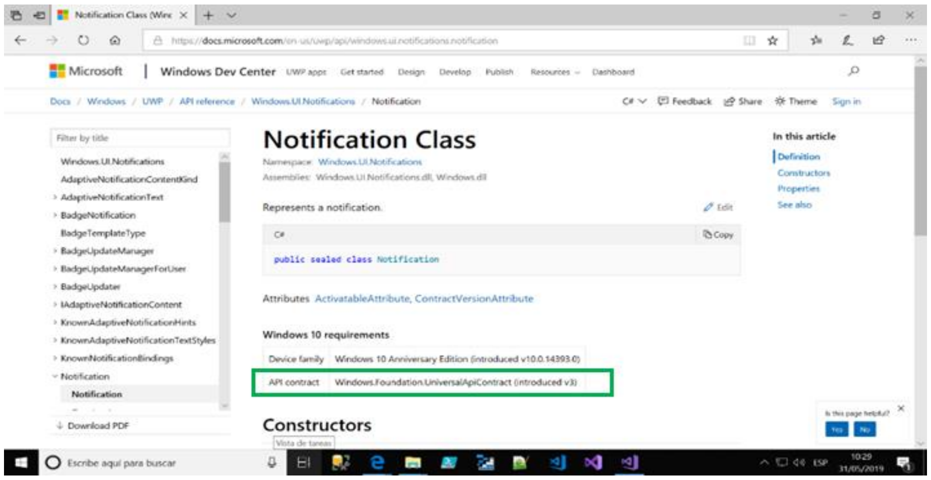
ents.

Call Windows 10 APIs:
Test content file

First, you should check in the following link whether the Windows 10 API that you'll 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.Notification API that requires a packaged identity:



Notification Class

Namespace: Windows.UI.Notifications
Assemblies: Windows.UI.Notifications.dll, Windows.dll

Represents a notification.

```
public sealed class Notification
```

Attributes ActivatableAttribute, ContractVersionAttribute

Windows 10 requirements

Device family	Windows 10 Anniversary Edition (introduced v10.0.14393.0)
API contract	Windows.Foundation.UniversalApiContract (introduced v3)

Constructors

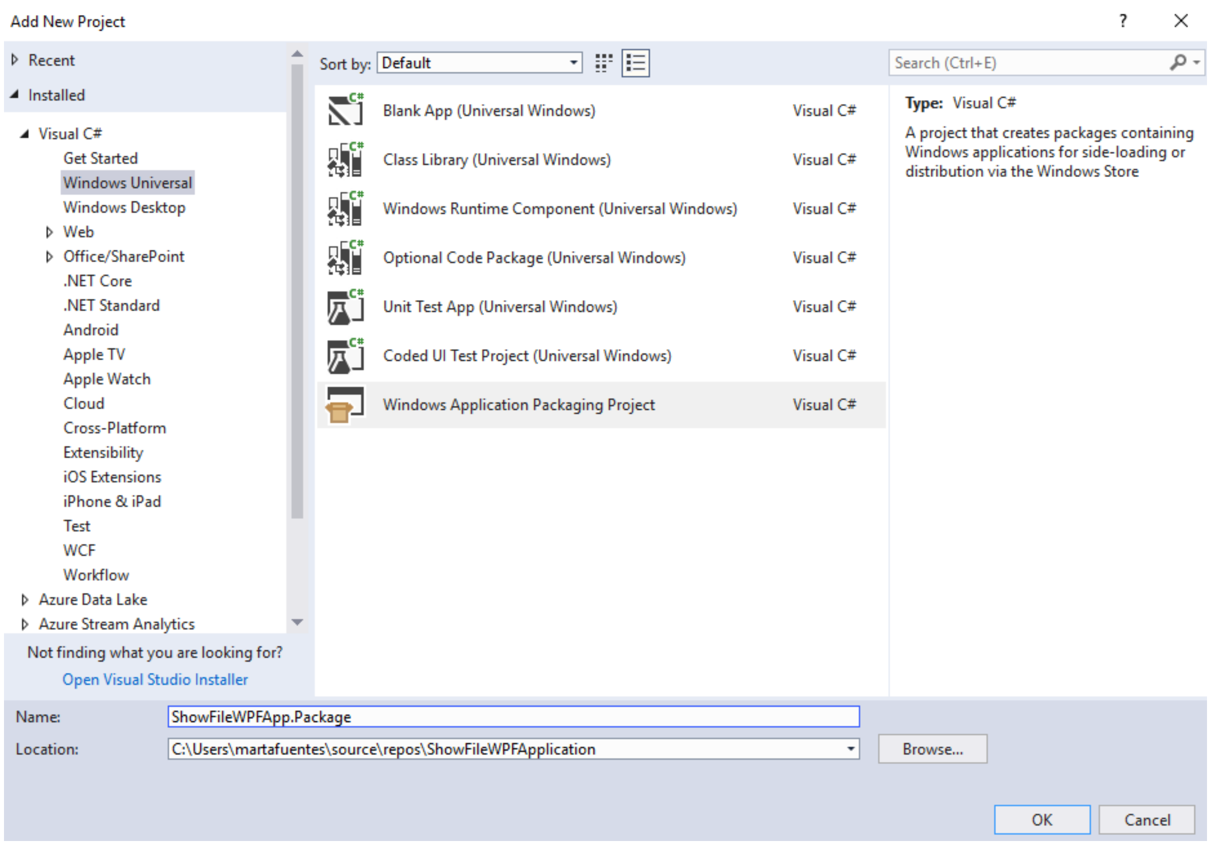
To access the WinRT API, add a reference to the `Microsoft.Windows.SDK.Contracts` NuGet package and this package will do the magic behind the scenes (see details at https://blogs.windows.com/windowsdeveloper/2019/04/30/calling-windows-10-apis-from-a-desktop-application-just-got-easier/).

You're now prepared to start adding some code.

Create a `ShowToastNotification` method that will be called on application startup. It just builds a toast notification from an XML pattern:

```
private void ShowNotification(string title, string content, string image)
{
    string xmlString = $@"<toast><visual><binding template =
'ToastGeneric'><text>{title}</text><text>{content}</text><image
src=>'{image}'</image></binding></visual></toast>";
    XmlDocument toastXml = new XmlDocument();
    toastXml.LoadXml(xmlString);
    ToastNotification toast = new ToastNotification(toastXml);
    ToastNotificationManager.CreateToastNotifier().Show(toast);
}
```

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



Add New Project

- Recent
- Installed
  - Visual C#
    - Get Started
    - Windows Universal
    - Windows Desktop
    - Web
    - Office/SharePoint
    - .NET Core
    - .NET Standard
    - Android
    - Apple TV
    - Apple Watch
    - Cloud
    - Cross-Platform
    - Extensibility
    - iOS Extensions
    - iPhone & iPad
    - Test
    - WCF
    - Workflow
  - Azure Data Lake
  - Azure Stream Analytics

Not finding what you are looking for?
Open Visual Studio Installer

Sort by: Default

Blank App (Universal Windows)	Visual C#
Class Library (Universal Windows)	Visual C#
Windows Runtime Component (Universal Windows)	Visual C#
Optional Code Package (Universal Windows)	Visual C#
Unit Test App (Universal Windows)	Visual C#
Coded UI Test Project (Universal Windows)	Visual C#
Windows Application Packaging Project	Visual C#

Search (Ctrl+E)

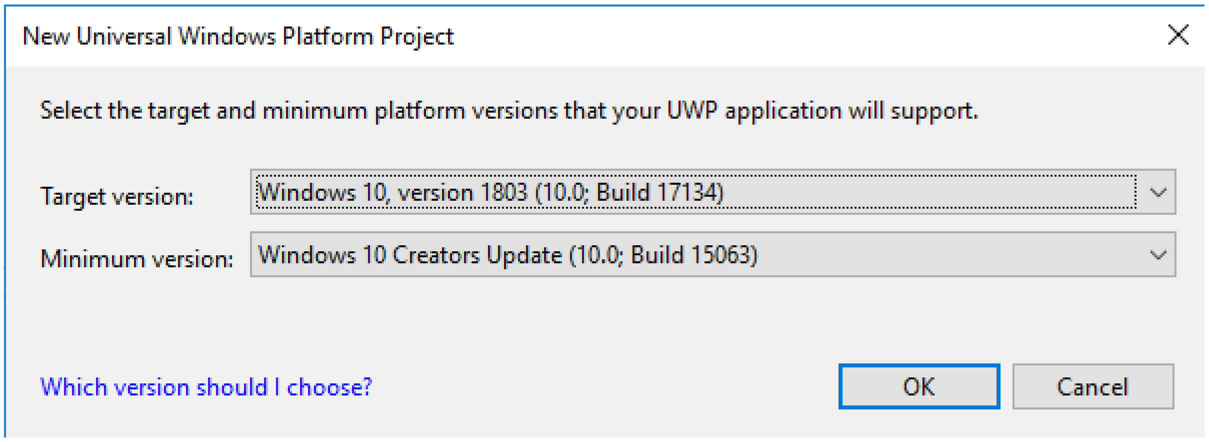
Type: Visual C#

A project that creates packages containing Windows applications for side-loading or distribution via the Windows Store

Name: ShowFileWPFApp.Package
Location: C:\Users\martafuentes\source\repos\ShowFileWPFApplication

Browse... OK Cancel

Select the minimum Windows version you want to support and the version you're 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 isn't available.



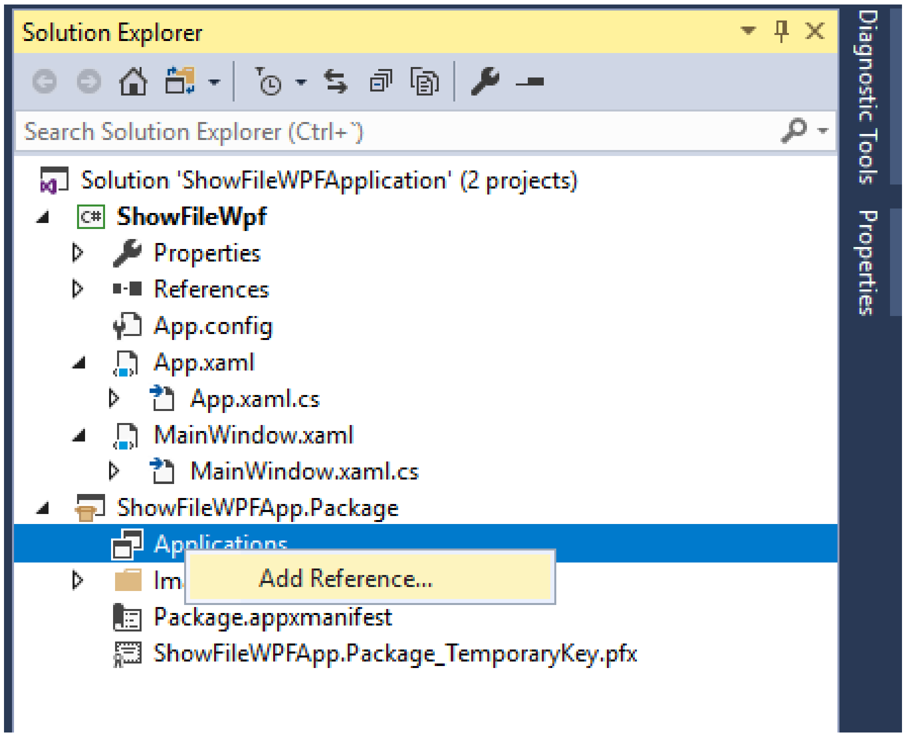
New Universal Windows Platform Project

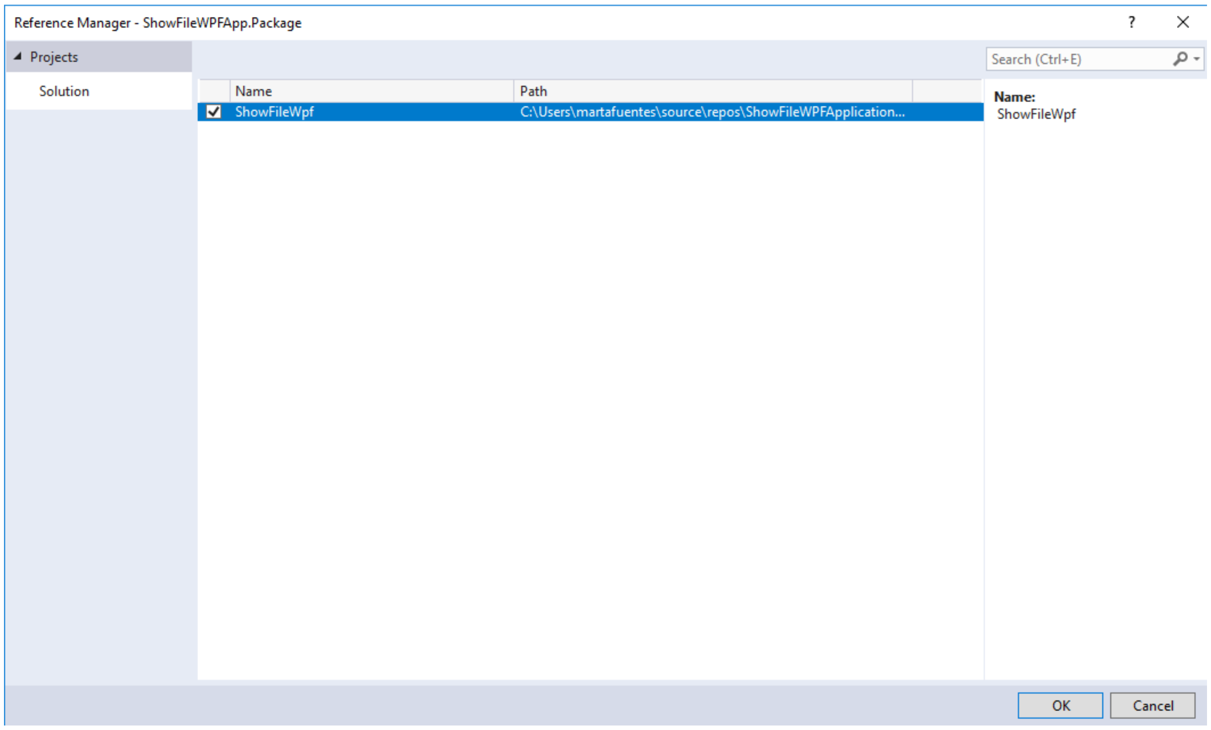
Select the target and minimum platform versions that your UWP application will support.

Target version: Windows 10, version 1803 (10.0; Build 17134)
Minimum version: Windows 10 Creators Update (10.0; Build 15063)

Which version should I choose? OK Cancel

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

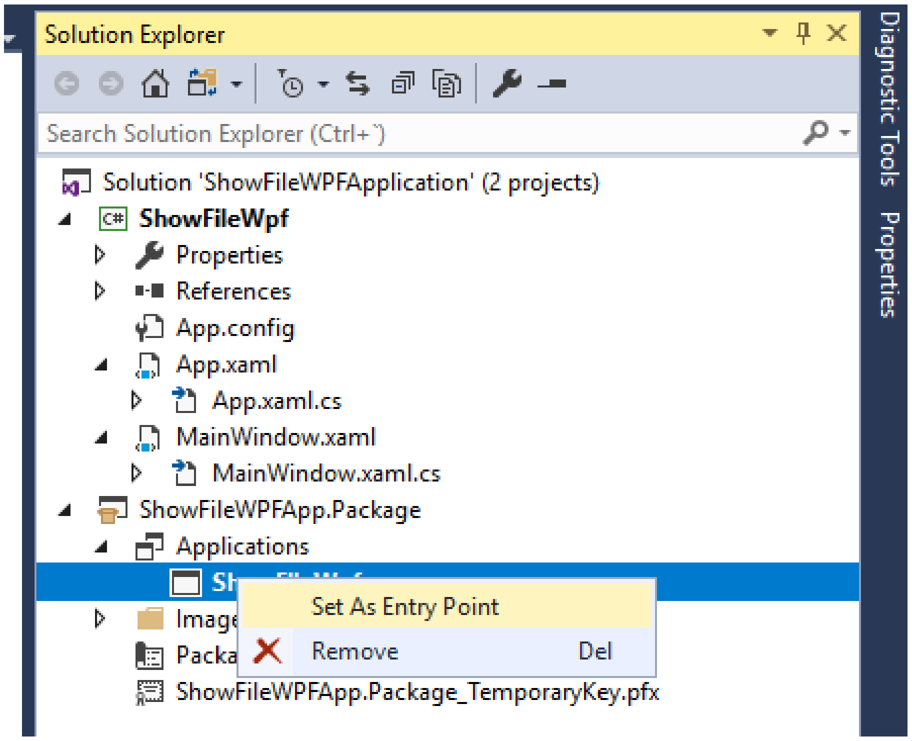




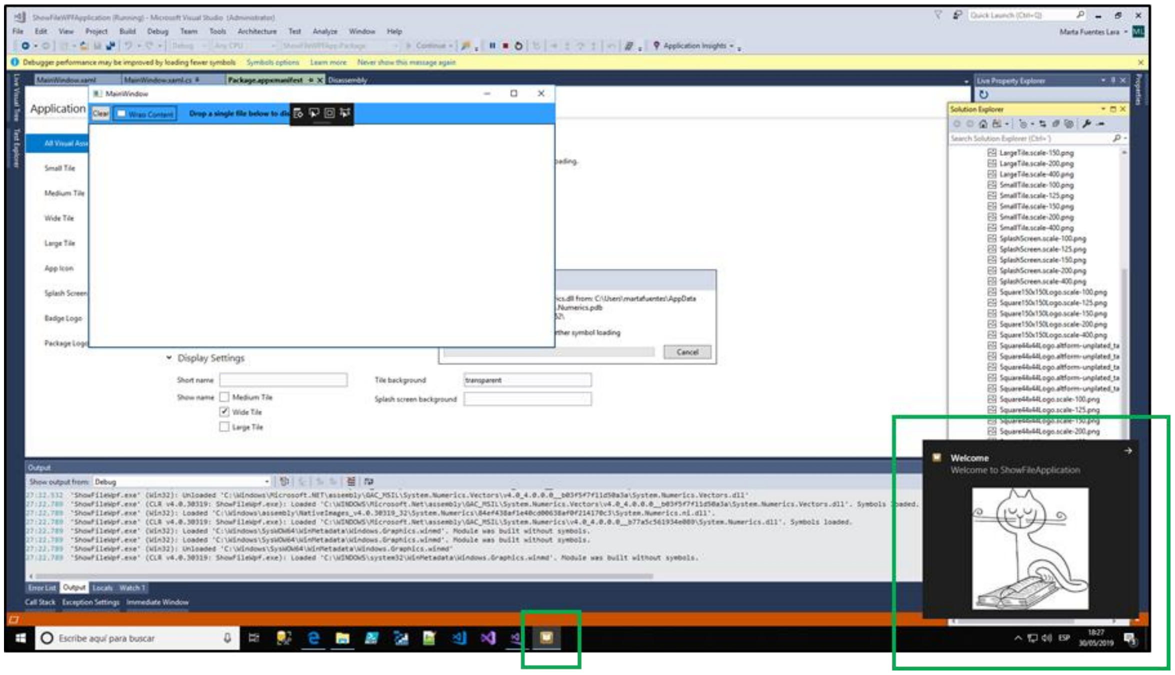
	Name	Path
✓	ShowFileWpf	C:\Users\martafuentes\source\repos\ShowFileWPFApplication...

Name:
ShowFileWpf

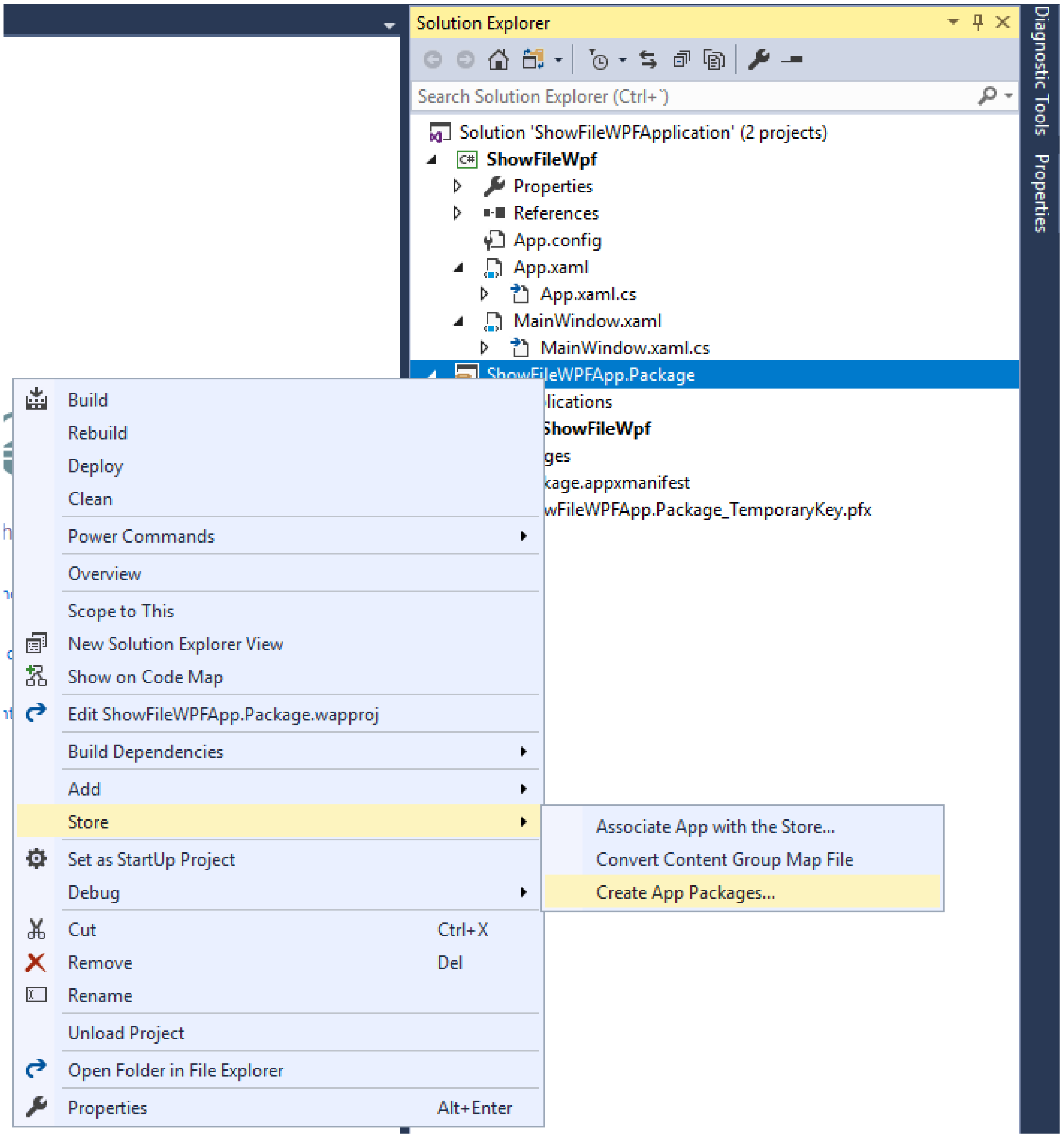
A Windows Packaging Project can package several apps so you should set which one is the Entry Point:



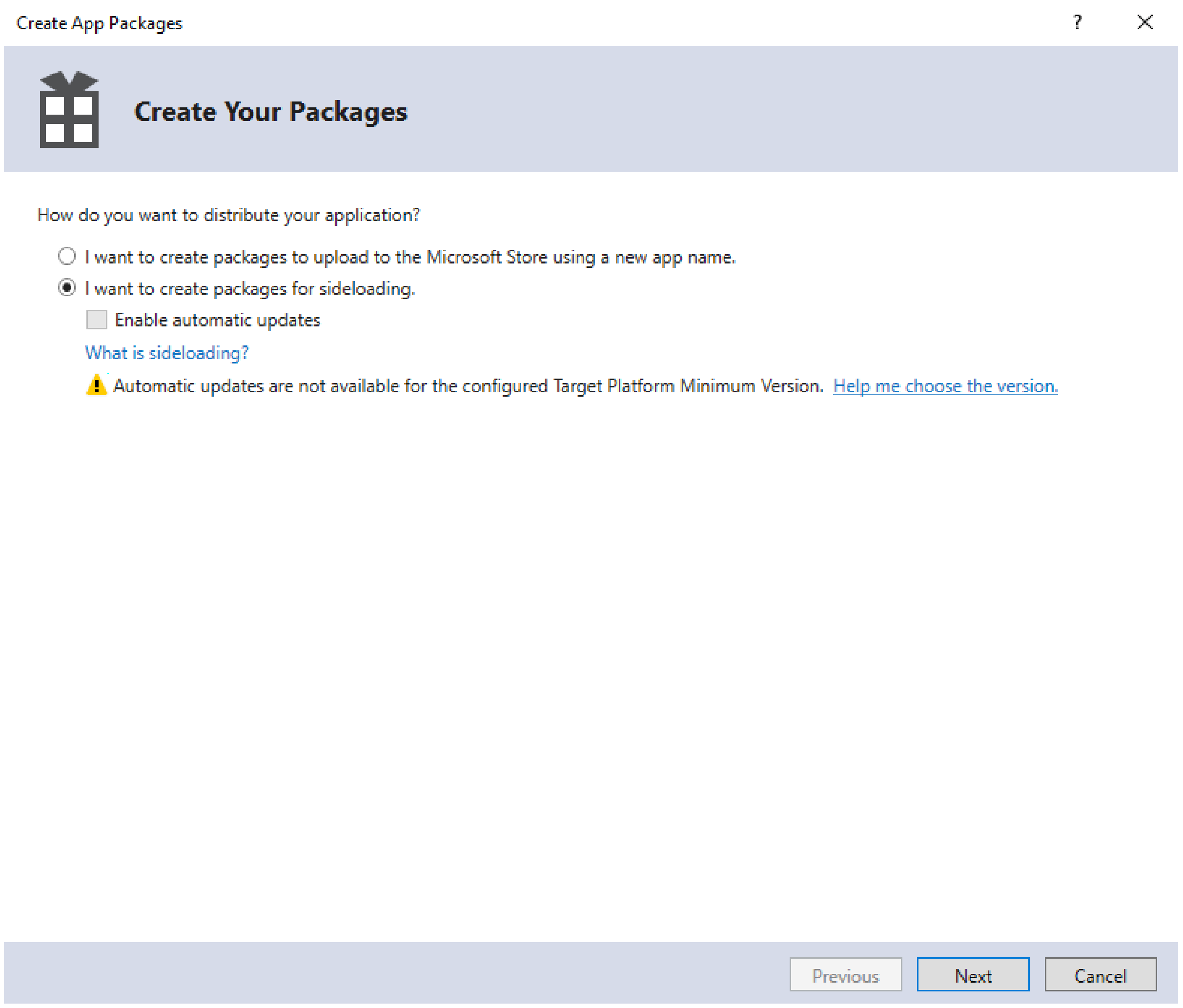
Next step is to set the WPF Project as the startup Project in the solution configuration. You can press F5 to compile and build and see the results.



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:



Create App Packages

Create Your Packages

How do you want to distribute your application?

( ) I want to create packages to upload to the Microsoft Store using a new app name.

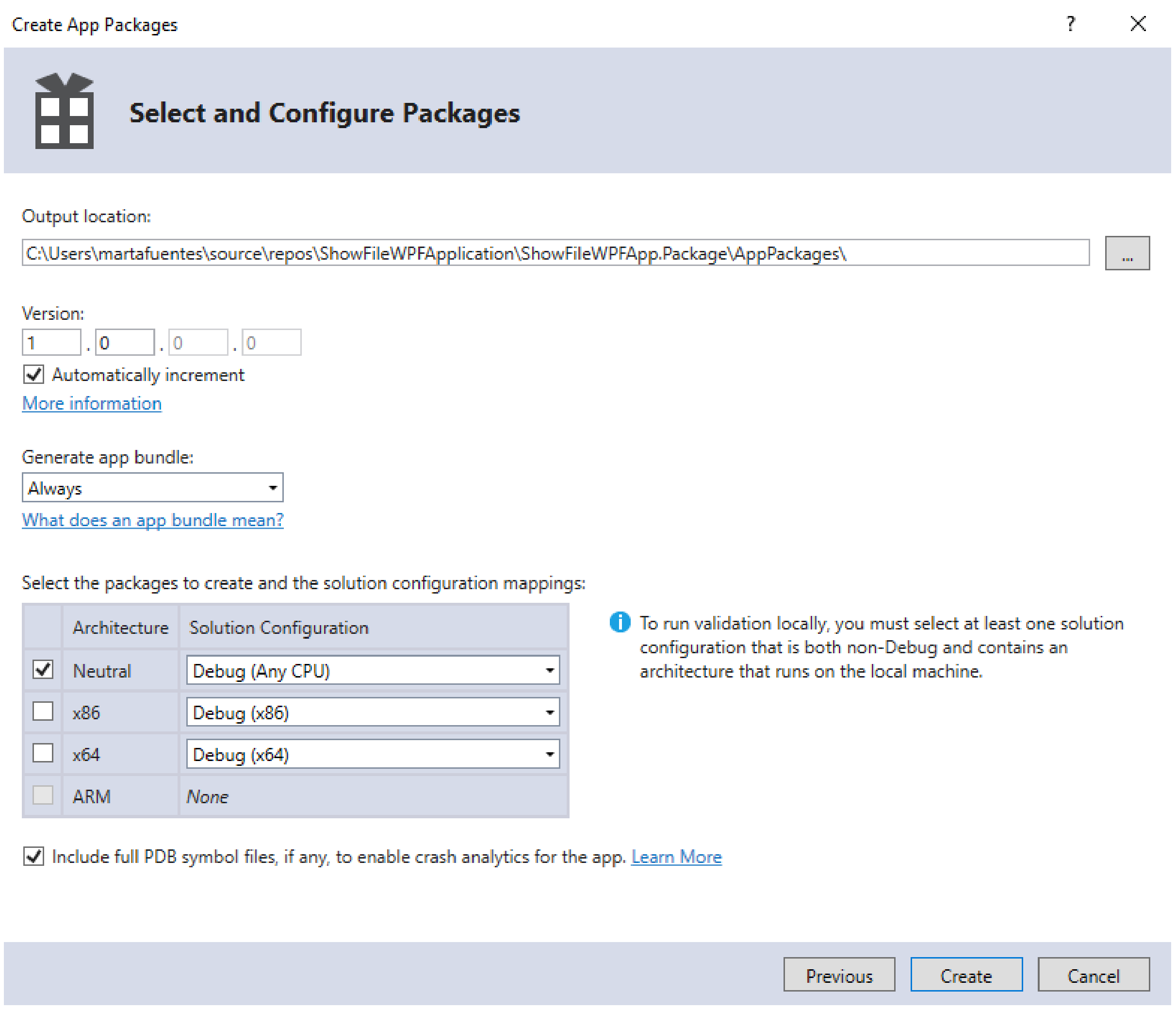
(●) I want to create packages for sideloading.

☐ Enable automatic updates

What is sideloading?

⚠ Automatic updates are not available for the configured Target Platform Minimum Version. Help me choose the version.

Select the application architecture of your app:


Create App Packages

Select and Configure Packages

Output location:

C:\Users\martafuentes\source\repos\ShowFileWPFApplication\ShowFileWPFApp.Package\AppPackages\

Version:

1 . 0 . 0 . 0

☑ Automatically increment

More information

Generate app bundle:

Always

What does an app bundle mean?

Select the packages to create and the solution configuration mappings:

	Architecture	Solution Configuration
☑	Neutral	Debug (Any CPU)
☐	x86	Debug (x86)
☐	x64	Debug (x64)
☐	ARM	*None*

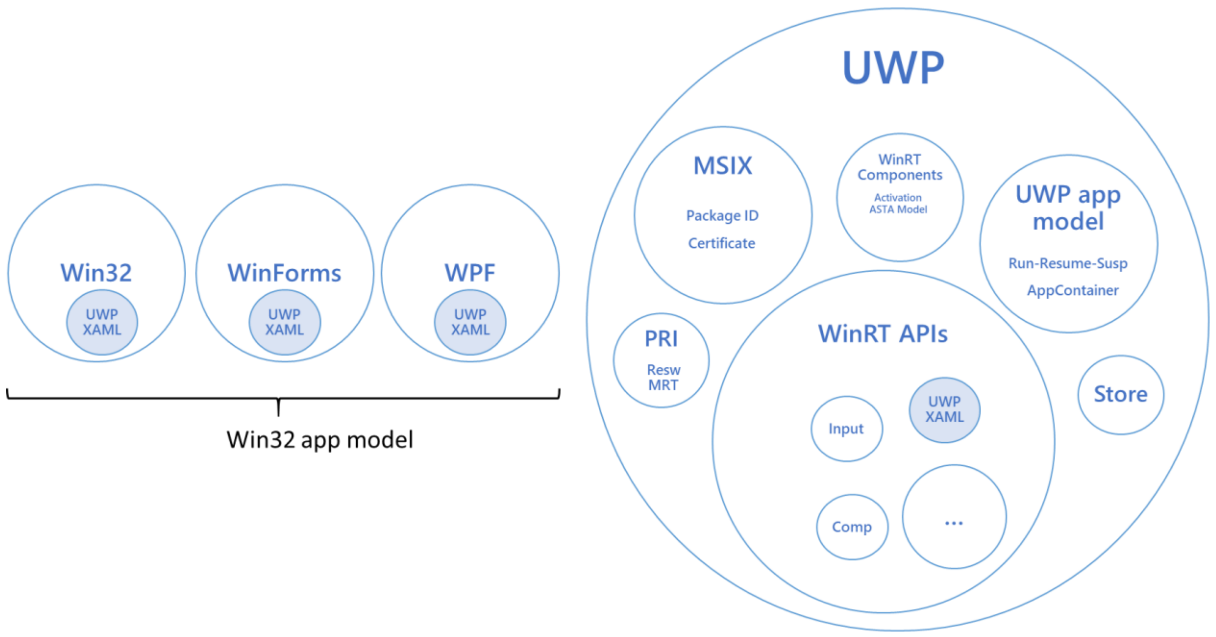
To run validation locally, you must select at least one solution configuration that is both non-Debug and contains an architecture that runs on the local machine.

☑ Include full PDB symbol files, if any, to enable crash analytics for the app. Learn More

Finally, create the package by 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 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. 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 `InitializeForCurrentThread` method 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're 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're 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 wrap some UWP controls into Windows Forms or WPF controls, hiding UWP concepts for those developers. 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

The power of XAML Islands extends to most first-party controls, third-party controls, and custom controls developed for UWP, which can be integrated into Windows Forms and WPF as “Islands” with fully functional UI. The `WindowsXamlHost` control for WPF and Windows Forms allows doing this.

For example, to use the `WindowsXamlHost` control in WPF, add a reference to the `Microsoft.Toolkit.Wpf.UI.XamlHost` package provided by the Windows Community Toolkit.

Once you’ve placed your `WindowsXamlHost` into your UI code, specify which UWP type you want to load. You can choose to use a wrapped control like a `Button` or a more complex one composed by several different controls, which are a custom UWP control.

The following example 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" />
```

There’s a clear recommendation on how to approach this and it’s better to have one single and bigger XAML Island containing a custom composite control than to have several islands with one control on each.

One of the core features of XAML is binding and it works between your Win32 code and the island. So, you can bind, for instance, 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.

To see a walkthrough about how to use XAML Islands, see:

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 third 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` UWP control in your .NET Core 3.x app.
- To 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 an independent UWP Class Library project that you reference in the same solution as your WPF or Windows Forms project.

You can check a detailed step-by-step process description at:

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

The Windows UI Library (WinUI 2)

Besides the inbox Windows 10 controls that comes with the OS, the same UWP XAML team also deliver additional controls 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

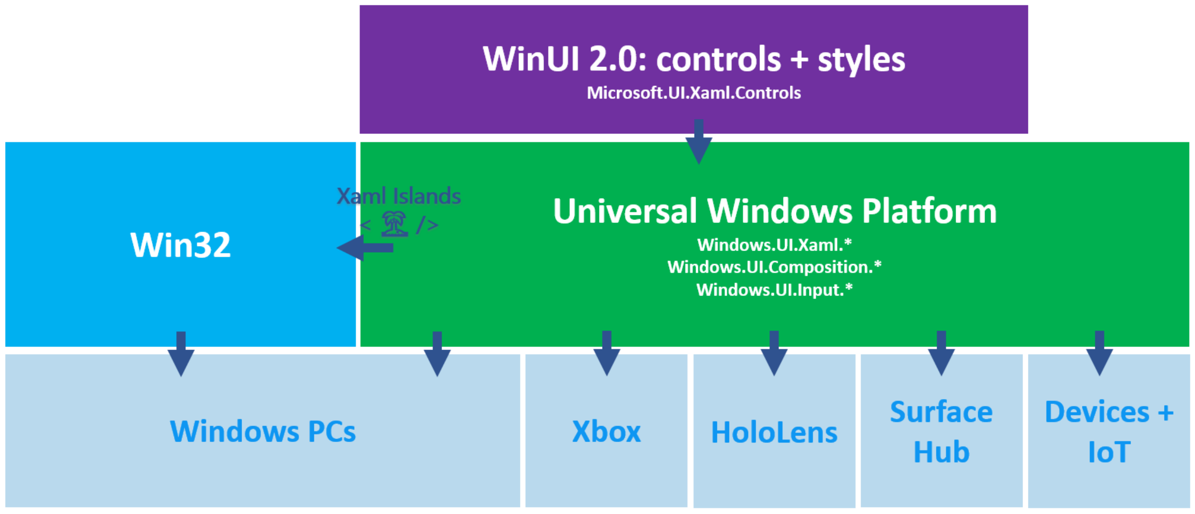
XAML Islands are intended for existing Win32 apps that want to improve their user experience by leveraging new UWP controls and behaviors without a full rewrite of the app. You could already leverage Windows 10 APIs, but up until XAML Islands, only non-UI related APIs.

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

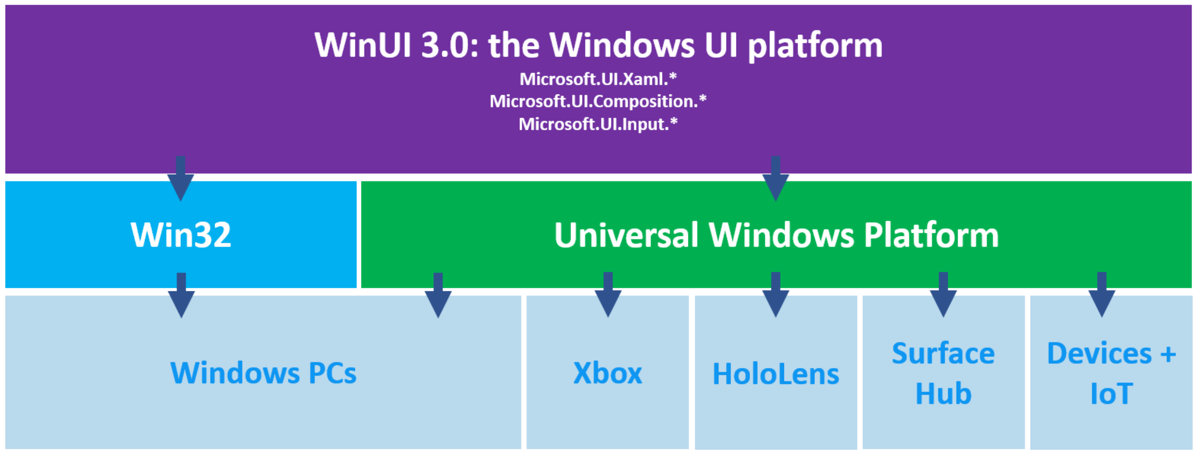
The road ahead XAML Islands: WinUI 3.0

Since Windows 8, the Windows UI platform, including the XAML UI framework, visual composition layer, and input processing has 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 the pace of innovation when you develop 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 be fully decoupled from the UWP SDK:



XAML framework will now be developed on GitHub and shipped out of band as NuGet packages.

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 (for example, 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 ship as part of WinUI instead.

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, and so on. To use just WinUI 3 in a Win32 app model, the WinUI content should be hosted by another UI Framework (Windows Forms, WPF, and so on) 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 for 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 allow that Win32 apps can use WinUI 3 as standalone UI Framework; no need to load Windows Forms or WPF.

Within this aggregation, WinUI 3 will let developers easily mix and match the right combination of:

- App model: UWP, Win32
- Platform: .NET Core 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.

If you want to know more details, Microsoft is sharing this roadmap in https://github.com/microsoft/microsoft-ui-xaml/blob/master/docs/roadmap.md.

Example of migrating to .NET Core 3.1

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 from WinForms and WPF versions.

Migration process overview

The migration process consists of four sequential steps:

1. **Preparation**: Understand the dependencies the project has to have an idea of what's 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 project format. Create a new project file with this format or update the one you have to use the SDK style.
3. **Fix code and build:** Build the code in .NET Core addressing API-level differences between .NET Framework and .NET Core. If needed, update third-party packages to the ones that support .NET Core.
4. **Run and test:** There might be differences that don't show up until run time. So, don't forget to run the application and test that everything works as expected.

Preparation

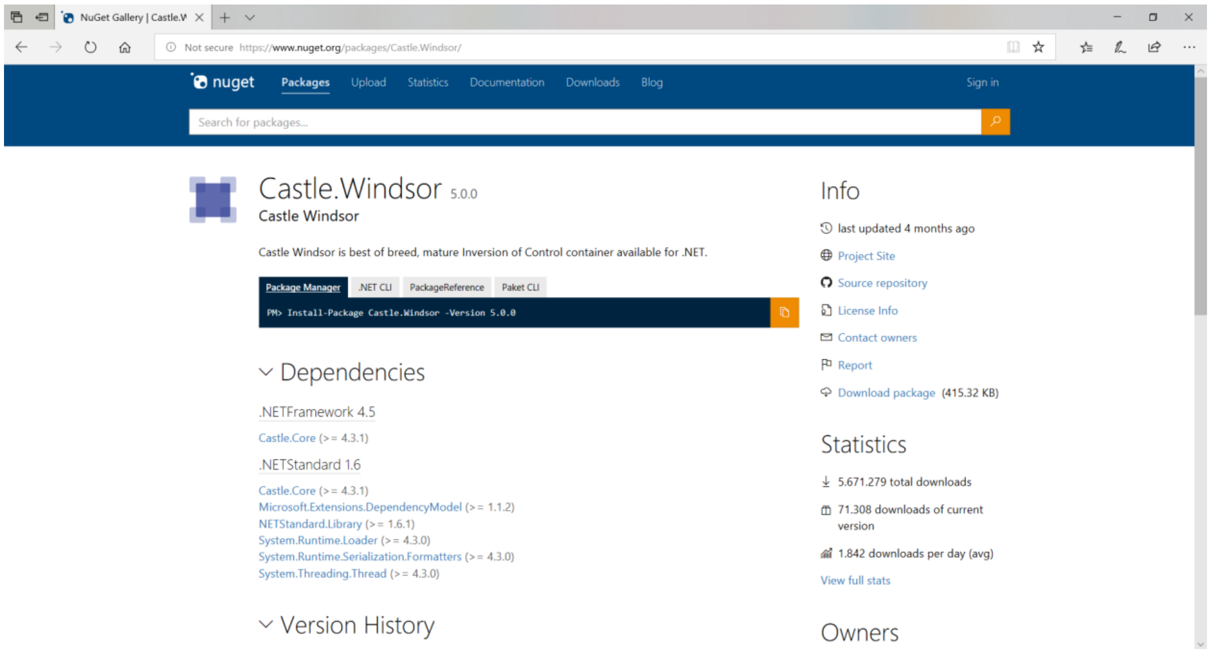
Migrate packages.config file

In a .NET Framework application, all references to external packages are declared in the *packages.config* file. In .NET Core, there's no longer the need to use the *packages.config* file. Instead, use the PackageReference property inside the project file to specify the NuGet packages for your app.

So, you need to transition from one format to another. You can do the update manually, taking the dependencies contained in the *packages.config* file and migrating them to the project file with the `PackageReference` format. Or, you can let Visual Studio do the work for you: right-click on the *packages.config* file and select the **Migrate packages.config to PackageReference** option.

Verify every dependency compatibility in .NET Core

Once you've migrated the package references, you must check each reference for compatibility. You can explore the dependencies of each NuGet package your application is using on nuget.org. If the package has .NET Standard dependencies, then it's going to work on .NET Core because .NET Core 3.1 supports all versions of .NET Standard. The following image shows the dependencies for the `Castle.Windsor` package:



Castle.Windsor 5.0.0
Castle Windsor

Castle Windsor is best of breed, mature Inversion of Control container available for .NET.

Package Manager | .NET CLI | PackageReference | Paket CLI

```
PM> Install-Package Castle.Windsor -Version 5.0.0
```

Dependencies

.NETFramework 4.5

Castle.Core (>= 4.3.1)

.NETStandard 1.6

Castle.Core (>= 4.3.1)
Microsoft.Extensions.DependencyModel (>= 1.1.2)
NETStandard.Library (>= 1.6.1)
System.Runtime.Loader (>= 4.3.0)
System.Runtime.Serialization.Formatters (>= 4.3.0)
System.Threading.Thread (>= 4.3.0)

Version History

Info

last updated 4 months ago
Project Site
Source repository
License Info
Contact owners
Report
Download package (415.32 KB)

Statistics

5.671.279 total downloads
71.308 downloads of current version
1.842 downloads per day (avg)
View full stats

Owners

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

Maybe the project is referencing older package versions that don't support .NET Core, but you might find newer versions that do support it. So, updating packages to newer versions is generally a good recommendation. However, you should consider that updating the package version can introduce some breaking changes that would force you to update your code.

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's possible to depend on .NET Framework packages from a .NET Core application. Don't forget to test it extensively because it can cause run-time errors if the external package calls an API that isn't available on .NET Core. This is great for when you're using an old package that isn't going to be updated and you can just retarget to work on the .NET Core.

Check for API compatibility

Since the API surface in .NET Framework and .NET Core is similar but not identical, you must check which APIs are available on .NET Core and which aren't. You can use the .NET Portability Analyzer tool to surface APIs used that aren't present on .NET Core. It looks at the binary level of your app, extracts all the APIs that are called, and then lists which APIs aren't available on your target framework (.NET Core 3.1 in this case).

You can find more information about this tool at:

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

An interesting aspect of this tool is that it only surfaces the differences from your own code and not code from external packages, which you can't 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'll want to update your existing project to the new .NET Core format. However, 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 for 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. If you need to modify UI elements in the designer, you can switch to the old project to do that. And since assets are linked, they'll be updated in the .NET Core project as well.

The SDK-style project for .NET Core is a lot simpler than .NET Framework's project format. And apart from the previously mentioned `PackageReference` entries, you won't need to do much more. The new project format includes certain file extensions by default, such as `.cs` and `.xaml` files, without the need to explicitly include them in the project file.

Assembly.info considerations

Attributes are autogenerated on .NET Core projects. If the project contains an *AssemblyInfo.cs* file, the definitions will be duplicated, which will cause compilation conflicts. You can delete the older *AssemblyInfo.cs* file or disable autogeneration by adding the following entry to the .NET Core project file:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
  <PropertyGroup>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  </PropertyGroup>
</Project>
```

Resources

Embedded resources are included automatically but resources aren't, so you need to migrate the resources to the new project file.

Package references

With the **Migrate packages.config to PackageReference** option, you can easily move your external package references to the new format as previously mentioned.

Update package references

Update the versions of the packages you've found to be compatible, as shown in the previous section.

Fix the code and build

Microsoft.Windows.Compatibility

If your application depends on APIs that aren't available on .NET Core, such as Registry, ACLs, or 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's a tool called API Analyzer (https://docs.microsoft.com/dotnet/standard/analyzers/api-analyzer) that helps you identify APIs that aren't compatible with your code.

Use #if directives

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

Technologies not available on .NET Core

Some technologies aren't available on .NET Core, such as:

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

That's why you need to find a replacement for these technologies if you're using them in your application. For more information, see the .NET Framework technologies unavailable on .NET Core article.

Regenerate autogenerated clients

If your application uses autogenerated code, such as a WCF client, you may need to regenerate this code to target .NET Core. Sometimes, you can find some missing references since they may not be included as part of the default .NET Core assemblies set. Using a tool like 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 general rule, we've previously stated that you better update every single package version to be compatible with .NET Core. However, you can find that targeting an updated and compatible version of an assembly just doesn't pay off. If the cost of change isn't acceptable, you can consider rolling back package versions keeping the ones you use on .NET Framework. Although they may not be targeting .NET Core, they should work well unless they call some unsupported APIs.

Run and test

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

In this final step, you can find several different issues depending on the complexity of your application and the dependencies and APIs you're using.

For example, if you use configuration files (*app.config*), you may find some errors at run time like Configuration Sections not present. Using the `Microsoft.Extensions.Configuration` NuGet package should fix that error.

Another reason for errors is the use of the `BeginInvoke` and `EndInvoke` methods because they aren't supported on .NET Core. They aren't supported on .NET Core because they have a dependency on Remoting, which doesn't exist on .NET Core. To solve this issue, try to use the `await` keyword (when available) or the Task.Run method.

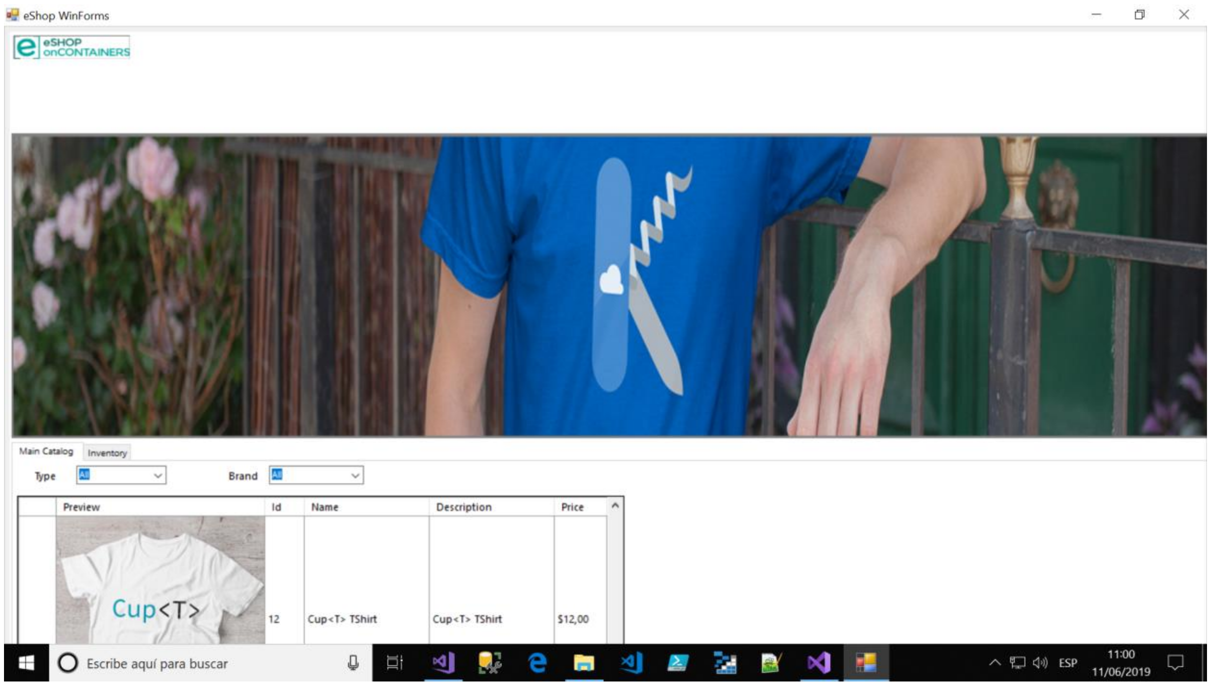
You can use compatibility analyzers to let you identify APIs and code patterns in your code that can potentially cause problems at run time with .NET Core. Go to http://github.com/dotnet/platform-compat and use the .NET API analyzer on your project.

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/tree/master/eShopLegacyNTier/src/eShopWinForms. You can find the complete result of the migration at https://github.com/dotnet-architecture/eShopModernizing/tree/master/eShopModernizedNTier/src/eShopWinForms.

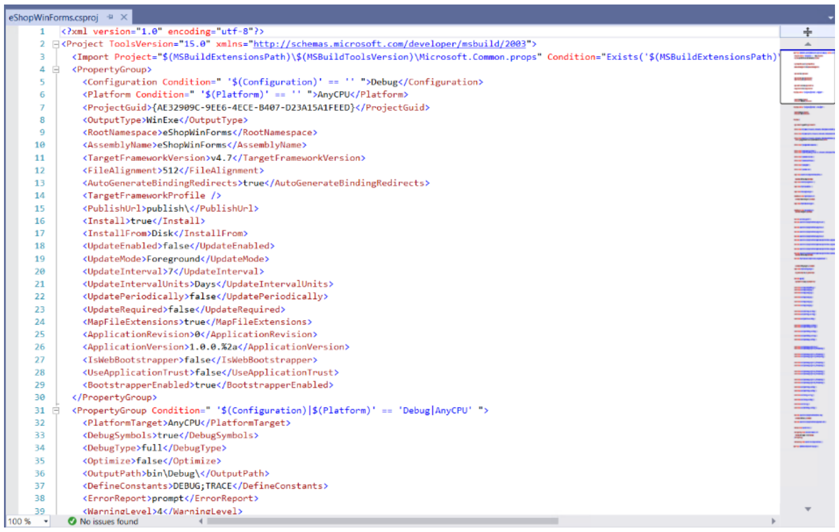
This application shows a product catalog and allows the user to navigate, filter, and search for products. From an architecture point of view, the app relies on an external WCF service that serves as a façade to a back-end database.

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



Preview	Id	Name	Description	Price
	12	Cup<T> TShirt	Cup<T> TShirt	$12,00

If you open the *.csproj* project file, you can see something like this:



```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" Condition="Exists('$(MSBuildExtensionsPath)
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProjectGuid>{AE32909C-9EE6-4ECE-B407-D23A15A1FEED}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <RootNamespace>eShopWinForms</RootNamespace>
    <AssemblyName>eShopWinForms</AssemblyName>
    <TargetFrameworkVersion>v4.7</TargetFrameworkVersion>
    <FileAlignment>512</FileAlignment>
    <AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
    <TargetFrameworkProfile />
    <PublishUrl>publish\</PublishUrl>
    <Install>true</Install>
    <InstallFrom>Disk</InstallFrom>
    <UpdateEnabled>false</UpdateEnabled>
    <UpdateMode>Foreground</UpdateMode>
    <UpdateInterval>7</UpdateInterval>
    <UpdateIntervalUnits>Days</UpdateIntervalUnits>
    <UpdatePeriodically>false</UpdatePeriodically>
    <UpdateRequired>false</UpdateRequired>
    <MapFileExtensions>true</MapFileExtensions>
    <ApplicationRevision>0</ApplicationRevision>
    <ApplicationVersion>1.0.0.%2a</ApplicationVersion>
    <IsWebBootstrapper>false</IsWebBootstrapper>
    <UseApplicationTrust>false</UseApplicationTrust>
    <BootstrapperEnabled>true</BootstrapperEnabled>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
```

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

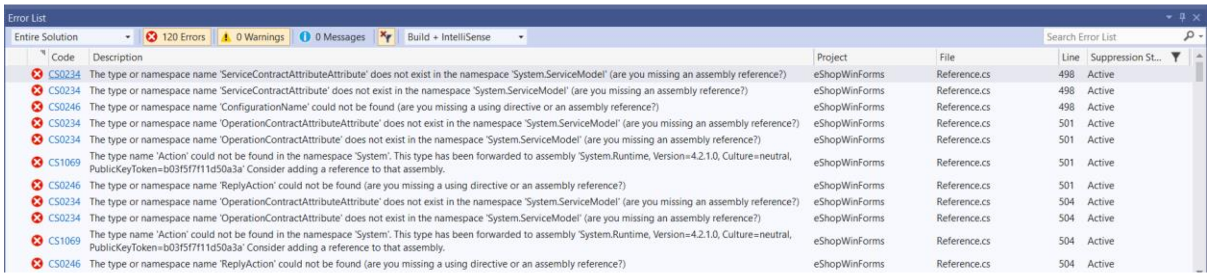
In the Solution Explorer, right click on the Windows Forms project and select **Unload Project** > **Edit**.

Now you can update the .csproj file. You'll delete the entire content and replace it with the following code:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
    <PropertyGroup>
        <OutputType>WinExe</OutputType>
        <TargetFramework>netcoreapp3.1</TargetFramework>
        <UseWindowsForms>true</UseWindowsForms>
        <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    </PropertyGroup>
</Project>
```

Save and reload the project. You're now done updating the project file and the project is targeting the .NET Core.

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



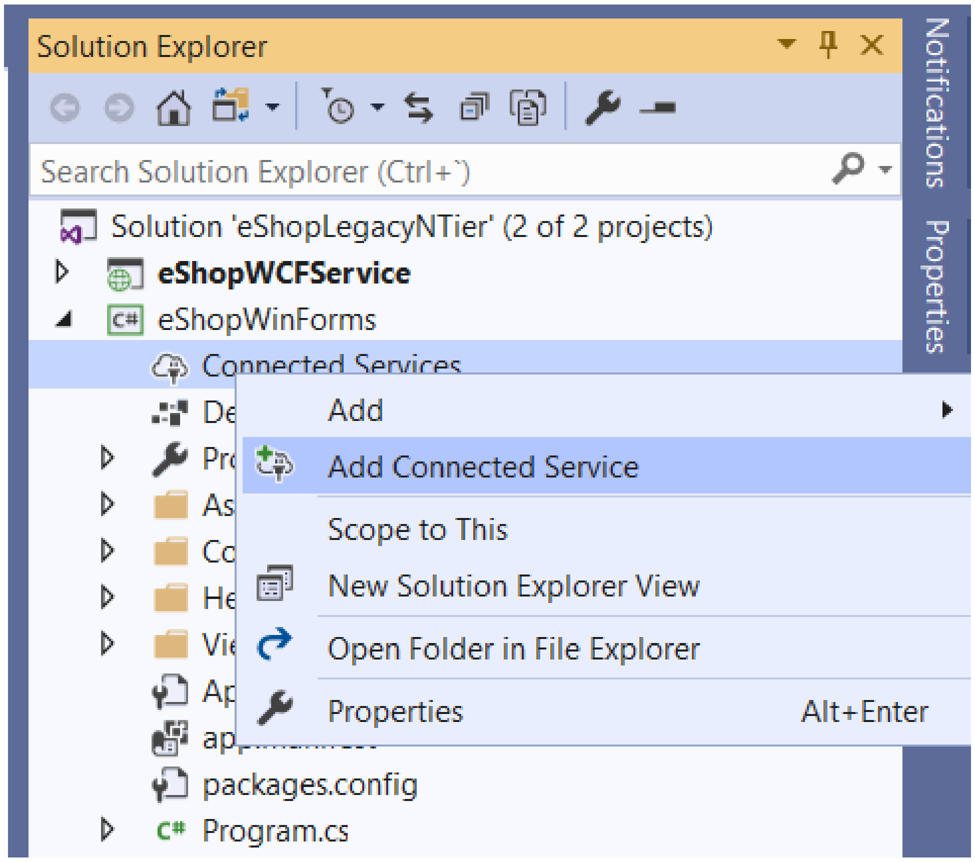
Error List

Entire Solution | 120 Errors | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

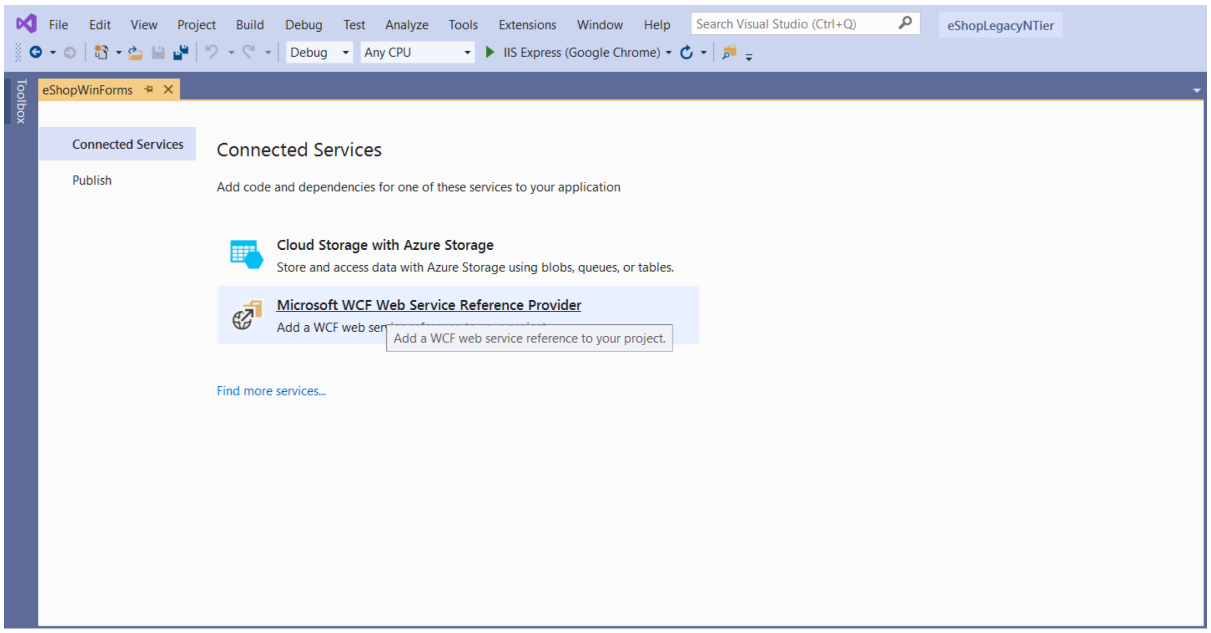
Code	Description	Project	File	Line	Suppression St...
CS0234	The type or namespace name 'ServiceContractAttributeAttribute' does not exist in the namespace 'System.ServiceModel' (are you missing an assembly reference?)	eShopWinForms	Reference.cs	498	Active
CS0234	The type or namespace name 'ServiceContractAttribute' does not exist in the namespace 'System.ServiceModel' (are you missing an assembly reference?)	eShopWinForms	Reference.cs	498	Active
CS0246	The type or namespace name 'ConfigurationName' could not be found (are you missing a using directive or an assembly reference?)	eShopWinForms	Reference.cs	498	Active
CS0234	The type or namespace name 'OperationContractAttributeAttribute' does not exist in the namespace 'System.ServiceModel' (are you missing an assembly reference?)	eShopWinForms	Reference.cs	501	Active
CS0234	The type or namespace name 'OperationContractAttribute' does not exist in the namespace 'System.ServiceModel' (are you missing an assembly reference?)	eShopWinForms	Reference.cs	501	Active
CS1069	The type name 'Action' could not be found in the namespace 'System'. This type has been forwarded to assembly 'System.Runtime, Version=4.2.1.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' Consider adding a reference to that assembly.	eShopWinForms	Reference.cs	501	Active
CS0246	The type or namespace name 'ReplyAction' could not be found (are you missing a using directive or an assembly reference?)	eShopWinForms	Reference.cs	501	Active
CS0234	The type or namespace name 'OperationContractAttributeAttribute' does not exist in the namespace 'System.ServiceModel' (are you missing an assembly reference?)	eShopWinForms	Reference.cs	504	Active
CS0234	The type or namespace name 'OperationContractAttribute' does not exist in the namespace 'System.ServiceModel' (are you missing an assembly reference?)	eShopWinForms	Reference.cs	504	Active
CS1069	The type name 'Action' could not be found in the namespace 'System'. This type has been forwarded to assembly 'System.Runtime, Version=4.2.1.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' Consider adding a reference to that assembly.	eShopWinForms	Reference.cs	504	Active
CS0246	The type or namespace name 'ReplyAction' could not be found (are you missing a using directive or an assembly reference?)	eShopWinForms	Reference.cs	504	Active

Delete the *Reference.cs* file and generate a new Service Client.

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



The Connected Services window opens. Select the **Microsoft WCF Web Service** option.



Connected Services

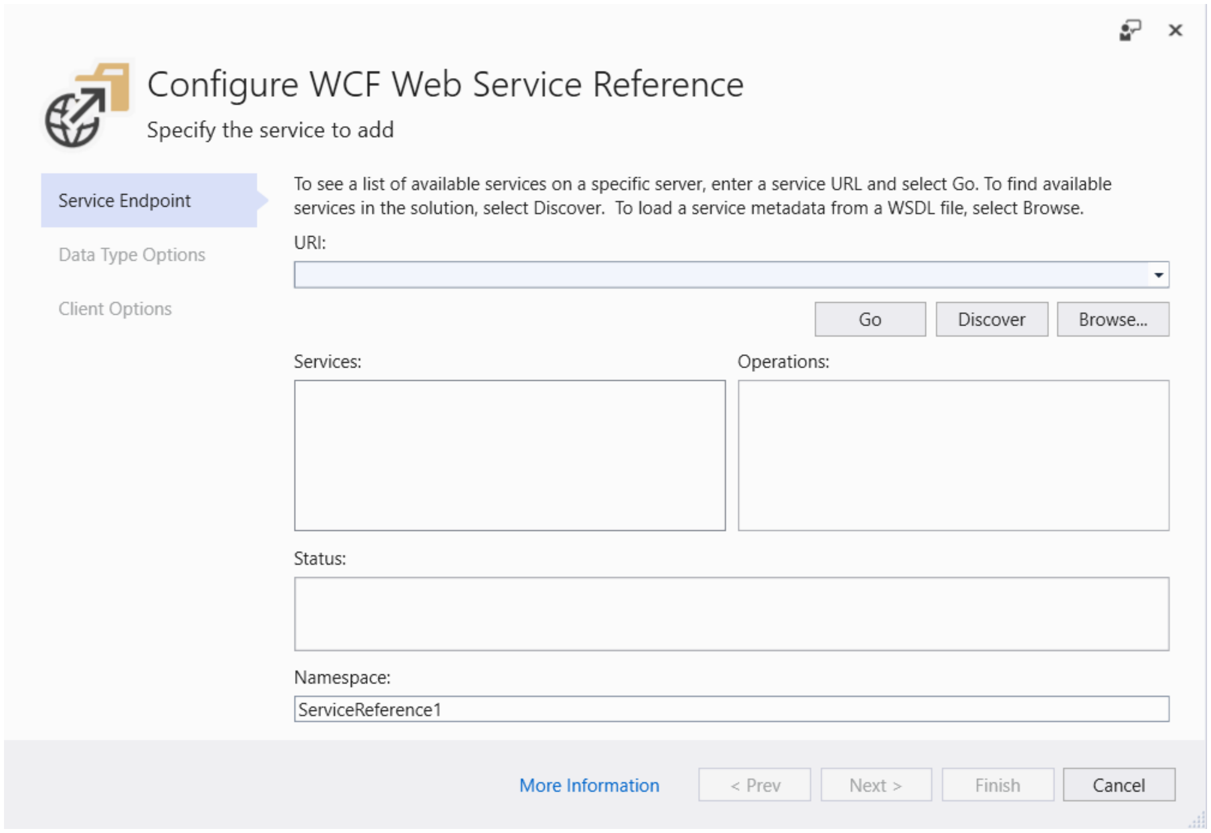
Add code and dependencies for one of these services to your application

Cloud Storage with Azure Storage
Store and access data with Azure Storage using blobs, queues, or tables.

Microsoft WCF Web Service Reference Provider
Add a WCF web ser
Add a WCF web service reference to your project.

Find more services...

If you have the WCF Service in the same solution as we have in this example, you can select the **Discover** option instead of specifying a service URL.


Configure WCF Web Service Reference

Specify the service to add

To see a list of available services on a specific server, enter a service URL and select Go. To find available services in the solution, select Discover. To load a service metadata from a WSDL file, select Browse.

URI:

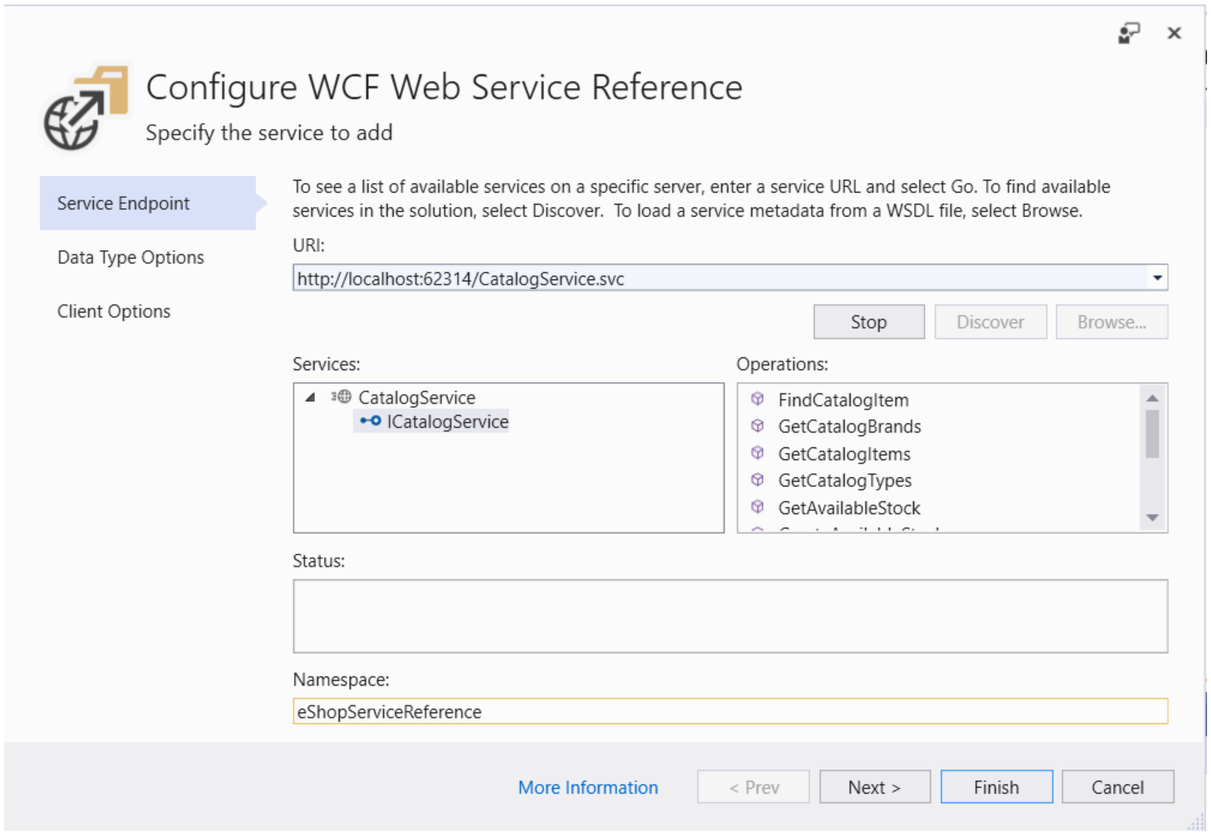
Services:

Operations:

Status:

Namespace: ServiceReference1

Once the service is located, the tool reflects the API contract implemented by the service. Change the name of the namespace to be `eShopServiceReference` as shown in the following image:



Configure WCF Web Service Reference

Specify the service to add

To see a list of available services on a specific server, enter a service URL and select Go. To find available services in the solution, select Discover. To load a service metadata from a WSDL file, select Browse.

URI: http://localhost:62314/CatalogService.svc

Services: CatalogService / ICatalogService

Operations: FindCatalogItem, GetCatalogBrands, GetCatalogItems, GetCatalogTypes, GetAvailableStock

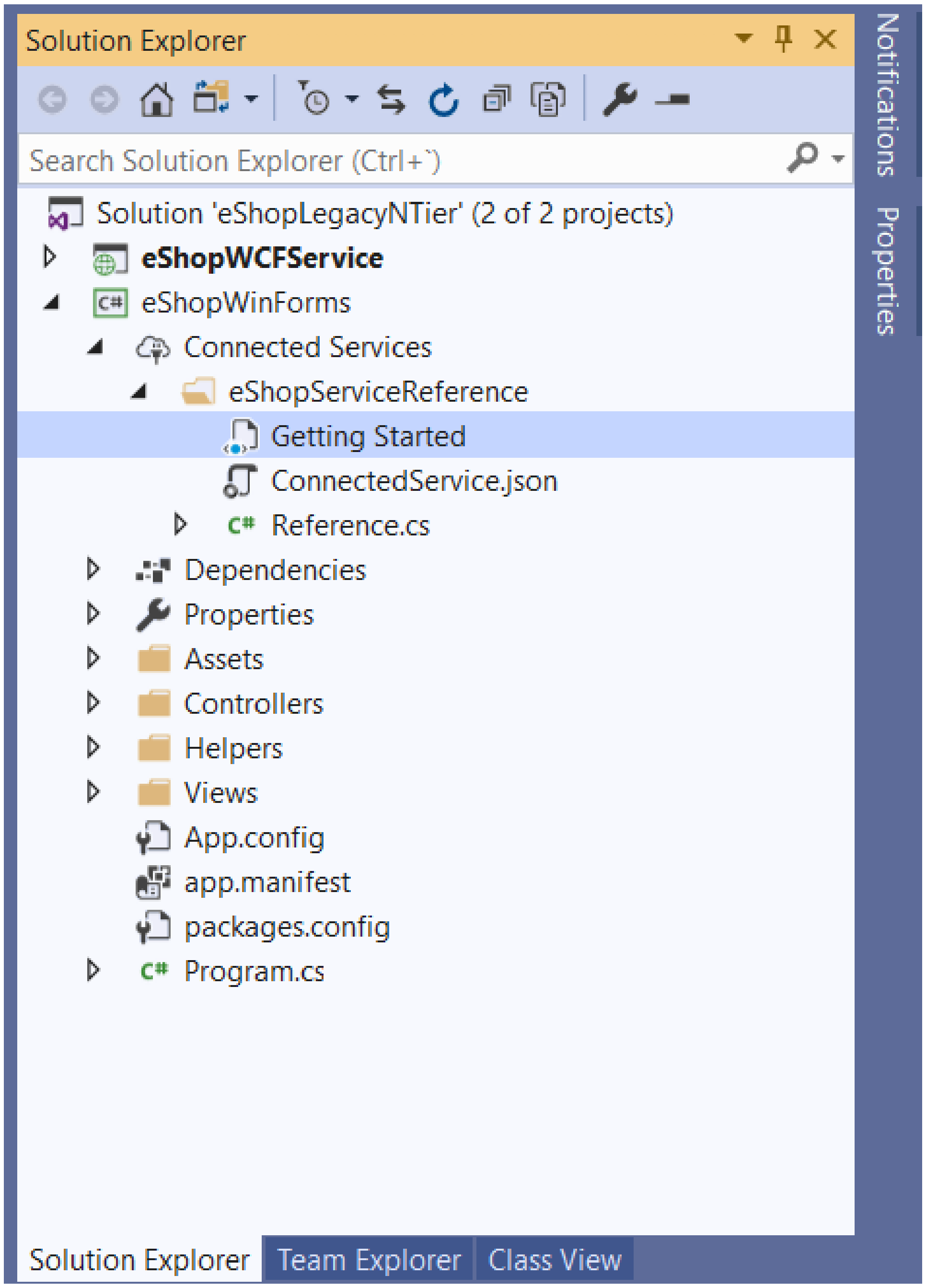
Status:

Namespace: eShopServiceReference

Select the **Finish** button. After a while, you'll see the generated code.

You should see three autogenerated files:

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



If you compile again, you'll 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-style project, every code file present underneath the project file location is included by default. That is, the new .NET Core project tries to compile the files inside the *Helper* folder. Since that folder isn't needed, you can safely delete it.

If you compile the project again and execute it, you won't see the product images. The problem is that now the path to the files has slightly changed. To fix this issue, you need to add another level of depth in the path, updating in the file `CatalogView.cs` the line:

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

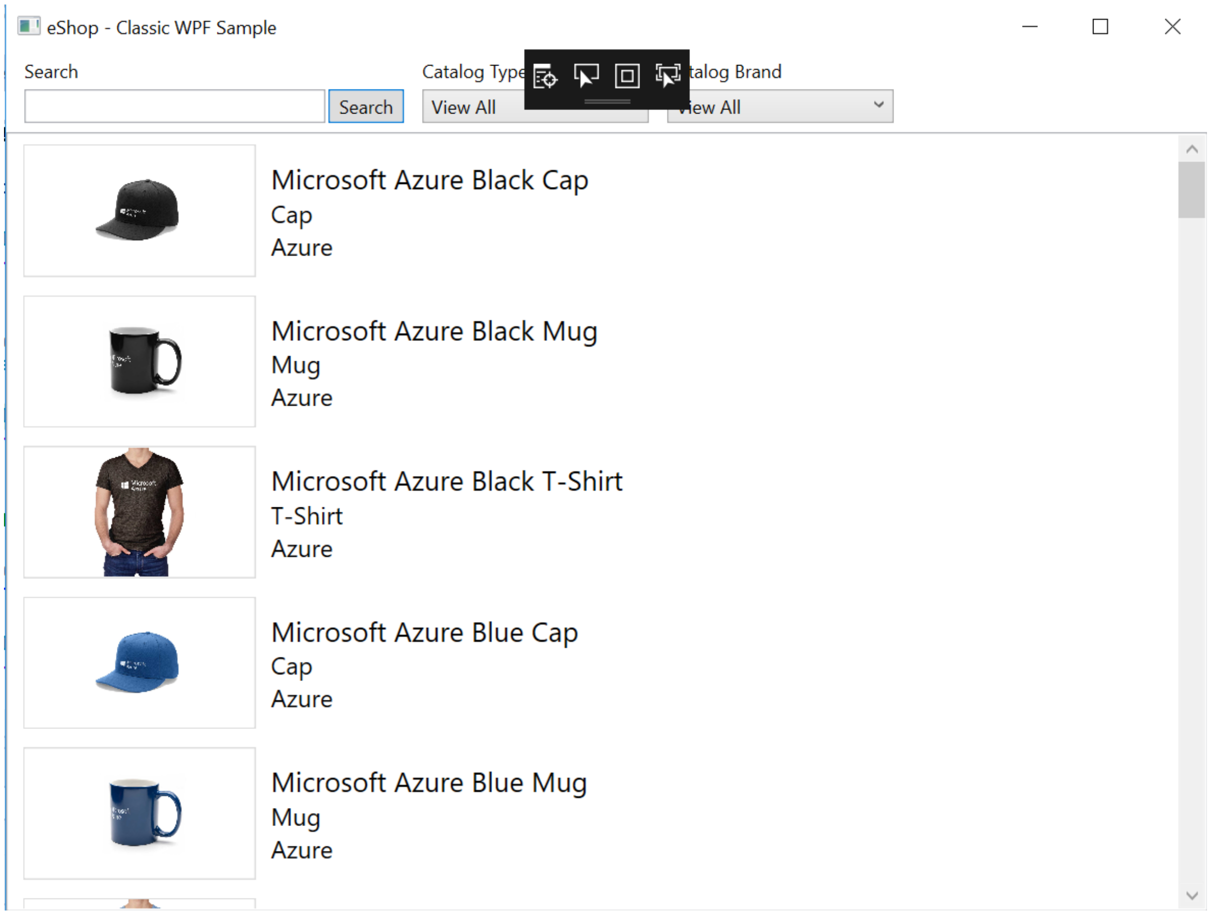
to

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

After this change, you can check that the application launches and runs as expected on .NET Core.

Migrating a WPF application

We'll use the `Shop.ClassicWPF` sample application to perform the migration. The following image shows a screenshot of the app before migration:



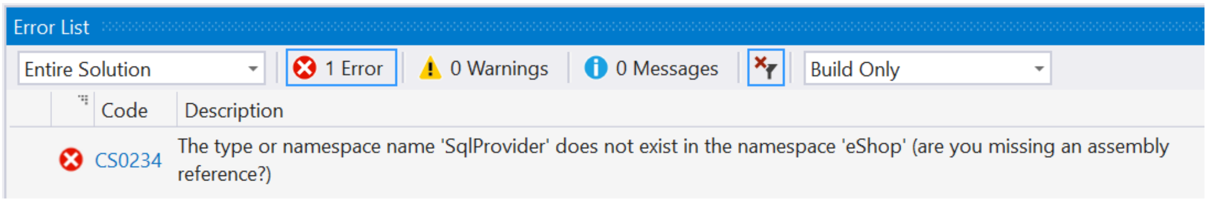
This application uses a local SQL Server Express database to hold the product catalog information. This database is accessed directly from the WPF application.

First, you must update the *.csproj* file to the new SDK style used by .NET Core applications. You'll follow the same steps described in the Windows Forms migration: you'll unload the project, open the *.csproj* file, update its contents, and reload the project.

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

```xml
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
    <PropertyGroup>
        <OutputType>WinExe</OutputType>
        <TargetFramework>netcoreapp3.1</TargetFramework>
        <UseWPF>true</UseWPF>
        <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    </PropertyGroup>
</Project>
```

If you reload the project and compile it, you'll get the following error:

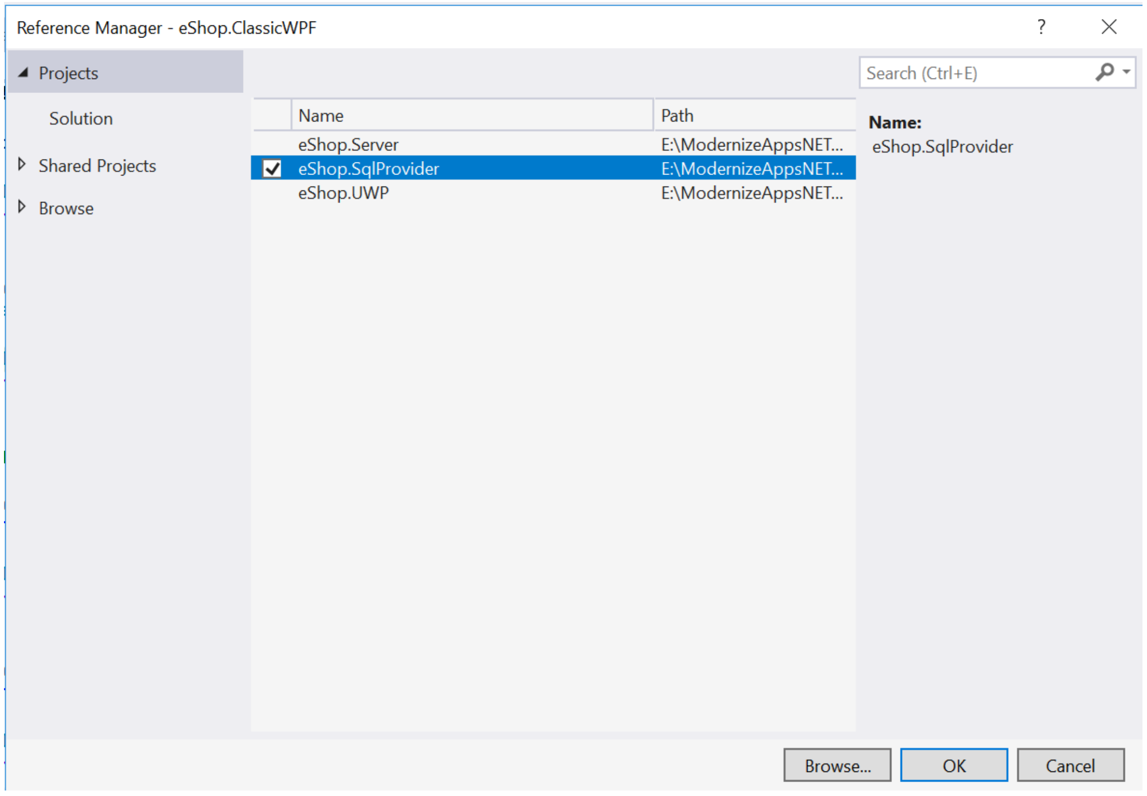


CS0234 The type or namespace name 'SqlProvider' does not exist in the namespace 'eShop' (are you missing an assembly reference?)

Since you've deleted all the *.csproj* contents, you've lost a project reference specification present in the old project. You just need to add this line to the *.csproj* file to include the project reference back:

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

You can also let Visual Studio help you by right-clicking on the **Dependencies** node and selecting **Add Project Reference**. Select the project from the solution and click **OK**:



	Name	Path
	eShop.Server	E:\ModernizeAppsNET...
✓	eShop.SqlProvider	E:\ModernizeAppsNET...
	eShop.UWP	E:\ModernizeAppsNET...

Name:
eShop.SqlProvider

Once you add the missing project reference, the application compiles and runs as expected on .NET Core.

CHAPTER 6

Deploying Modern Desktop Applications

When you develop desktop applications, one thing to consider is how your application is going to be packaged and deployed to the users' machines. The problem with packaging, deployment, and installation is that it usually falls under the umbrella of the IT professionals, who care about different things than developers.

These days, we're all familiar with the DevOps concept, where developers and IT Pros work closely to move applications to their production environments. But if you've been in the desktop battle for more than 10 years, you might have seen the following story. A team of developers works together hard to meet the project deadlines. Business people are nervous since they need the system working on many user's machines to run the company. On "D-Day", the project manager checks with every developer that their code is working well and that everything is fine, so they can ship. Then, the package team comes in generating the setup for the app, distribute it to every user machine and a set of test users run the application. Well, they try, because before showing any UI, the application throws an exception that says "Method ~ of object ~ failed". Panic starts flowing through the air and a brief investigation points to a young and tired developer that has introduced a third-party control, that certainly "worked on the dev machine".

Installing desktop applications have traditionally been a nightmare for two main reasons:

- Lack of close collaboration culture between dev and IT teams.
- Lack of a solid packaging and deploying technology we can build upon.

In fact, we've been living with the fact that sometimes you regret that you installed an app because:

- It ends up having some undesired side effects on your machine.
- Some applications that were previously installed stop working.

Additionally, you can't just restore the system to its original state by uninstalling the app. We're so used to live this situation that we've coined terms like "DLL Hell" or "Winrot".

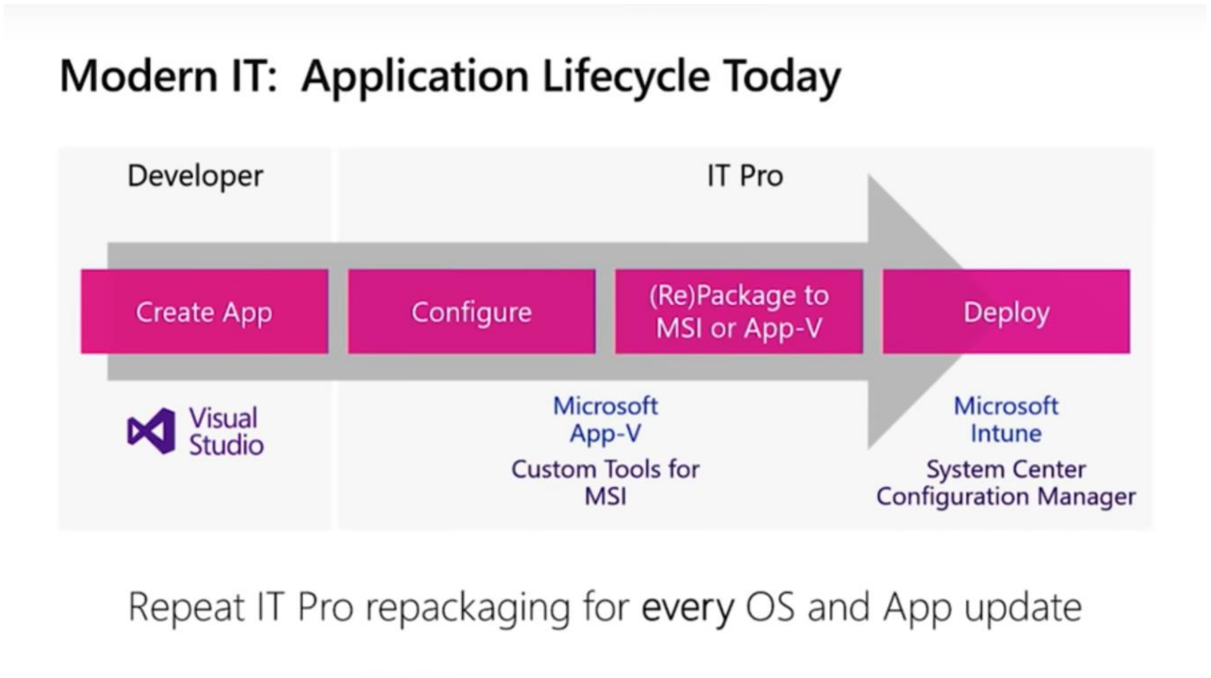
In this chapter, we'll talk about MSIX. MSIX is the brand-new technology from Microsoft that tries to capture the best of previous technologies to provide a solid foundation for the packaging technology of the future.

What does a packaging technology have to do with modernization? Well, it turns out that packaging is fundamental for the enterprise IT with lots of money invested there. Modernization isn't only related to using the latest technologies. It's also related to reducing time to market from the moment a business requirement is defined until your company delivers the feature to your client.

The modern application lifecycle

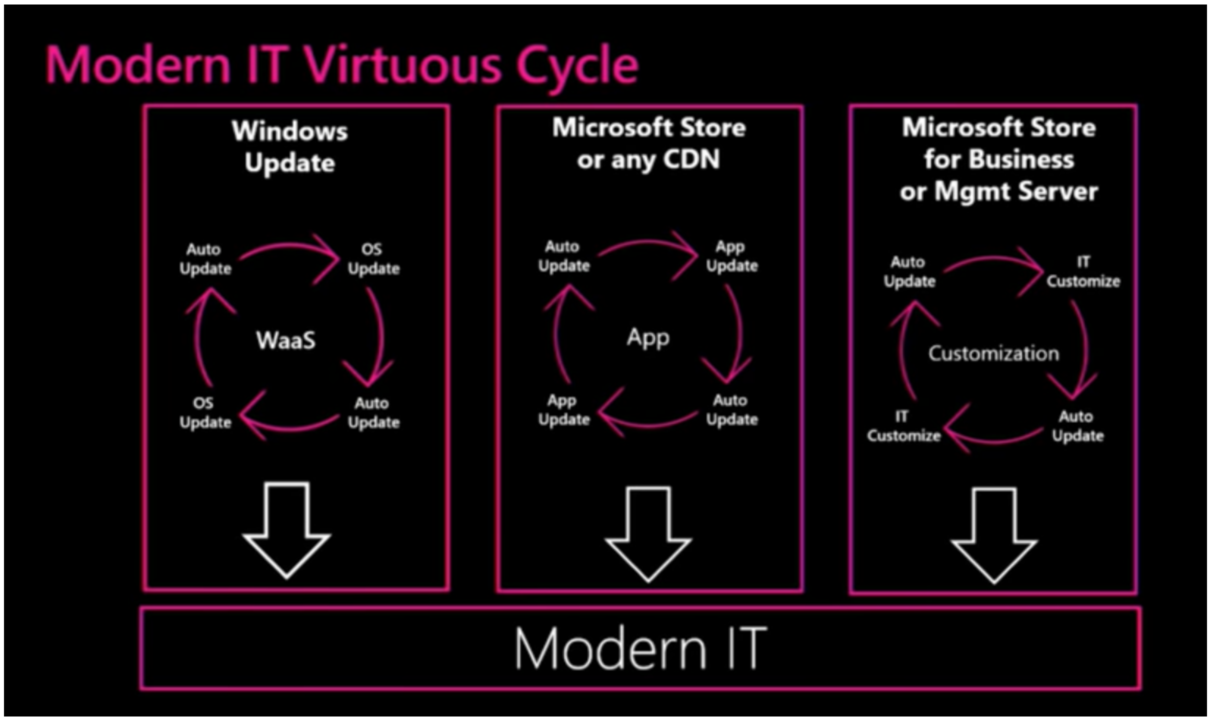
Today, developers write and build the code for an app and then pass the generated assets to the IT Pros. Then, the IT Pros reconfigure the app and repackage it, typically in an MSI or more recently in an App-V packaging format. The app is then deployed through different channels and tools. One of the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

You can see the process reflected on the following picture:


Modern IT: Application Lifecycle Today
Developer
IT Pro
Create App
Configure
(Re)Package to MSI or App-V
Deploy
Visual Studio
Microsoft App-V
Custom Tools for MSI
Microsoft Intune
System Center Configuration Manager
Repeat IT Pro repackaging for every OS and App update

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

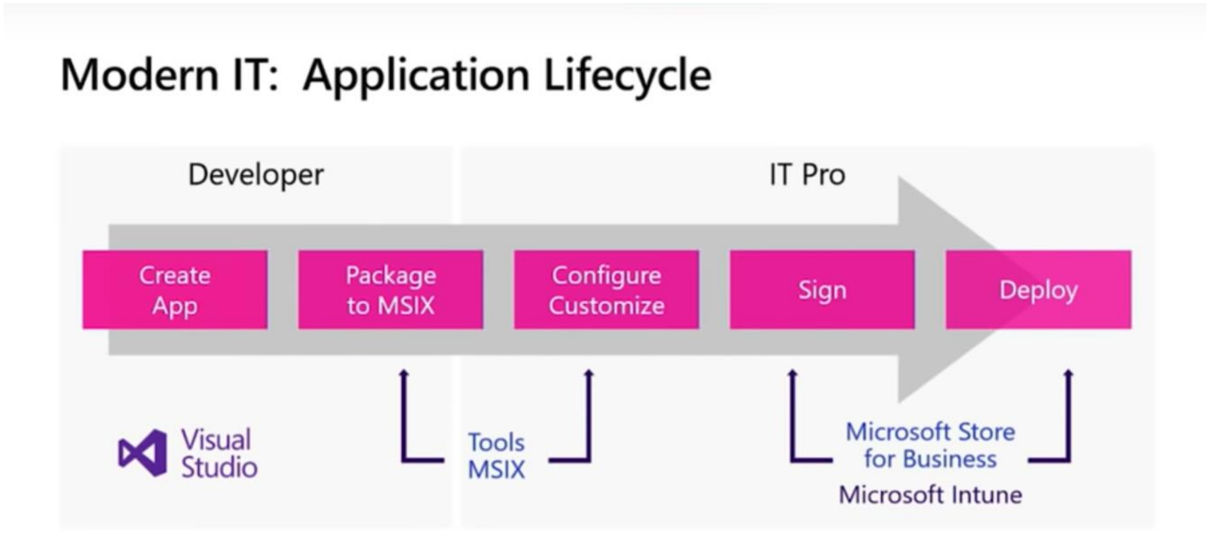
- OS updates
- Application updates
- Customization



The previous diagram shows that you can:

- Update the underlying OS without having to repackage your apps.
- Enable customizations from IT without the need to repackage the original developer package.

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



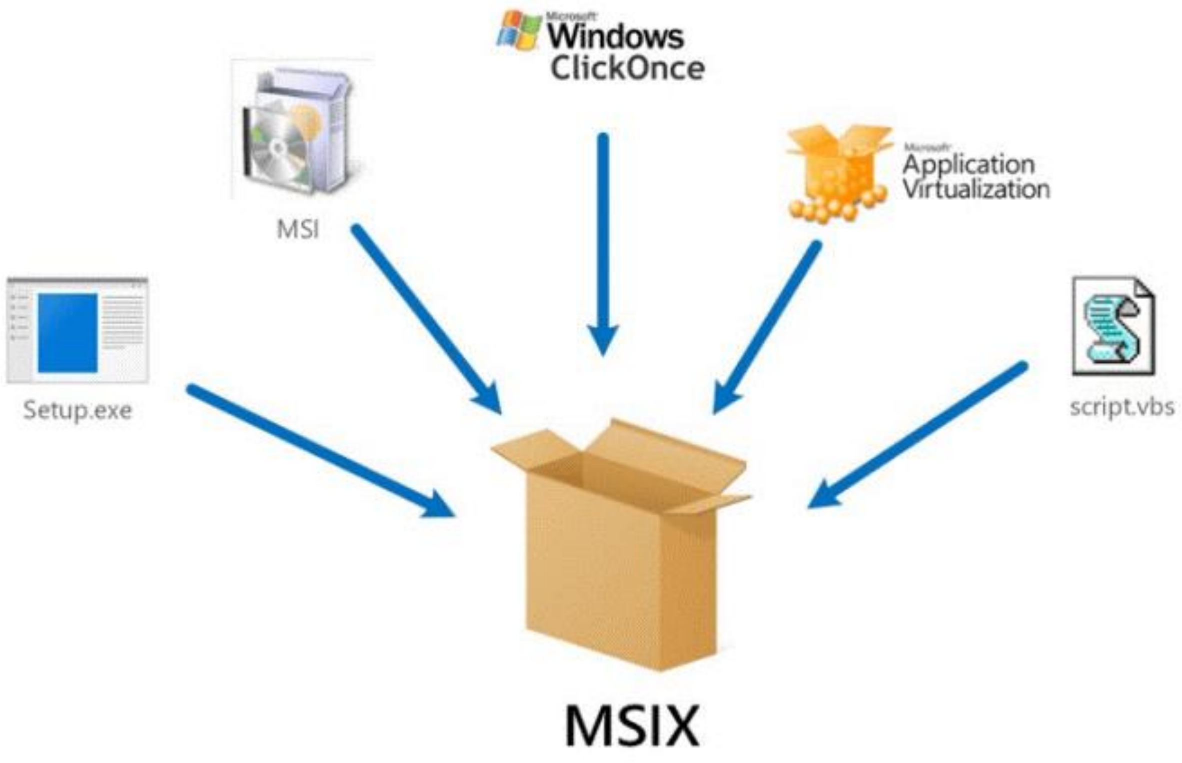
Developers create the app and generate an MSIX package that IT Pros can consume and configure without the need of repackaging. Along with the MSIX technology, Microsoft has created tools to allow IT to customize and configure packages without repackaging.

MSIX: The next generation of deployment

Before MSIX, there were several packaging technologies available like setup wizards, MSI, ClickOnce, App-V, and scripting. Each of these technologies has their own strengths and Microsoft has decided to pick the best of all to build MSIX. MSIX is built on the foundations of these existing technologies picking the best of each:

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

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


Microsoft Windows ClickOnce
MSI
Microsoft Application Virtualization
Setup.exe
script.vbs
MSIX

Benefits of MSIX

Never regret installing an app

MSIX provides a predictable, reliable, and safe deployment. 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.

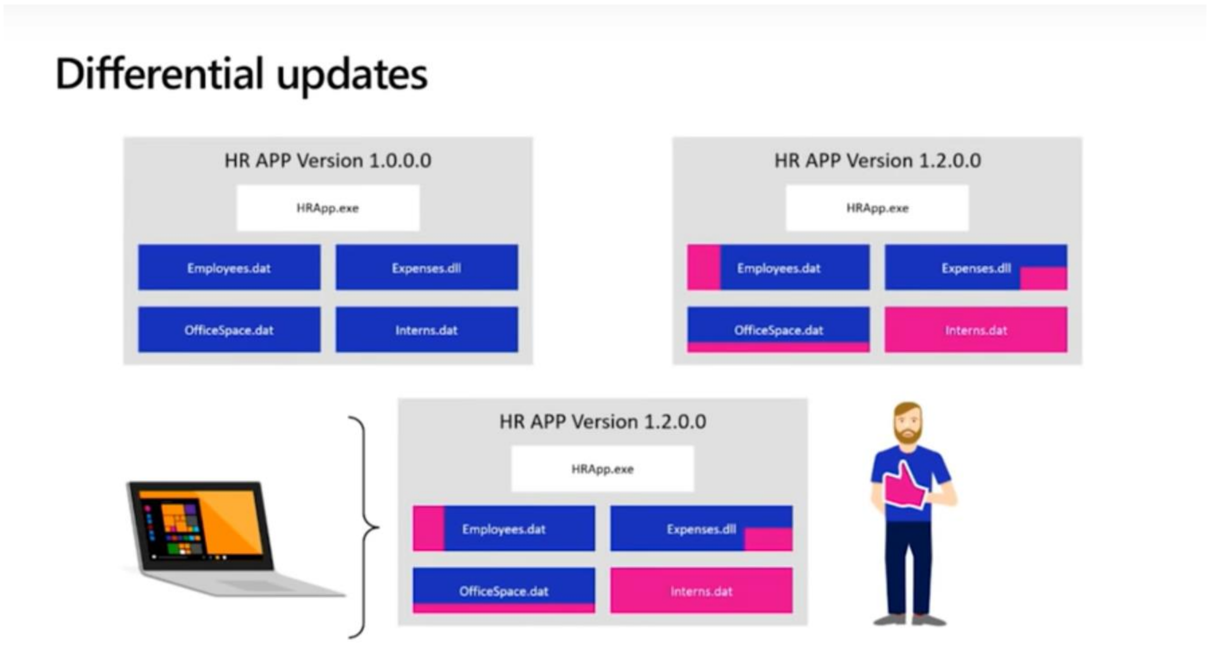
Disk space optimization

MSIX is optimized to reduce the footprint that an application has on the user's machine disk space. It creates a single instance storage of your files. That is, if you have two different packages with the same DLL, the DLL isn't installed twice. The platform takes care of that problem because it knows all the files that a particular app installed 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 block level enabling a feature called differential updates. What this means is that only the updated byte blocks are downloaded on application updates.



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

With the 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 identify in a standard way for every application. It also provides a way to sign your assets providing a solid security foundation.

OS managed

The OS handles all the processes for installing, updating, and removing an application. Applications are installed per user but downloaded only once, minimizing the disk footprint. Microsoft is working on providing the MSIX experience also on Windows 7.

Windows provides integrity for the app

With the use of digital signatures, you can guarantee that you don't install an application from untrusted sources. MSIX also prevents tampering because:

- It keeps a record of file hashes.
- It detects if a 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, even if you want to do xCopy deployment, ClickOnce, or going to the Store, you can use the same MSIX package.

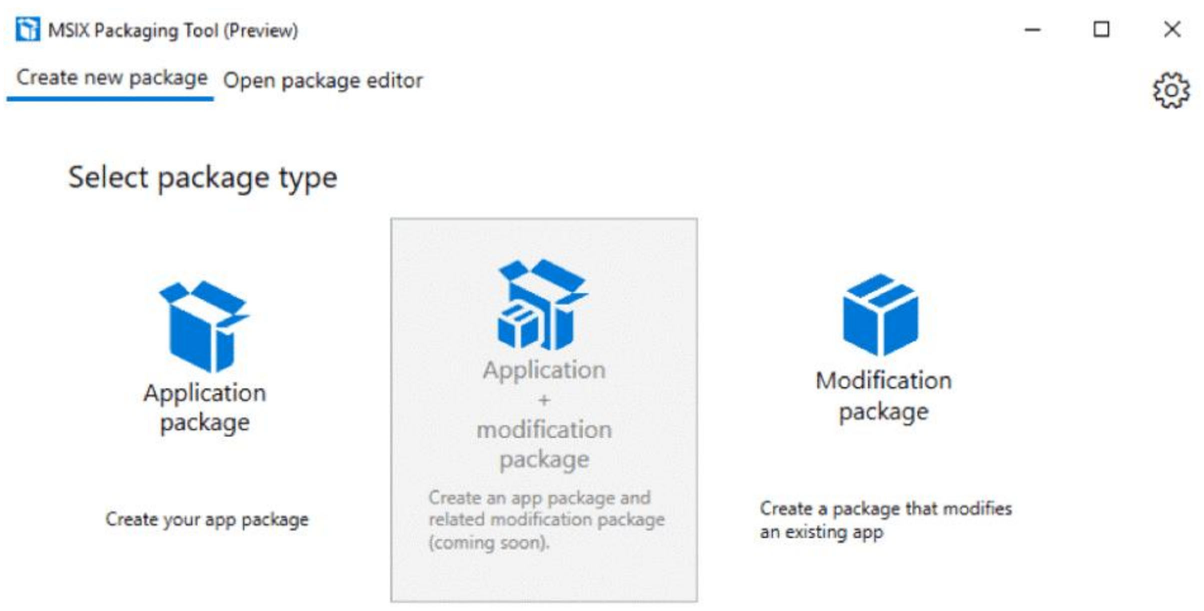
Tools

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 to the MSIX format. It offers both an interactive UI and a command line for conversions and gives you the ability to convert an application without having the source code.


MSIX Packaging Tool (Preview)
Create new package
Open package editor
Select package type
Application package
Create your app package
Application + modification package
Create an app package and related modification package (coming soon).
Modification package
Create a package that modifies an existing app

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.

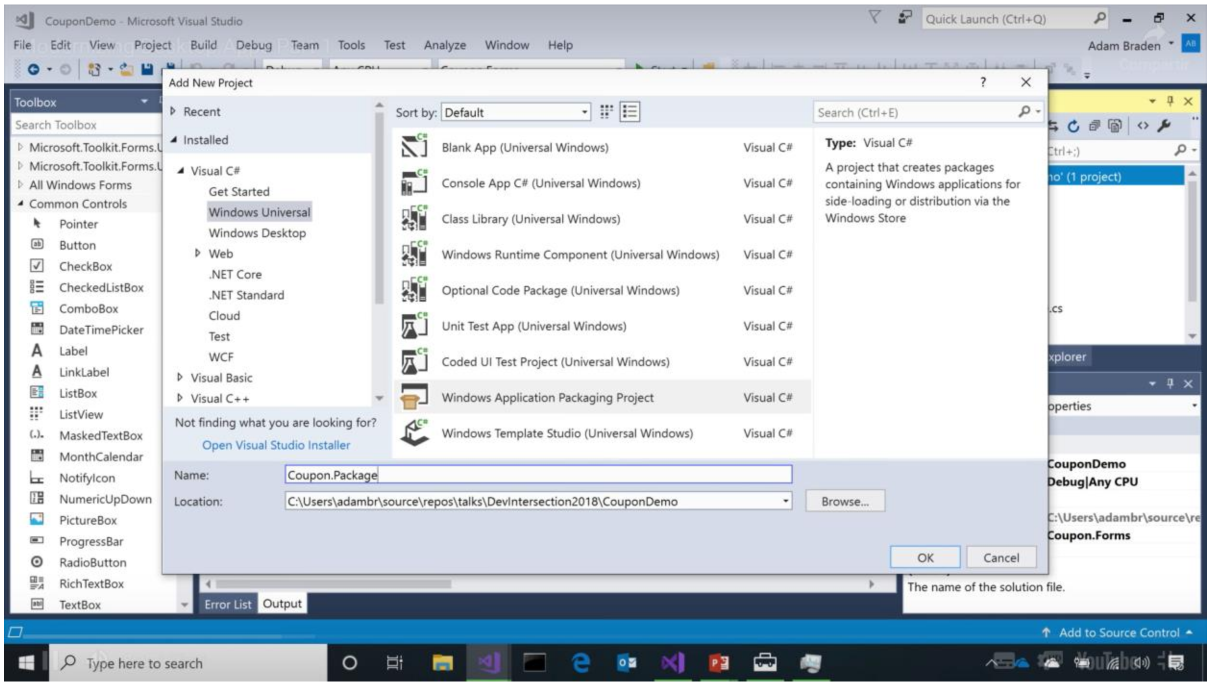
App Installer

App Installer allows 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 app from the web, optional packages, and related sets.

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

Let's go through the process to create an MSIX package from an existing Win32 application. In this example, we'll use a Windows Forms app.

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


CouponDemo - Microsoft Visual Studio
Add New Project
Recent
Installed
Visual C#
Get Started
Windows Universal
Windows Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual C++
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Blank App (Universal Windows) Visual C#
Console App C# (Universal Windows) Visual C#
Class Library (Universal Windows) Visual C#
Windows Runtime Component (Universal Windows) Visual C#
Optional Code Package (Universal Windows) Visual C#
Unit Test App (Universal Windows) Visual C#
Coded UI Test Project (Universal Windows) Visual C#
Windows Application Packaging Project Visual C#
Windows Template Studio (Universal Windows) Visual C#
Browse...
OK
Cancel

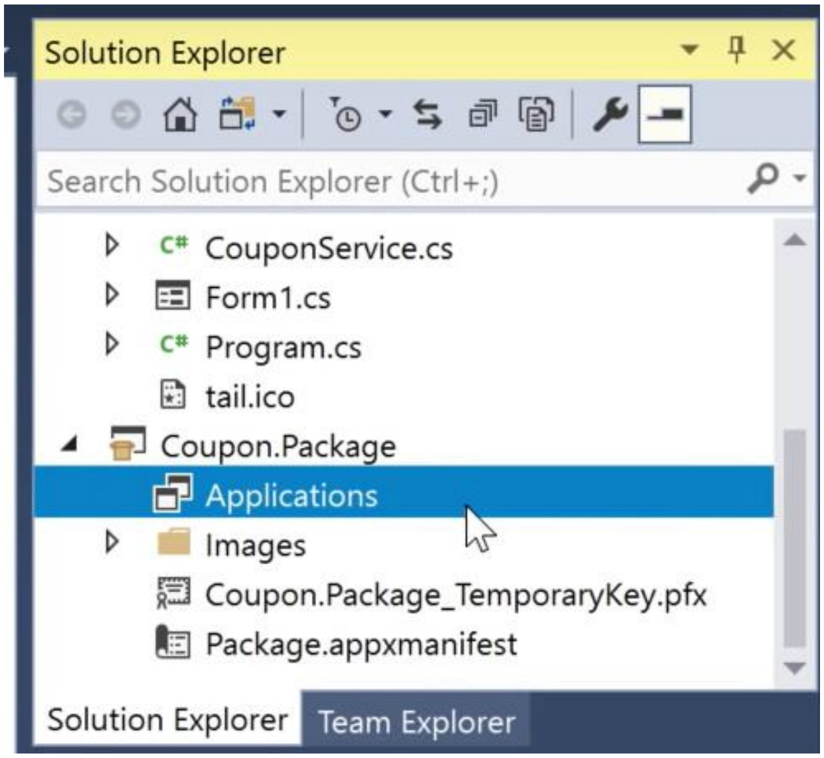
Type: Visual C#

A project that creates packages containing Windows applications for side-loading or distribution via the Windows Store

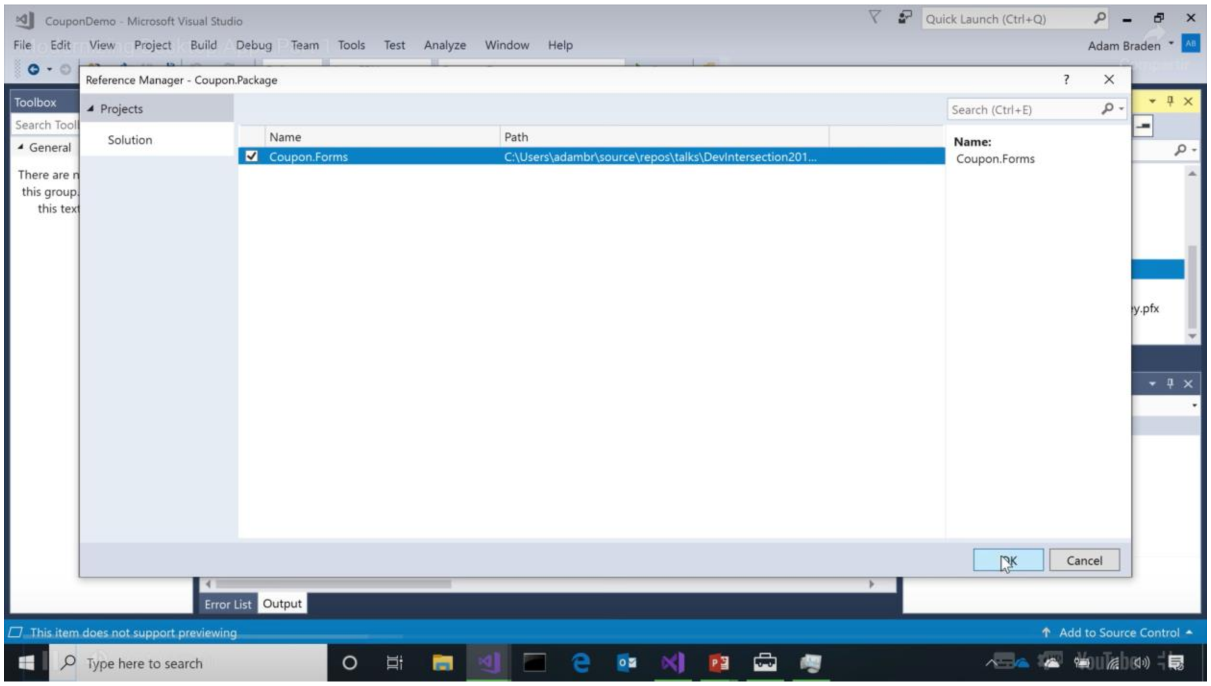
Name: Coupon.Package

Location: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo

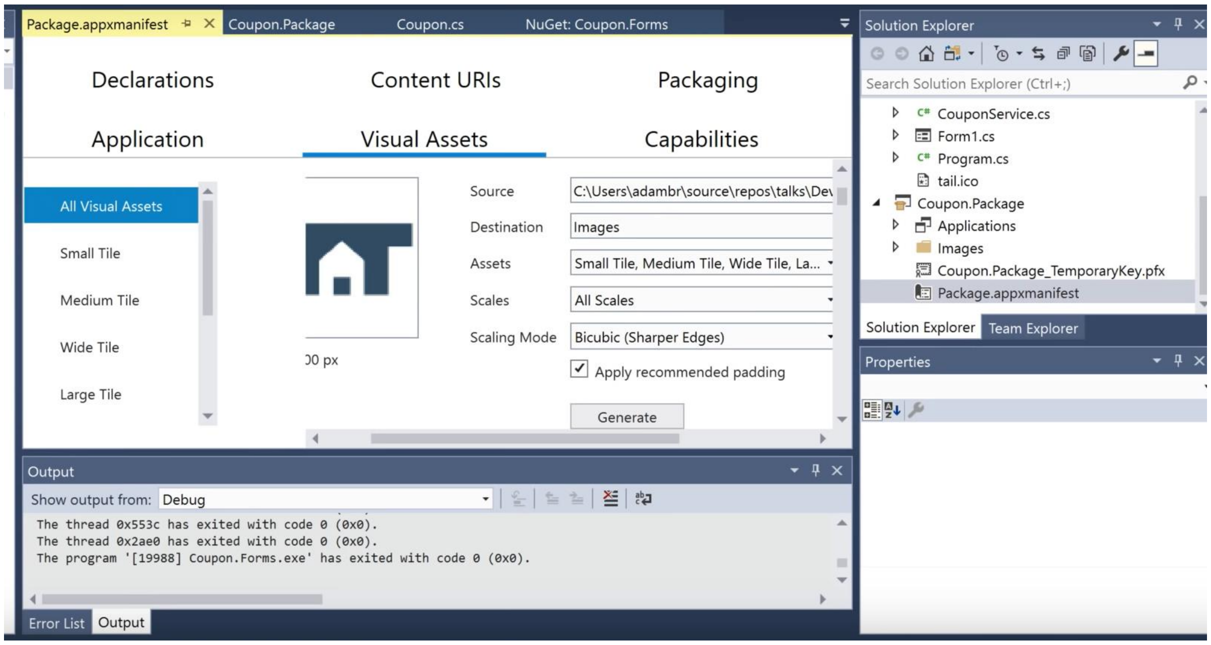
You'll see the structure of the packaging project and note a special folder called *Applications*. Inside this folder, you can specify which applications you want to include in the package. It can be more than one.



Right-click on the *Applications* folder and select the Windows Forms project you want to package from the Visual Studio solution.


CouponDemo - Microsoft Visual Studio
Reference Manager - Coupon.Package
Projects
Solution
Name
Path
Coupon.Forms
C:\Users\adambr\source\repos\talks\DevIntersection201...
Name:
Coupon.Forms
OK
Cancel

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



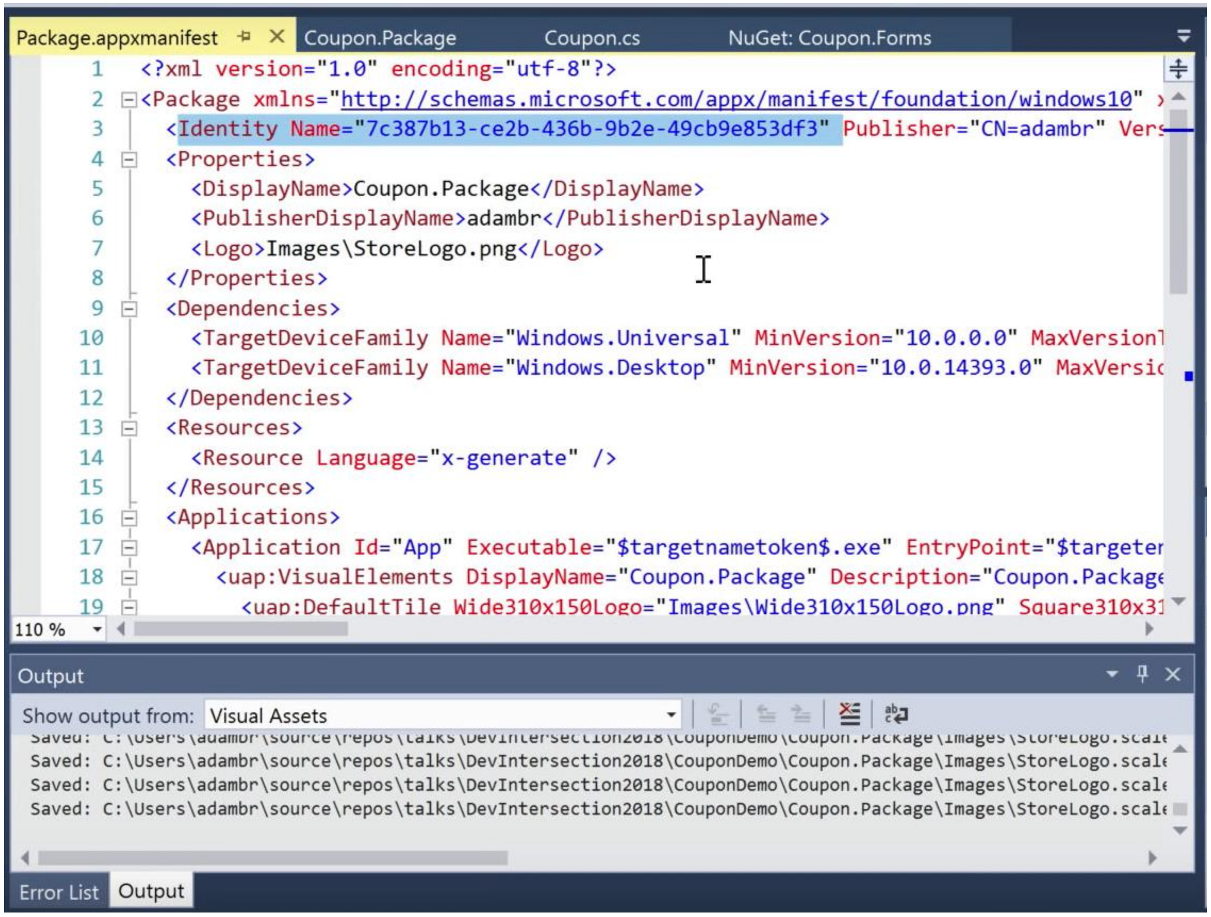
Source: C:\Users\adambr\source\repos\talks\Dev
Destination: Images
Assets: Small Tile, Medium Tile, Wide Tile, La...
Scales: All Scales
Scaling Mode: Bicubic (Sharper Edges)
✓ Apply recommended padding

Show output from: Debug

```
The thread 0x553c has exited with code 0 (0x0).
The thread 0x2ae0 has exited with code 0 (0x0).
The program '[19988] Coupon.Forms.exe' has exited with code 0 (0x0).
```

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

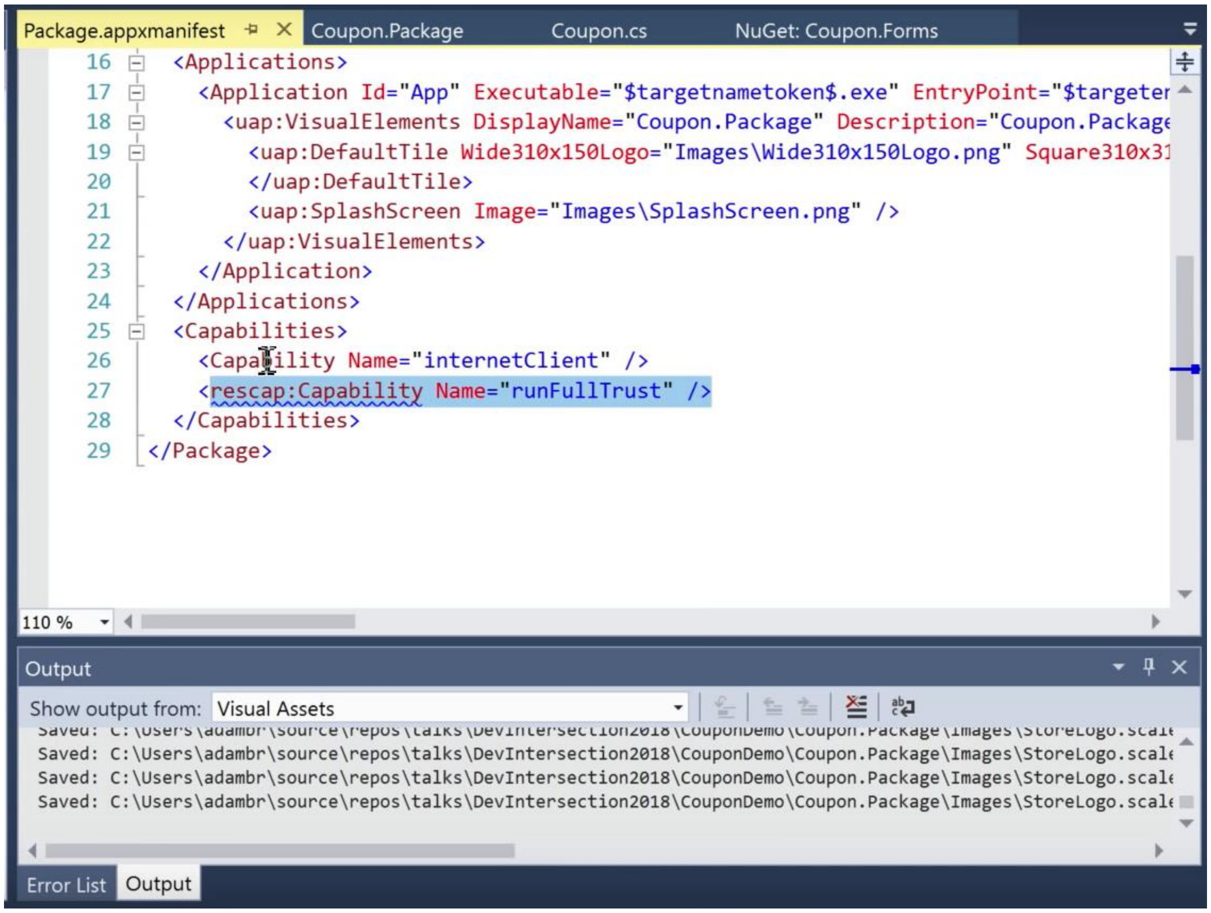
Right under `<Package>`, there's an `<Identity>` node. This is where your packaged application is going to get its identity, which will be managed by the OS.



```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10" >
  <Identity Name="7c387b13-ce2b-436b-9b2e-49cb9e853df3" Publisher="CN=adambr" Vers
  <Properties>
    <DisplayName>Coupon.Package</DisplayName>
    <PublisherDisplayName>adambr</PublisherDisplayName>
    <Logo>Images\StoreLogo.png</Logo>
  </Properties>
  <Dependencies>
    <TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.0.0" MaxVersionT
    <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.14393.0" MaxVersic
  </Dependencies>
  <Resources>
    <Resource Language="x-generate" />
  </Resources>
  <Applications>
    <Application Id="App" Executable="$targetnametoken$.exe" EntryPoint="$targeter
      <uap:VisualElements DisplayName="Coupon.Package" Description="Coupon.Package
        <uap:DefaultTile Wide310x150Logo="Images\Wide310x150Logo.png" Square310x31
```

```
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
```

In the `<Capabilities>` node, you can find all the requirements the application needs, paying special attention to the `<rescap:Capability Name="runFullTrust" \>`, which tells the OS to run the app in full trust mode since it's a Win32 application.

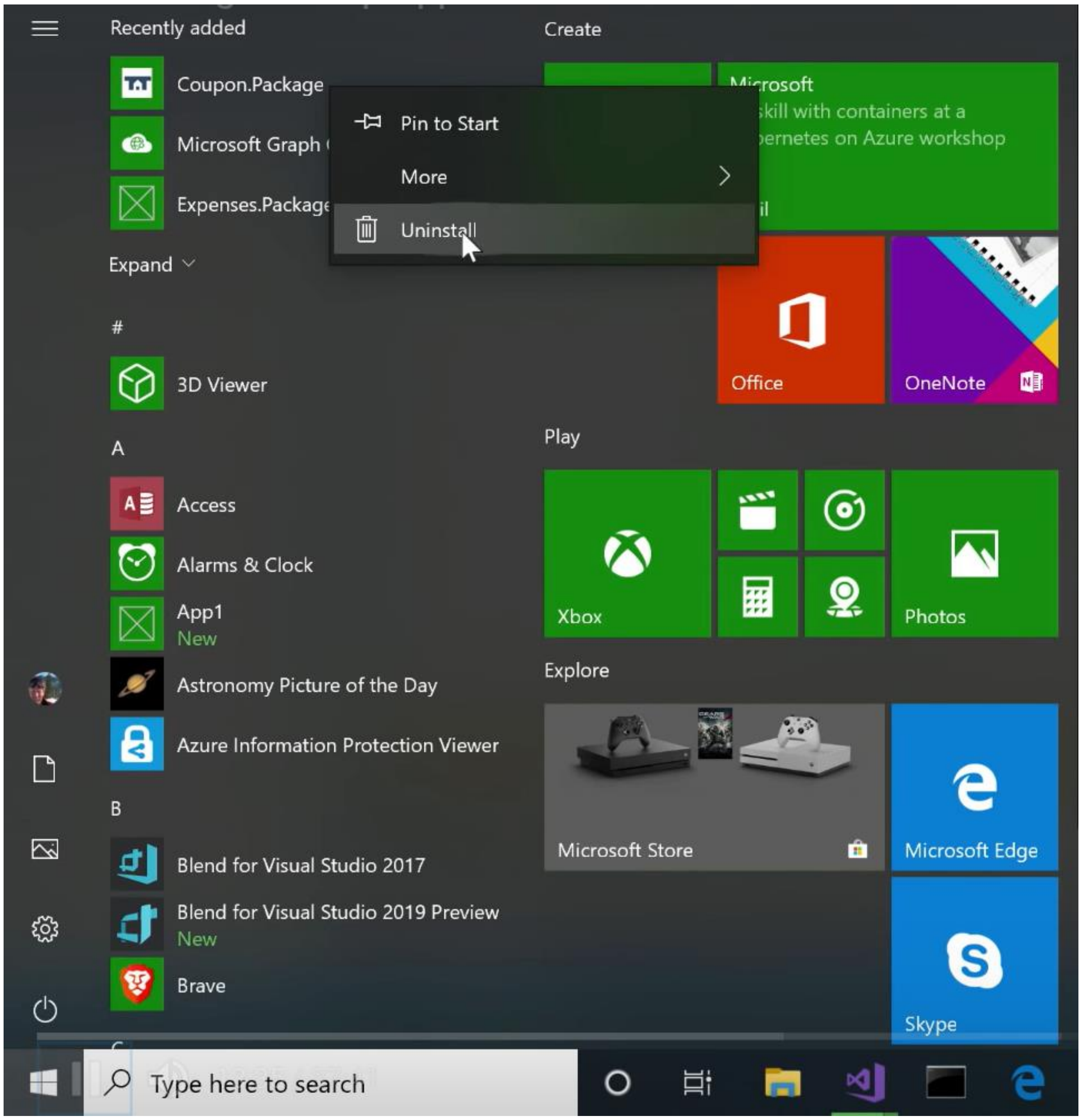


```
  <Applications>
    <Application Id="App" Executable="$targetnametoken$.exe" EntryPoint="$targeter
      <uap:VisualElements DisplayName="Coupon.Package" Description="Coupon.Package
        <uap:DefaultTile Wide310x150Logo="Images\Wide310x150Logo.png" Square310x31
        </uap:DefaultTile>
        <uap:SplashScreen Image="Images\SplashScreen.png" />
      </uap:VisualElements>
    </Application>
  </Applications>
  <Capabilities>
    <Capability Name="internetClient" />
    <rescap:Capability Name="runFullTrust" />
  </Capabilities>
</Package>
```

```
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
Saved: C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\Images\StoreLogo.scale
```

Set the packaging project as the startup project for the solution and select *Run*. This is going to:

- Compile the Windows Forms application.
- Create an MSIX package out of the build results.
- Deploy the packages.
- Install it locally on the development machine.
- Launch the app.

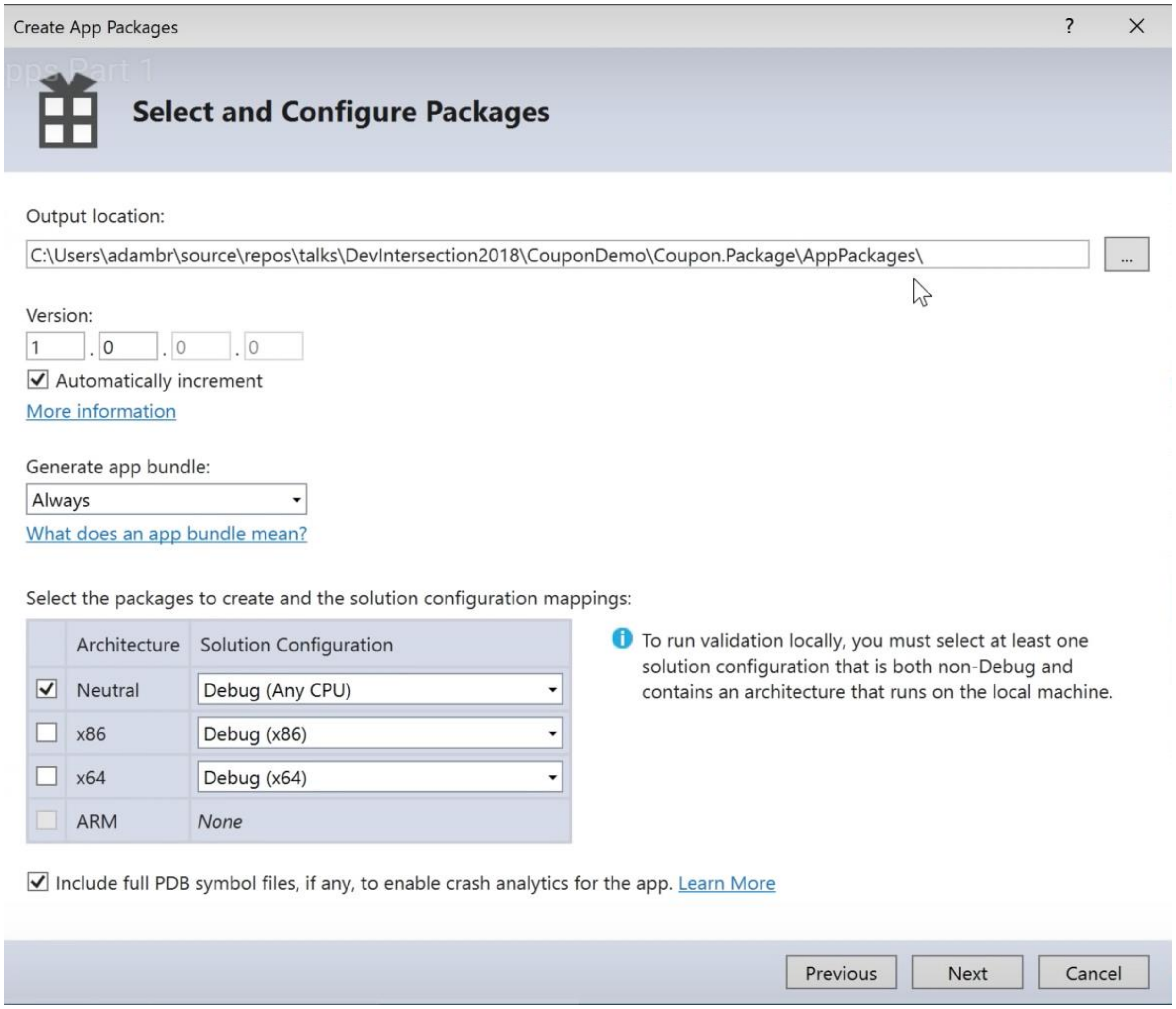


With this, you have the clean install and uninstall experience that MSIX provides fully integrated into Windows 10.

The final stage is about how you deploy the MSIX package to another machine.

Right-click on the packaging project, select the **Store** menu, and then select the **Create App Packages** option.

Then, you can choose between creating a package to upload to the store or creating packages for sideloading. In most modernization scenarios, you'll choose **I want to create packages for sideloading**.



Output location:

C:\Users\adambr\source\repos\talks\DevIntersection2018\CouponDemo\Coupon.Package\AppPackages\

Version:

1 . 0 . 0 . 0

☑ Automatically increment

More information

Generate app bundle:

Always

What does an app bundle mean?

Select the packages to create and the solution configuration mappings:

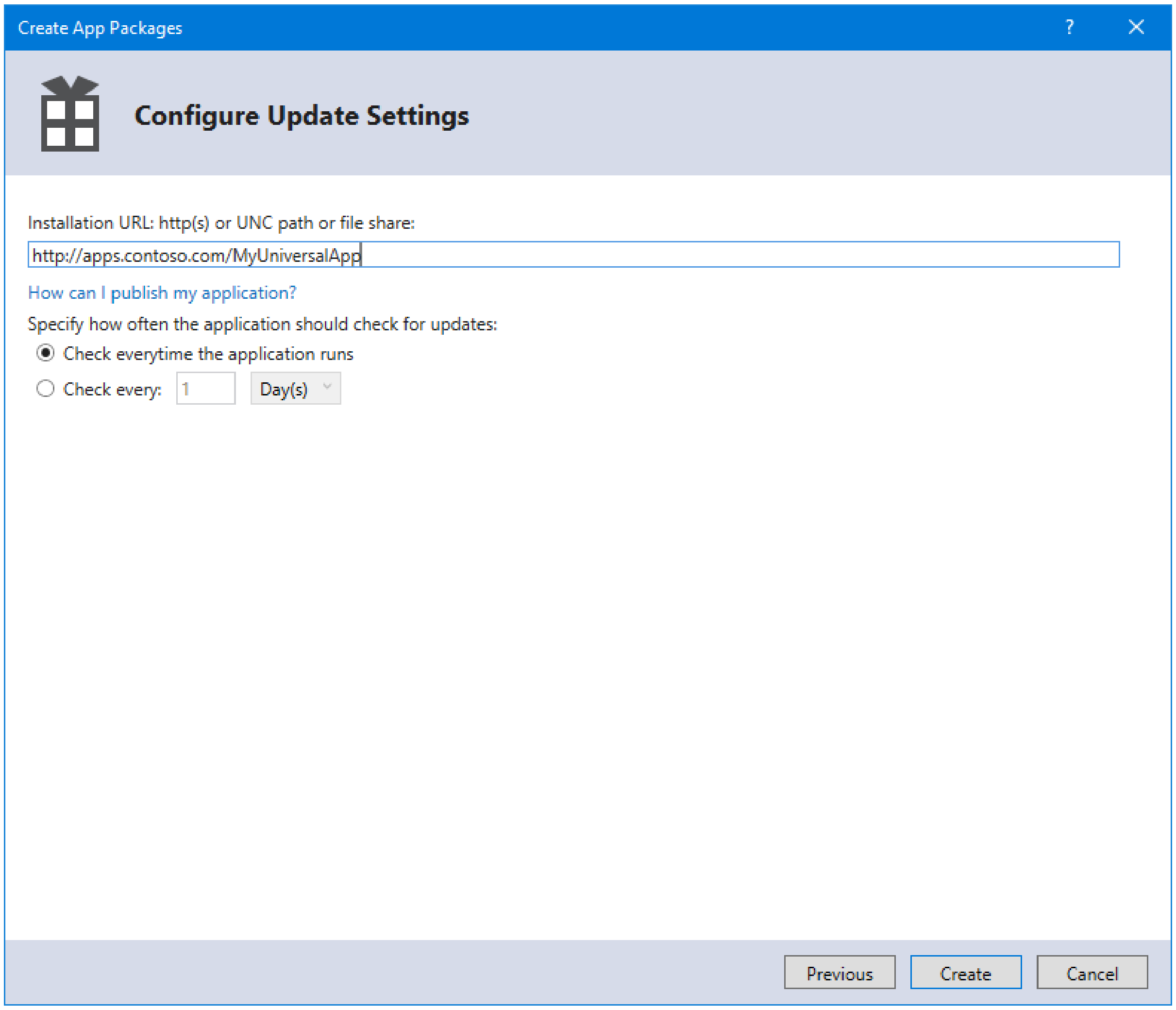
	Architecture	Solution Configuration
☑	Neutral	Debug (Any CPU)
☐	x86	Debug (x86)
☐	x64	Debug (x64)
☐	ARM	*None*

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

There you can select the different architectures you want to target as 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.



Installation URL: http(s) or UNC path or file share:
http://apps.contoso.com/MyUniversalApp

Specify how often the application should check for updates:
● Check everytime the application runs
○ Check every: 1 Day(s)

You can choose to use a web server of a shared UNC path on your enterprise file servers. Pay attention to the settings to specify how you want to update your application. We'll cover application updates in the next section.

For a detailed step-by-step guide, see Package a desktop app from source code using Visual Studio.

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 pull them to your users.

Using a combination of Windows 10 features and MSIX packages, you can provide a great updating experience for your users. In fact, the user doesn't need to be technical at all but still benefit from a seamless application update experience.

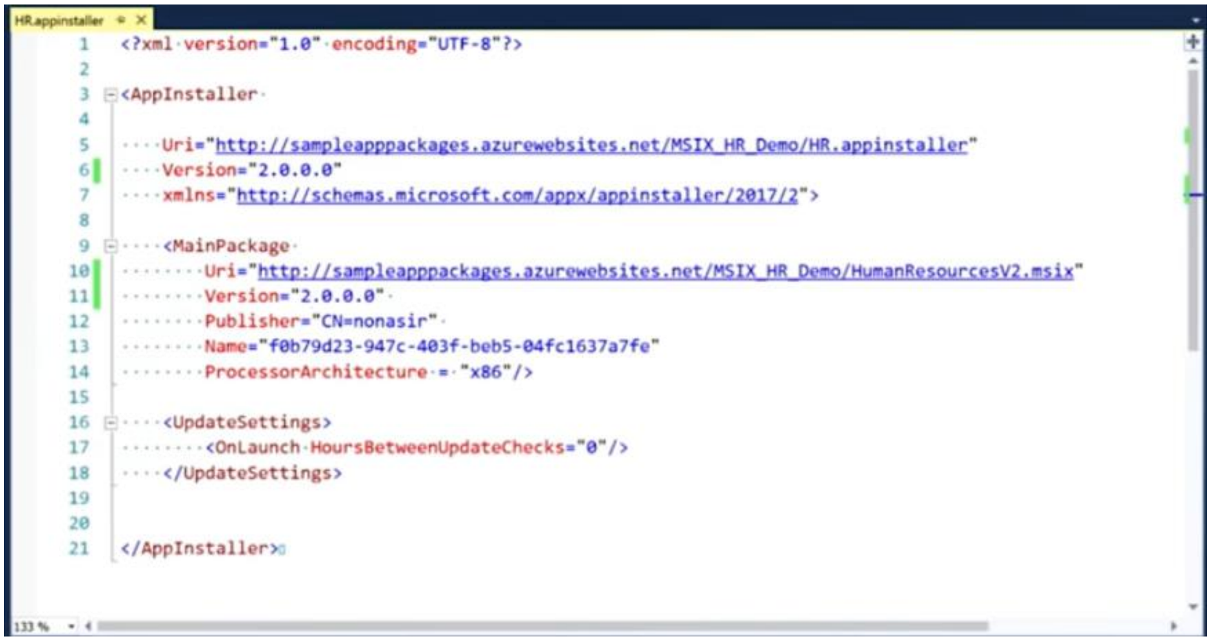
You can configure your updates to interact with the user in two different ways:

- User prompted updates: The OS shows some autogenerated nice UI to notify the user about 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, when the application launches or on a regular basis. Thanks to the side-loading features, you can even get these updates while the application is running.

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

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



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

<AppInstaller

    Uri="http://sampleapppackages.azurewebsites.net/MSIX_HR_Demo/HR.appinstaller"
    Version="2.0.0.0"
    xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">

    <MainPackage
        Uri="http://sampleapppackages.azurewebsites.net/MSIX_HR_Demo/HumanResourcesV2.msix"
        Version="2.0.0.0"
        Publisher="CN=nonasir"
        Name="f0b79d23-947c-403f-beb5-04fc1637a7fe"
        ProcessorArchitecture = "x86"/>

    <UpdateSettings>
        <OnLaunch HoursBetweenUpdateChecks="0"/>
    </UpdateSettings>


</AppInstaller>
```

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

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

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

MSIX creates a user interface for installation automatically showing some properties of the package. This allows for a common installation experience for every app.

Once you’ve generated the new MSIX package and moved it to the deployment server, you just have to edit the *.appinstaller* file to reflect these 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.