MSIX

SUCCINCTLY

BY **MATTEO PAGANI**

SUCCINCTLY EBOOK SERIES

Syncfusion®

MSIX Succinctly

By

Matteo Pagani

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

I'm a developer with a strong passion for client development and, especially, the Windows platform. Since my graduation, I've been able to turn my passion into a full-time job and, additionally, I've started to share it with other developers by opening a blog, writing articles for Microsoft and other independent technical websites, and speaking at many official and community conferences worldwide, like Microsoft Ready, BUILD, Codemotion, WPC, and Microsoft Ignite.

I'm the author of many books about Windows Phone and Universal Windows apps development, both in Italian and English.

I've been a Microsoft MVP in the Windows Platform Development category and a Nokia Developer Champion for almost five years until, in November 2014, I joined Microsoft as a Windows AppConsult engineer. In this role, I support developers all around the world in piloting new development tools and technologies for Windows.

Chapter 1 Introduction to MSIX

“Modernize: to make (something) modern and more suited to present styles or needs.”

In the tech world, we see amazing new software products being launched every day: beautiful websites, rich mobile applications, holographic experiences, and many more. We see these applications being advertised on tech websites, social networks, and sometimes even on television. These products are installed on every smartphone or PC; they are developed with an agile approach and constantly updated; they leverage the latest and greatest development technologies available on the market.

However, there’s an even larger number of applications that are critical for banks, big enterprises, hospitals, and schools. These applications are big and complex, and they were written many years ago. They perform important tasks, so they can’t be easily replaced with a new and modern product.

The world we’re living in is becoming a challenge for these kinds of applications. Users are more tech-savvy today, and they expect more from an application. Security challenges are increasing day by day, and as the number of users grows, applications need to scale accordingly.

Many companies, including Microsoft, have launched great frameworks and tools for building modern native desktop applications, such as Universal Windows Platform, Electron, and Progressive Web Apps. However, in many cases, the developer must start from scratch to leverage them. Unfortunately, due to their complexity, it’s not always feasible to entirely rewrite these applications to accommodate the new requirements

The same challenge applies to application packaging and deployment. Windows 10 introduced a more powerful deployment model, but in the beginning, it was applicable only to the new generation of modern applications, based on platforms like the Microsoft Store or the Microsoft Store for Business.

This is where app modernization comes in. Why do we have to completely rewrite an application? Why not just use the existing product and focus on the components that need to be enhanced? Or why can’t we use the same deployment techniques of modern applications for classic desktop applications?

In this book, we’re going to understand better how Windows 10 can help us take our Windows applications to the next level without having to rewrite them. The starting point will be MSIX, a new packaging format that Microsoft introduced in Windows 10 1809 (which was launched in October 2018). It aims to replace all the existing packaging and deployment technologies, like MSI and App-V.

The Universal Windows Platform

With the release of Windows 8, Microsoft introduced a new kind of application: Windows Store apps, based on a new framework called Windows Runtime. Unlike the .NET Framework, the Windows Runtime is a native layer of APIs that are exposed directly by the operating system to applications that want to consume them. With the goal to make the platform viable for every developer, the Windows Runtime introduced *language projections*, which are layers added on top of the runtime to allow developers to interact with it using well-known and familiar languages, like C# and C++.

The Windows Runtime libraries (called Windows Runtime components) are described using special metadata files that make it possible for developers to access the APIs using the specific syntax of the language they're using. This way, projections can also respect the language conventions and types, like uppercase if you use C#, or camel case if you use JavaScript. Additionally, Windows Runtime components can be used across multiple languages; for example, a Windows Runtime component written in C++ can be used by an application developed in C# and XAML.

With the release of Windows 10, Microsoft introduced the Universal Windows Platform, which can be considered the successor of the Windows Runtime, since it's built on top of the same technology. The unique point of the Universal Windows Platform is that it offers a common set of APIs across every platform; whether the app is running on a desktop, an Xbox One, or a HoloLens, you're able to use the same APIs to reach the same goals. This was a major step forward compared to the original Windows Runtime, which didn't provide this kind of cross-device support. You were able to share code and UI between a PC project and a mobile project, but, in the end, developers needed a way to create, maintain, and deploy two different solutions.

Figure 1 shows where the Universal Windows Platforms fits in the overall Windows 10 architecture: every device, despite having a different shell, shares the same kernel, which is the Windows Core. On top of the core, the Universal Windows Platform provides a consistent set of APIs to access to all the functionalities that are exposed by the core, like network access, storage, and sensors. A Windows app runs on top of the Universal Windows Platform, and it's a single binary package.

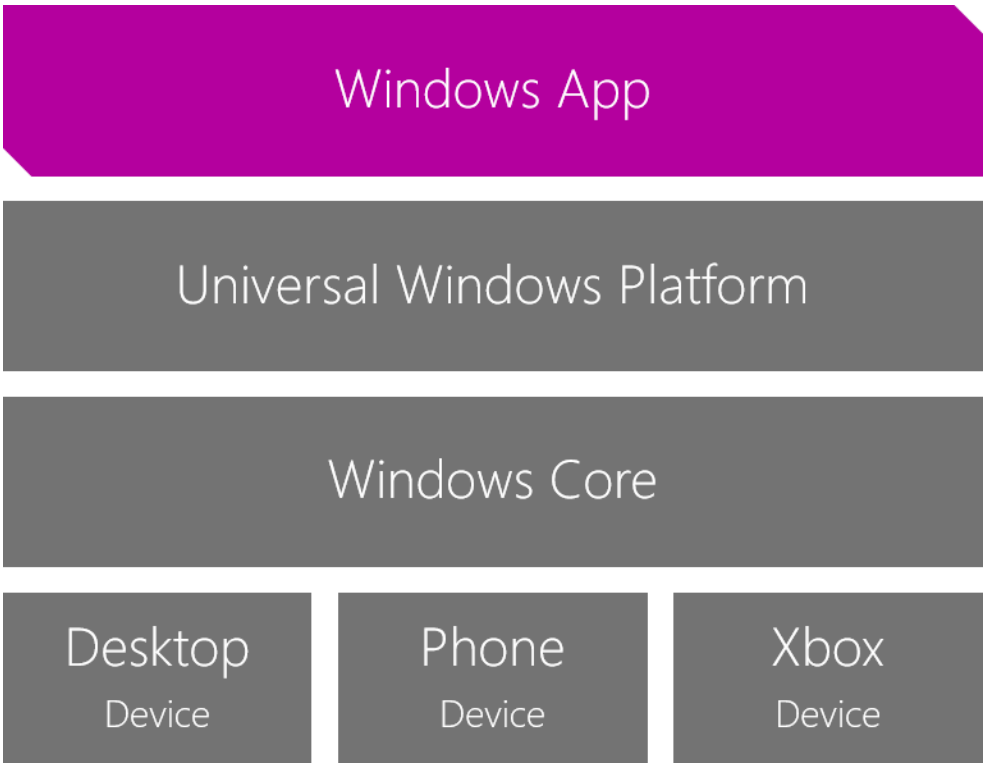


Figure 1: The Windows 10 ecosystem for developers

Another important shift in the Universal Windows Platform is that, when it comes to the C# and XAML projection, it doesn't leverage the standard .NET Framework, but a new, modern framework called .NET Core.

.NET Core can be considered the successor of .NET Framework, and was built from scratch according to the new Microsoft principles: open source and cross platform. The adoption of .NET Core and .NET Standard, the new formal specification of .NET APIs, makes it easier to reuse the parts of the code base that don't have a specific dependency on the platform with other projects, no matter if it's a web application or a traditional desktop application. However, it's important to underline that this doesn't mean that Universal Windows Platform apps can also run on other platforms like Android and iOS. The Universal Windows Platform, in fact, is built on top of .NET Core, and it leverages its common base, but it adds a set of APIs and features that are Windows-specific.

.NET Core, in the first releases, was focused on building cloud and web workloads. Things are changing with the release of .NET Core 3.0, which allows desktop frameworks (WPF and Windows Forms) to run on top of it instead of using the full .NET Framework.

Modern applications and desktop applications

With the release of the Windows Runtime in Windows 8 first, and the Universal Windows Platform in Windows 10 later, the Windows team has added first-class support to all the new features that you can expect from a modern application, like:

- Native support for new interaction paradigms, like touch, inking, and voice.
- New security features, like authentication with biometric parameters (fingerprint, face recognition, etc.).
- Fluid user experience and animations.
- Integration with all the new Windows 10 features, like Cortana, Timeline, live tiles, and cross-device communication.
- A better deployment system.

The Universal Windows Platform is also frequently updated. Windows 10 introduced the concept of *Windows as a service.* The operating system is constantly updated with new features, which are deployed twice per year with the release of a feature update. Every update also comes with a new version of the Windows SDK, which brings new APIs, improvements, and support to interact with the new features.

Universal Windows Platform applications have a minimum footprint and a better security model, thanks to the usage of containers. They can be easily installed and kept up to date using distribution platforms like the Microsoft Store or the Microsoft Store for Business. In addition to providing access to a wider set of APIs, the Universal Windows Platform introduced a very different application model compared to the existing one leveraged by Win32 applications, since it was built with security and privacy in mind. As such, Universal Windows Platform applications run inside a sandbox, they don't have access to the registry, and they can freely read and write data only in a specific local folder. Any operation that is potentially dangerous requires the declaration of a capability and the consent of the user. Some examples are accessing the files in the Pictures library, using the microphone or the webcam, and retrieving the location of the user. Everything is controlled by a *manifest file*, which is an XML file that describes the identity of the application: its unique identifier, its capabilities, its visual aspect, and its integration with the Windows 10 extension points. These applications are deployed using new platforms like the Microsoft Store or the Microsoft Store for Business, which solve many of the distribution challenges, like auto-updates and license tracking.

On the other hand, classic desktop platforms like Windows Forms and WPF are considered stable and mature and are often still the first choice when it comes to developing a traditional enterprise application optimized for mouse and keyboard. Additionally, they can leverage the whole Windows API surface, so they are limited only by the running context (standard user vs. administrator). They can read and write from the registry, read and write in any folder, and interact with external peripherals. This approach is very powerful, but it also has some downsides. Applications can perform malicious tasks, like accessing any file on your computer, or interacting with any hardware device connected to the computer. When it comes to distribution, enterprises need flexibility, and they need to be able deploy applications in multiple ways: from a website, using System Center Configuration Manager or Intune, etc. As such, IT pros leverage technologies like MSI, ClickOnce, or App-V to satisfy all the requirements of the company.

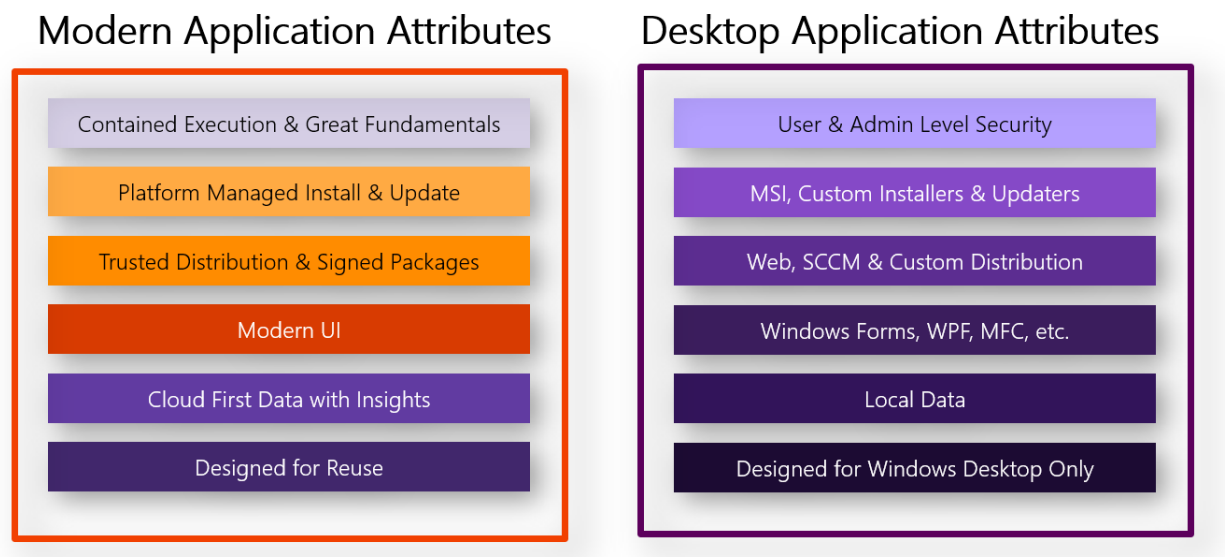


Figure 2: The main features of the two development ecosystems for Windows

In the beginning, the two worlds we have just described were in conflict—either you developed your applications for the modern desktop by leveraging the Universal Windows Platform model, or you kept using the Win32 technologies like WPF and Windows Forms. These technologies are still perfectly reliable and supported, but they don't allow you to take advantage of all the benefits of a modern ecosystem, and they can't be deployed using the new Windows 10 model.

The release of the Windows 10 Anniversary Update has started to change this approach by introducing a technology called Desktop Bridge. Thanks to it, traditional desktop applications had the chance to leverage the same deployment model of the Universal Windows Platform.

Desktop Bridge allows us to describe Win32 applications with a manifest file, too, which means that they get an identity. For developers, this feature allows the usage of APIs of the Universal Windows Platform. In the following years, the story has evolved with many new tools and technologies for app modernization: MSIX, XAML Islands, .NET Core 3.0, etc. For IT pros, this means that classic desktop applications can take advantage of all the benefits of the new deployment model introduced in Windows 10.

The goal is to remove the barrier between traditional desktop development and modern development. No matter which development technology you're using, you have the chance to leverage all the benefits of a modern ecosystem without rewriting your application from scratch by integrating the existing code with the features you're interested in.

MSIX, the new packaging format introduced in Windows 10, version 1809, is the starting point of this modernization journey. Let's learn a bit more about it.

A new approach to packaging

Windows 8 was the first version of Windows that introduced a new packaging technology for applications, called AppX. At the time of the release, however, it wasn't very flexible. It supported only Windows Store applications, packages could be deployed only from the Store, and sideloading was enabled only on devices unlocked for development purposes or with a special license.

Windows 10, release after release, has continued to enhance this packaging format by opening it up more and more to be suitable for any kind of app distribution, regardless of the technology (UWP, WPF, Windows Forms, Java, etc.) and the deployment approach (Store, enterprise management tools, sideloading, etc.).

Here is the timeline of the key changes:

- **Windows 10:** The first release of Windows 10 removed the requirement to have a special license to sideload enterprise applications packaged using AppX. It was enough to turn on the **Sideload apps** option in the Windows settings.
- **Windows 10, version 1511:** The first update for Windows 10, called the November Update, has turned on by default the option to allow package sideloading. Every user, without changing the Windows configuration, was able to install packages coming from a trusted source, like your workplace.
- **Windows 10, version 1607:** The Anniversary Update introduced the Desktop Bridge. Thanks to this technology, the AppX packaging format has been extended to the Win32 ecosystem. From now on, applications built with WPF, Windows Forms, Java, etc., can be packaged using AppX and take advantage of the new deployment format and the opportunity to be published on the Microsoft Store.
- **Windows 10, version 1809:** This version of Windows introduced MSIX, the evolution of the AppX format. Built on the same fundamentals, it further opened the ecosystem to better support enterprise scenarios and helped IT departments move to a better deployment approach.

Another big change compared to AppX is that MSIX is aligned to the new Microsoft philosophy and, as such, it's an open-source and cross-platform project. It offers SDKs to work with this packaging format for all the major platforms, either mobile (Android and iOS) or desktop (MacOS, Linux). Of course, this doesn't mean that MSIX packages can be deployed out of the box on every platform. The owner of the other operating systems, in fact, would need to implement MSIX support first. However, thanks to these SDKs, we aren't limited to creating or unpacking MSIX packages only on Windows.

The benefits of MSIX for developers

Until now, the installation and deployment of Windows applications never followed a specific standard. The developer has always been in charge of identifying the best experience for customers, often by leveraging third-party tools.

As such, developers are required to take care of many tasks that go beyond the development of the application, like:

- Identifying the right installer technology from among the ones available on the market (MSI, ClickOnce, InstallShield, etc.).
- Handling the installation and uninstallation process to make sure that everything is deployed in the right place and that, when the user doesn't need the app anymore, they can uninstall it, leaving the system in a reliable state.
- Implementing a custom solution to handle the auto-update of the application. Delivering a bad update experience can lead users to stick with the version they have downloaded and never update it, which prevents them from accessing new features or critical security updates.
- Providing a platform to distribute and sell their application, like a website integrated with a monetization system.

This approach has many downsides for the final users, especially since the rise of the mobile market. Users are now accustomed to having a single point of access to download applications (typically a store), which provides a seamless experience. You press a button, and the application is installed. You press another button, and the application is removed. Everything is automatically kept up to date, without any user intervention. The purchase process is also safer, because you aren't sharing your credit card data directly with the developer, but with the owner of the platform, which is typically a trusted company (such as Apple or Google).

For most desktop applications, this isn't the case. Applications are distributed from a website, which the user must find with a search engine and browser. Sometimes, this approach leads to the download of malicious applications that use popular brands and names to spread malware and viruses. Additionally, based on the installer technology chosen by the developer, installing, uninstalling, or updating an application can be a challenging task, especially for users who aren't really tech-savvy.

Windows 10 introduced a different approach to help users handle Windows applications in an easier way, by providing an experience closer to the one people find on their phones. Thanks to a central point of distribution, the Microsoft Store, users don't have to browse the web to find the application they need. Because of the trusted distribution environment, users can be sure they're downloading the right application, and not a malicious clone. The Microsoft Store can also automatically take care of many tasks that, in the past, needed to be implemented by the developer, like automatic updates or monetization.

In the first releases of Windows 10, this new approach was reserved to Universal Windows Platform applications, since AppX supported only this modern framework. With the Windows 10 Anniversary Update, Microsoft opened it up to classic desktop applications, after introducing the Desktop Bridge. This new technology allowed developers to release Win32 applications packaged as AppX on the Microsoft Store, or to deploy them manually in a managed environment, like an enterprise.

In the end, the new packaging format isn't just a way to deploy applications in a better way, but also to help you move your applications forward. Packaged applications, in fact, get an **application identity**, which means that they can be univocally identified by Windows 10. This feature allows an application to leverage the APIs provided by the Universal Windows Platform and start using features like toast notifications, live tiles, and Windows Hello. In Chapter 6, we're going to see how, after we have packaged our classic desktop application using MSIX, we can start to add new features offered by the Windows 10 ecosystem.

The benefits of MSIX for IT pros

Deployment has always been a pain point not only for developers, but also for IT pros in charge of maintaining the application's catalog of enterprises with thousands of devices. When it comes to deploying an application within a company, in fact, IT pros can rarely afford to deploy it as is. In most cases, they need to customize the installer to accommodate the requirements of the company, such as specific configurations or branding. To satisfy this need in the past, Microsoft created technologies like App-V and Custom Tools for MSI, which allow you to take the installer provided by a developer and customize the deployment process.

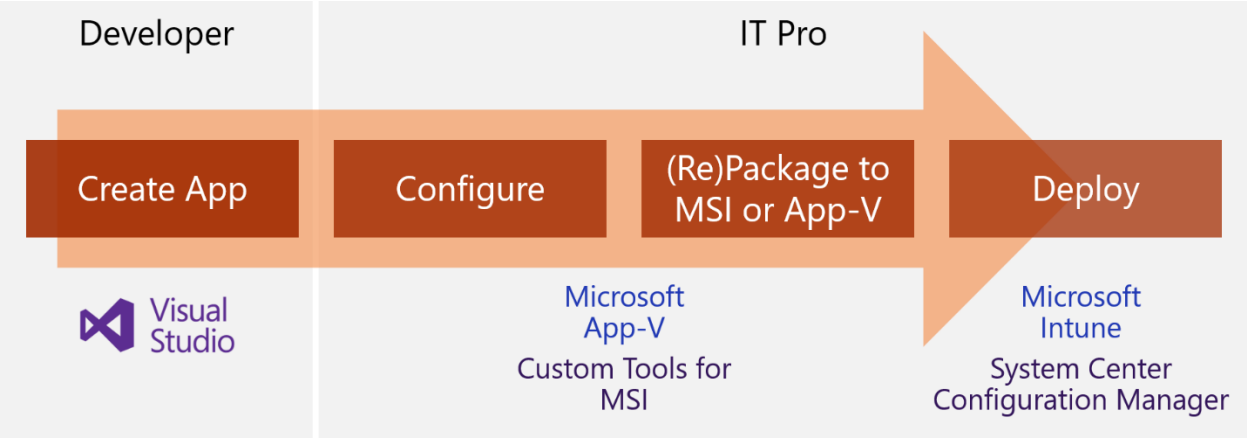


Figure 3: The typical application lifecycle today in the IT pro world, as described by John Vintzel in his session "MSIX: Inside and out" at Ignite 2018

This approach, however, has often created a packaging paralysis within companies. Since applications, customizations, and the operating system are bound together, every time a new update is released, the IT department is required to repackage the application from scratch. This increases costs and time and slows down the company's ability to release new updates to the employees.

The MSIX packaging format, instead, helps to separate the various aspects of the application deployment, allowing IT pros to independently deploy Windows updates, application updates, or customizations. This way, when a developer releases a new version of the application, or Microsoft publishes a new Windows update, it can be directly deployed without any repackaging process in between.

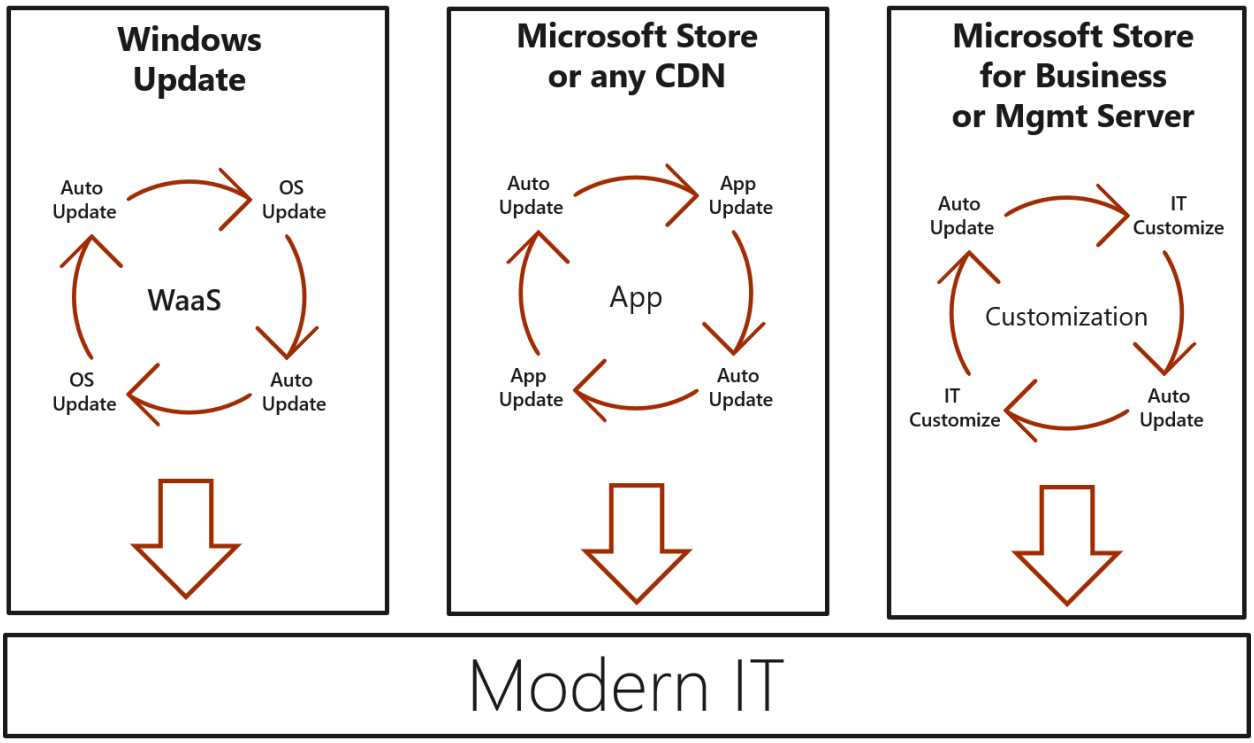


Figure 4: In the modern IT, all the components of the infrastructure should be able to update independently, as described by John Vintzel in his session "MSIX: Inside and out" at Ignite 2018

MSIX is enterprise-friendly. In addition to supporting new distribution platforms, like the Microsoft Store (including the private versions, the Microsoft Store for Business, and the Microsoft Store for Education), it works just fine with all the existing enterprise management tools, like System Center Configuration Manager and Microsoft Intune. It also supports manual distribution by publishing the package on a website or on a network share.

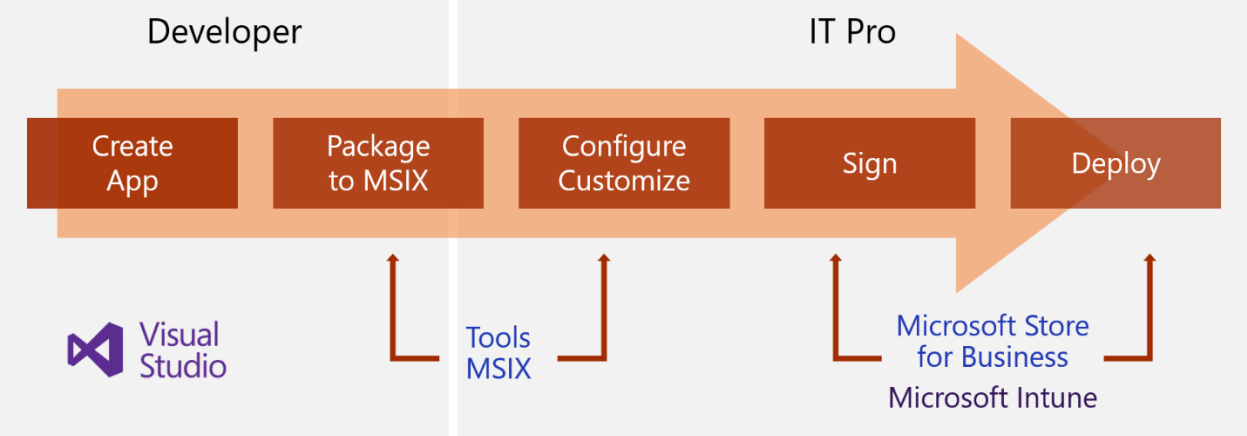


Figure 5: The modern application lifecycle in the IT Pro world, as described by John Vintzel in his session "MSIX: Inside and out" at Ignite 2018

The features of MSIX deployment

The goal of the MSIX packaging format is to become the deployment solution moving forward by helping to solve all the challenges that developers and IT pros have faced in the past with other distribution formats.

Let's look at the main benefits of adopting MSIX compared to other installer technologies.

Simple packaging

MSIX is a very simple packaging format. It's basically a compressed file that contains the application, plus a set of files that are required by Windows to handle the deployment. The most important file is the **manifest**, which is an XML file that describes the application and defines its identity. Thanks to the manifest, Windows can handle the deployment and understand if the application is already installed, or if the current package is an update.

Being declarative, it's very simple to integrate with Windows 10, since all the extension points (like supporting file-type association or startup tasks) are handled with a simple entry in the manifest. We're going to learn more about Windows 10 integration in Chapter 5.

Secure and reliable

MSIX provides a more reliable deployment approach since it's handled directly by Windows. As such, the operating system can keep track of all the files and registry keys that are created during the deployment phase and at runtime and perform a clean uninstall when the application is removed. This helps Windows be reliable and stable over time, since it reduces the number of unused registry keys and files that are often left in the system by traditional uninstallers.

MSIX is also safer, because packages must come from the Microsoft Store or from a trusted source, which means that they must be signed with a digital certificate that is trusted by the machine. In the case of an enterprise, for example, they can be signed with an internal certificate generated by the company. If a package isn't signed, or isn't signed with a trusted certificate, there's no way to deploy it on a machine for a regular user.

MSIX also has built-in anti-tamper technology. If the files that compose the application are changed by a third-party application or by manual intervention, Windows will prevent the execution. If the application has been downloaded from the Microsoft Store, Windows will repair it automatically. If it's coming from a different source, it will be up to the user or the IT department to reinstall it.

Flexible distribution

One of the original blockers in the adoption of the AppX packaging format by enterprises was the limited distribution opportunities. Only the Microsoft Store was supporting the distribution of these new packages. This approach works great for consumers, but enterprises need more control over the deployment.

Over time, the deployment opportunities for AppX have steady increased, and today, MSIX can be deployed with any technology currently used to distribute traditional installers, like System Center Configuration Manager, Microsoft Intune, or simply an internal network share or website. Additionally, thanks to a technology included in Windows 10 called **App Installer**, you can implement some of the benefits of store distribution, like auto updates, even with a more traditional deployment approach. We're going to learn more about this opportunity in Chapter 7.

OS managed and optimized

Windows oversees every step of the deployment, from the installation to the removal to the update process. As such, it's able to automatically optimize many of these processes, without any special requirement for the developer or the IT pro:

- **Incremental updates**: Developers don't need to handle the optimization of the update process. They can just create a new package with the whole content of the application, no matter what the size is. Windows, before downloading the update from any source (the Microsoft Store, Intune, a web location, etc.), will check the hash table of the files included in the original package. It will match them with the ones of the updated package. Windows will download only the files that have changed, helping to save bandwidth and disk space.
- **Single instance of storage files**: If you deploy multiple packages with the same content (for example, a framework or library), Windows will maintain a single copy on the disk and, with a linking mechanism, will provide the files to the application that needs them. This process is completely transparent to the applications, which will behave as if the files are physically included inside the deployment folder. However, files will be stored just once, helping to save disk space.
- **Resource packages**: MSIX packages can be split in multiple subpackages that contain resources related to a specific configuration, like images or translations. Windows will download only the relevant packages for the current configuration of the user.

The architecture of an MSIX package

An MSIX package is a simple file container for an application. Inside it, you will find all the files that make up your application, plus a set of special files and folders:

- **AppManifest.xml**: The manifest file, which describes the application. We're going to see more details about the manifest file later in this section.
- **VFS**: A special folder used to abstract the file system and provide virtualized access to system folders. We'll see more details later in this chapter.
- **Assets**: The folder that contains the default image assets leveraged by Windows 10. These are the icons used for the Start menu entry, the tile, and the icon in the taskbar.
- **A set of files with the .dat extension**: These files map the various registry hives that are part of the system registry, and they include the keys that must be created during the deployment phase.

When you deploy an MSIX package, its content is automatically uncompressed in a special, system-protected folder: **C:\Program Files\WindowsApps.** The name of the folder where the application will be stored depends on the **Package Full Name**, which is a unique identifier. It's a combination of the name of the publisher, the name of the app, the version number, the target CPU architecture, and the hash of the publisher.

For example, this is the path of the MSIX Packaging Tool, the tool by Microsoft to package applications as MSIX, which we're going to explore in Chapter 2.

C:\Program Files\WindowsApps\Microsoft.MsixPackagingTool_1.2019.402.0_x64__8wekyb3d8bbwe

The folder name is composed of the following elements:

- **Microsoft**: The publisher's name.
- **MsixPackagingTool**: The application's name.
- **1.2019.402.0**: The version number.
- **x64**: The CPU architecture for which the application is compiled.
- **8wekyb3d8bbwe**: The publisher's hash.

The most important file of the package is the manifest, named **AppxManifest.xml**. It's an XML file defining many important aspects of the application, like its identity, the leveraged capabilities, and the extensions. Visual Studio includes a visual editor for the manifest, which makes it easier to edit it. However, since we don't always have the opportunity to access the source code of the project, it's important to understand the XML structure.

Let's see how the manifest looks.

Code Listing 1

```
<?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"


xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"


xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"


xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows
10/restrictedcapabilities" >

  <Identity Name="NotepadPlusPlus" Publisher="CN=mpagani" Version="1.0.0.0"
ProcessorArchitecture="x64" />
```

```
  <Properties>
    <DisplayName>Notepad++</DisplayName>
    <PublisherDisplayName>Matteo Pagani</PublisherDisplayName>
    <Description>Reserved</Description>
    <Logo>Assets\StoreLogo.png</Logo>
  </Properties>
  <Resources>
    <Resource Language="en-us" />
  </Resources>
  <Dependencies>
    <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.17763.0"
MaxVersionTested="10.0.18362.0" />
  </Dependencies>
  <Capabilities>
    <rescap:Capability Name="runFullTrust" />
  </Capabilities>
  <Applications>
    <Application Id="Notepad" Executable="Notepad++\notepad++.exe"
EntryPoint="Windows.FullTrustApplication">
      <uap:VisualElements BackgroundColor="transparent"
DisplayName="Notepad++" Square150x150Logo="Assets\Square150x150Logo.png"
Square44x44Logo="Assets\Square44x44Logo.png" Description="Notepad++">
        <uap:DefaultTile Wide310x150Logo="Assets\Wide310x150Logo.png"
Square310x310Logo="Assets\Square310x310Logo.png"
Square71x71Logo="Assets\Square71x71Logo.png" />
      </uap:VisualElements>
      <Extensions>
        <uap:Extension Category="windows.fileTypeAssociation">
          <uap3:FileTypeAssociation Name="notepad">
            <uap:SupportedFileTypes>
              <uap:FileType>.txt</uap:FileType>
```

```
            <uap:FileType>.rtf</uap:FileType>
          </uap:SupportedFileTypes>
        </uap3:FileTypeAssociation>
      </uap:Extension>
    </Extensions>
  </Application>
 </Applications>
</Package>
```

Let's analyze the key elements.

The identity

The identity of the application is defined by the **`Identity`** tag, and it's composed of:

- A name.
- The identifier of the publisher of the application. It must match the subject of the digital certificate that will be used to sign the package.
- The version number.

The identity is used by Windows to handle the lifecycle of the package. For example, by matching the name and the publisher, it's able to understand whether a specific application is already installed in the system. Or, by using the version number, it's able to understand if Windows needs to perform an upgrade or a clean installation.

The **`Properties`** section contains some other important information that defines the identity of the application, like the name and the publisher that will be displayed to the user. This information is particularly important if you're going to publish the application on the Microsoft Store. The application's name, in fact, must match the name you reserved on the Microsoft Store. The publisher's name, instead, must be the same one you chose when you opened an account on the Microsoft Partner Center, the web portal used to submit applications on the Microsoft Store.

The capabilities

Capabilities are used by Windows to understand which features are leveraged by the application, and they act like a security gateway for the most sensitive ones. If an application doesn't declare a specific capability, all the APIs that try to invoke the related feature will return an error. Some examples of capabilities are internet access, camera access, and location access.

This capability model mostly applies to Universal Windows Platform applications and components. Classic desktop applications, since they are built with a wide range of technologies released before the advent of Windows 10, can't leverage such a tailored capability system, and they will simply be ignored.

However, there's a key capability that must be declared by every Win32 application, called **`runFullTrust`**, which allows them to keep leveraging the classic Windows model (no sandbox, direct access to the file system and the registry, etc.). It's a **restricted capability**. From an enterprise point of view, it doesn't require any special attention. The difference comes if you want to publish the application on the Store. In this case, you'll be asked the reason you need to use such a restricted capability. You can find more details in Chapter 7.

The application

The **`Application`** node defines the main features of the application itself. The most relevant attributes are **`Executable`** and **`EntryPoint`**, which define the main process to invoke when the user launches the application.

These elements of the manifest make another difference between a Universal Windows Platform application and a classic desktop application:

- In a Universal Windows Platform application, the **`EntryPoint`** points to the main class that handles the activation of the application. Typically, it's called **`App`**.
- In a classic desktop application, the **`EntryPoint`** attribute is set to the fixed keyword **`Windows.FullTrustApplication`**, while the **`Executable`** attribute points to the main executable.

The **`Application`** entry also has different sub-nodes for handling the visual elements of the application. They include a reference to the various assets used to display the application's entry in the Start menu, in the taskbar, or as a tile in the Start screen.

Extensions

The **`Extensions`** section is optional, and it will be included only for applications that integrate with Windows 10. Some examples of integration are file type association, supporting launch at startup, and defining a global alias. The most relevant extensions will be detailed in Chapter 5.

In Code Listing 1, you can see, for example, the required entry to set up file type association. Thanks to this entry, Windows will offer to users the opportunity to open the application automatically when they double-click on a file with the extension .txt or .rtf.

The container

Unlike Universal Windows Platform applications, which run inside a sandbox, classic desktop applications packaged as MSIX still retain full access to the operating system. However, to make the application safer and more reliable, MSIX provides a lightweight container that helps to virtualize some of the critical aspects of an application.

Let's see them in detail.

The virtual file system

An MSIX package contains a special folder called **VFS**, which can map most of the system folders that you find in Windows, like *C:\Windows*, *C:\Windows\System32*, *C:\Program Files*. When a packaged application needs to load a file from one of these folders, Windows will look first inside the VFS folder of the package, and if it doesn't find it, it will forward the request to the real system folder.

Here is a list of the most relevant supported folders, taken from the official documentation:

Table 1: System folders supported by the virtual file system

System location	Redirected location in the VFS folder
C:\Windows\System32	SystemX86
C:\Windows\SysWOW64	SystemX64
C:\Program Files (x86)	ProgramFilesX86
C:\Program Files	ProgramFilesX64
C:\Program Files (x86)\Common Files	ProgramFilesCommonX86
C:\Program Files\Common Files	ProgramFilesCommonX64
C:\Windows	Windows
C:\ProgramData	Common AppData

This feature was introduced to help package application-handling dependencies, like frameworks and libraries, in an easier way. Thanks to this approach, you can mitigate the problem known as "DLL hell," which happens when you have multiple applications on your system that depend on the same framework or runtime. At some point, you may have to install another application that has a dependency on a newer version of the same framework. Since this framework can be installed only at a system level, you are forced to overwrite the existing version, which can lead the other applications to break or not behave well.

Thanks to the virtual file system, you can embed inside every package the exact version of the framework required by the application. Since each package lives in its own folder, it won't try to replace other versions of the same framework you may have on your machine. The existence of the virtual file system is completely transparent for the application: it will continue to think that the libraries are part of the operating system. As such, no code changes are required to support this feature.

Additionally, thanks to the file-linking optimization supported by MSIX packages, even if multiple applications should include the same version of a framework, disk space will be optimized, because the files will be physically saved just once, and then linked in each package that requires it.

This feature is also helpful for providing a better deployment experience for the user. Thanks to this approach, users can install an application and use it right away, without having to manually download any required dependency. If you're interested in publishing your packaged application on the Microsoft Store, this is a strict requirement. An application that doesn't automatically satisfy all the dependencies and requires the user to manually download them from a website or another source, is in violation of the Microsoft Store policies.

Registry virtualization

Each packaged application has access to a virtualized version of the user hive in the system registry. This means that every time the application tries to create or update a registry key under the **HKEY_CURRENT_USER** hive, it will be stored in its private hive and not in the system registry. Read operations, instead, will be performed against a merged version of the registry. If the hive you're trying to read from doesn't exist in the local one, Windows will redirect the operation to the system one.

Keys that must be created in the system-wide hives, like **HKEY_LOCAL_MACHINE**, are stored, instead, in special files with .dat extensions, which are placed in the root of the packages. These keys will also be created in the virtual registry during the package deployment; the application will be able to read them, but they won't be visible in the real system registry. These files are typically created during the packaging process, using the MSIX Packaging Tool we're going to see in Chapter 2.

This virtualization helps to deliver a clean uninstallation process. When the application is removed, Windows will simply remove all the binary files that are used to store the registry keys, making sure that no orphan keys are left in the real system registry.

File system virtualization

Windows comes with a special folder called **AppData**, which lives under the user profile (*C:\Users\<username>\AppData*). This is the suggested location where applications should keep all the local application data: logs, databases, and, generally speaking, every file that has a tight relationship with your application. As such, it should be removed when it's uninstalled.

Like for the registry, Windows virtualizes this folder. Whenever your application tries to read or write files in the AppData folder, Windows will redirect the operation to the local storage of the application, which is a special folder visible only to it.

The local storage of each application is stored inside a folder with the following path:

C:\Users\<username>\AppData\Local\Packages\<Package Family Name>\LocalCache.

Using again the MSIX Packaging Tool as an example, this is the location where its AppData folder is redirected.

C:\Users\<username>\AppData\Local\Packages\Microsoft.MsixPackagingTool_8wekyb3d8bbwe\LocalCache

The folder name is composed of the following elements:

- **Microsoft** is the publisher's name.
- **MsixPackagingTool** is the application's name.
- **8wekyb3d8bbwe** is the publisher's hash.

The operation will be completely transparent for the application. Developers don't have to change the code to support this virtualization.

Like for the registry, this feature helps to deliver a clean uninstall. When the application is removed, Windows will delete the folder that contains the local storage of the application, making sure that there are no temporary files left in the system.

The requirements

MSIX is a great way to package your existing applications, but before starting the packaging process, it's important to know that you must satisfy a set of requirements. Some of them come from the technology being new and, as such, it still doesn't support all the scenarios. Some others are a consequence of the lightweight container we have talked about, which is transparent for most of the scenarios, but might be a blocker in some edge cases. It's important to highlight that not all the requirements listed here are complete blockers. Thanks to tools like the Package Support Framework, which will be described in detail in Chapter 4, you can overcome some of these limitations, even if you don't have access to the source code.

Let's see the most relevant ones.

.NET Framework version

If your application is built with the .NET Framework, you must target at minimum version 4.0. The reason is that MSIX doesn't support including older versions of the .NET Framework inside the package and, as such, you must rely on the version which ships with Windows 10, which is greater than 4.x.

Unless you have any special requirement, the suggested minimum version is .NET Framework 4.6.2.

Windows services and drivers

The MSIX packaging format doesn't currently support the deployment of NT services or custom kernel drivers. This is a blocker if your goal is to publish the application on the Microsoft Store, since you are not allowed to ask to the user to manually download a component that needs to be installed separately.

There are no restrictions when it comes to enterprise deployments. You can safely deploy the application as MSIX and then, using another technology, deploy the required driver or service.

The suggested path to deploy drivers is to certify and publish them on Windows Update.

Elevated security privileges

Applications packaged as MSIX are installed per user, and they are always expected to run in user mode, since users who install the application might not be system administrators. This is a strict requirement when the application is published to the Microsoft Store. For enterprise deployments, however, you can request administrative credentials if needed, even if it's not suggested.

Windows 10 version 1809 has added a new special capability that can be declared in the manifest file to support elevation at startup. However, it's a restricted capability, and third-party applications published on the Microsoft Store won't be allowed to use it. You can use it, instead, for applications that are deployed in an enterprise environment.

The restricted capability is called **`allowElevation`**.

Code Listing 2

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

<Package
xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows
10/restrictedcapabilities"

  IgnorableNamespaces="rescap">

   . . .  <f:Capabilities>

    <rescap:Capability Name="runFullTrust" />

    <rescap:Capability Name="allowElevation" />

  </f:Capabilities>

. . .</f:Package>
```

Sharing data using the registry or the AppData folder

We have learned that a Windows application packaged as MSIX uses a virtualized version of the registry and of the AppData folder, which are private and visible only to the application itself. Therefore, your application can't use one of these two locations to share data with other applications. Since all the registry keys and all the files in the AppData folder are visible only to the application itself, other applications can't access them.

Writing to the installation folder

As already mentioned, during deployment, MSIX packages are uncompressed in a special folder (*C:\Program Files\WindowsApps),* which is system-protected and read-only. As such, if the application tries to write or update a file stored in the installation folder (for example, a log file), the operation will fail because it doesn't have enough rights to perform the task.

This limitation can be overcome by changing the code to write the files in another, more suitable, location (like the AppData folder), or by using the Package Support Framework. We'll see the details on this in Chapter 4.

Access to the Current Working Directory

If writing files to the installation folder isn't allowed, you can safely perform reading operations. However, many applications use the concept of the current working directory (CWD) to perform this task, in which the path is returned by the shortcut created during the installation process. It typically matches with the folder where the application has been installed.

Since the deployment of a MSIX package doesn't generate a traditional shortcut, the Windows APIs used to retrieve the current working directory will return a different path: *C:\Windows\System32* if the application is compiled for a 32-bit architecture; or *C:\Windows\SysWOW64* if the application is compiled for a 64-bit architecture.

As such, if your application tries to read a file inside the package using the current working directory, the operation will fail because it will look for the file in the wrong folder.

As with writing to the installation folder blocker, this limitation can be overcome by changing the code or by leveraging the Package Support Framework.

Down-level support and MSIX Core

One of the challenges you encounter when you want to introduce a new packaging and deployment system is providing support to all the platforms used by your customers. MSIX was introduced in Windows 1809; however, many enterprises are still on previous versions of Windows 10, since they need more time before adopting a newer version. To address this scenario, Microsoft has added down-level support for MSIX with a series of Windows 10 updates. MSIX packages can now be deployed on Windows 10, versions 1803 and 1709. This feature is focused mainly on enterprises and, as such, the support covers deployment using sideloading, SSCM (Microsoft System Center Configuration Manager, formerly Systems Management Server), and Intune. To download, instead, MSIX-packaged applications from the Microsoft Store, you will still need at least Windows 10, version 1809.

What about Windows 7? The end of support is approaching soon (January 2020), but there are still many enterprises that haven't yet fully migrated to Windows 10. For this scenario, Microsoft is working on a project called **MSIX Core**, which is open source and currently on preview in GitHub. It's a tool that you can install on Windows 7 (or any other Windows version that doesn't support MSIX) to enable support for the new packaging format.

After installing the tool, you'll be able to double-click an MSIX file and trigger the installation of the application. All the included files will be copied over to your hard disk, all the files stored inside the virtual file system will be copied in the system folders, and a link to the application will be added to the Start menu. In the end, an entry will be included in the Add/Remove Programs panel to allow uninstallation.

MSIX Core isn't able to bring many of the MSIX features, like the file/registry virtualization or the optimized update experience. All these features are deeply rooted inside the Windows 10 kernel, and they would require significant updates to Windows 7 itself. However, thanks to MSIX Core, you won't have to maintain two different distribution systems based on the OS you're targeting, you'll be able to use MSIX regardless of the customer's operating system.

Chapter 2 Packaging Your Applications with the MSIX Packaging Tool

In Chapter 1, we learned all the basic concepts of the MSIX format, what the advantages are, and which things to keep in mind. We are ready to start packaging our first Windows application. In this chapter, we're going to explore this opportunity with a tool called MSIX Packaging Tool, which can convert an existing application even without owning the source code.

Packaging an installer: the MSIX Packaging Tool

Microsoft has released a new tool, with a special focus on IT pros who need to repackage their existing applications using the new format. The main feature of this tool is that it doesn't require the source code of the application. You can start from an installer, from a self-executable application, or from an App-V package.

It's called MSIX Packaging Tool, and it's available on the Microsoft Store. The only requirement is to use, at minimum, Windows 10 version 1809. It's also available on the Microsoft Store for Business, which will also give you the chance to download it for offline distribution.

How does the tool work? The key is the monitoring process, which is handled by a special driver that comes preinstalled with Windows 10. You must provide the tool with the installer of the application you want to repackage, which will be launched during the process. The tool will monitor all the changes performed by the installer to the operating system, like the deployment of files and the creation of registry keys. All these changes will be saved and then included inside the final MSIX package generated by the tool. The tool is smart enough to automatically handle all the special cases. For example, if the installer copies one or more files inside one of the system folders, the tool will automatically copy them in the proper subfolder of the VFS location.

Let's see a step-by-step conversion, so that we can better understand how the tool works. Since the tool doesn't require the source code of the application, we can use any installer that respects the requirements we have previously listed. For this sample conversion, we're going to use a very popular open-source application: Notepad++, the text editor. You can download the installer from the official website.

Once you have started the MSIX Packaging Tool from the Start menu, you will have to choose between creating an application package or a modification package. We're going to talk about modification packages in Chapter 3, so let's focus on an application package for the moment, which is the option to package a regular Windows application.

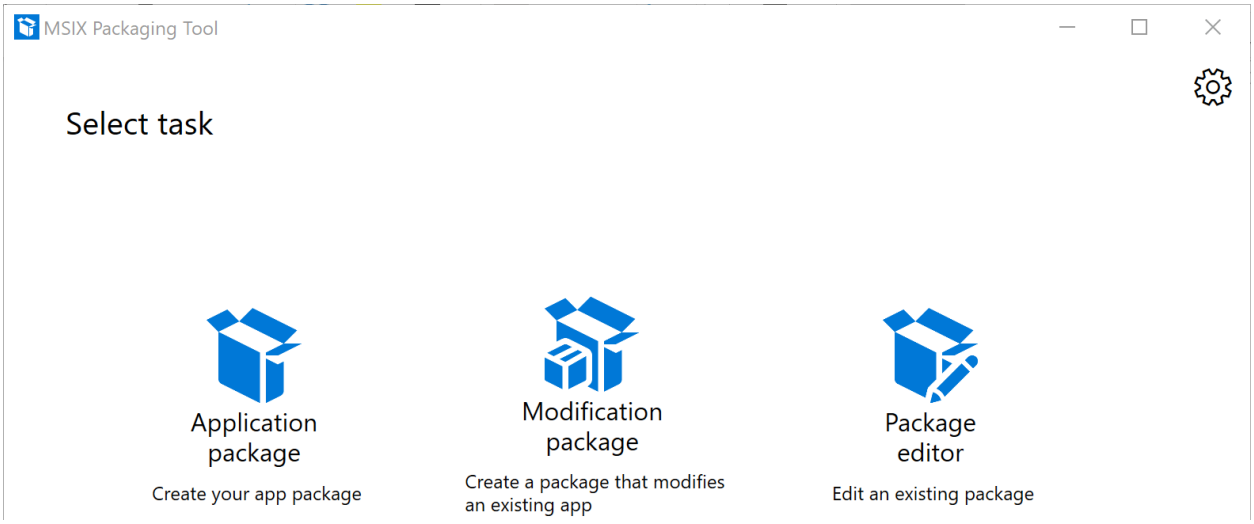


Figure 6: The main screen of the MSIX Packaging Tool

Let's see the process step by step.

Select installer

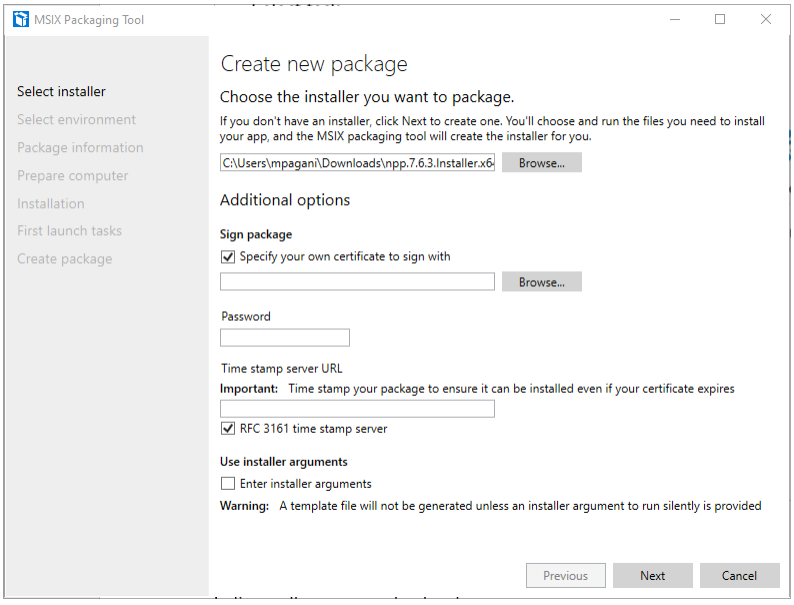


Figure 7: The screen of the "Select installer" step

The first step is to choose the installer of the application you want to package from your hard drive. If you don't have an installer (for example, because you want to package a self-executable application), you can skip this step. The tool supports any kind of installer, including App-V packages.

The next option is dedicated to the certificate to use for signing the package. As already mentioned in Chapter 1, this is a strict security requirement. To deploy an MSIX package on a machine, it must be signed with a valid certificate trusted by the machine. The only exception is if your goal is to deploy the application on the Microsoft Store. In this case, applications are automatically signed with a trusted certificate issued by Microsoft during the certification phase, removing the requirement of signing the package by yourself. In all the other scenarios, you must provide a valid certificate file from your hard disk and, optionally, its password. You can also specify a time stamp server URL, allowing the package to be installed even if the certificate expires. If you don't have a certificate, you can generate a self-signing one for testing. It isn't the best choice for enterprise or public distribution, but it's perfectly fine for testing purposes. You're going to learn more about certificates in Chapter 7.

The last option can be used to define additional installer arguments that will be passed when the setup is launched. Arguments can be used to customize the installation process, such as triggering a silent installation.

Once you have defined all the options, click **Next** to proceed to the next step.

Select environment

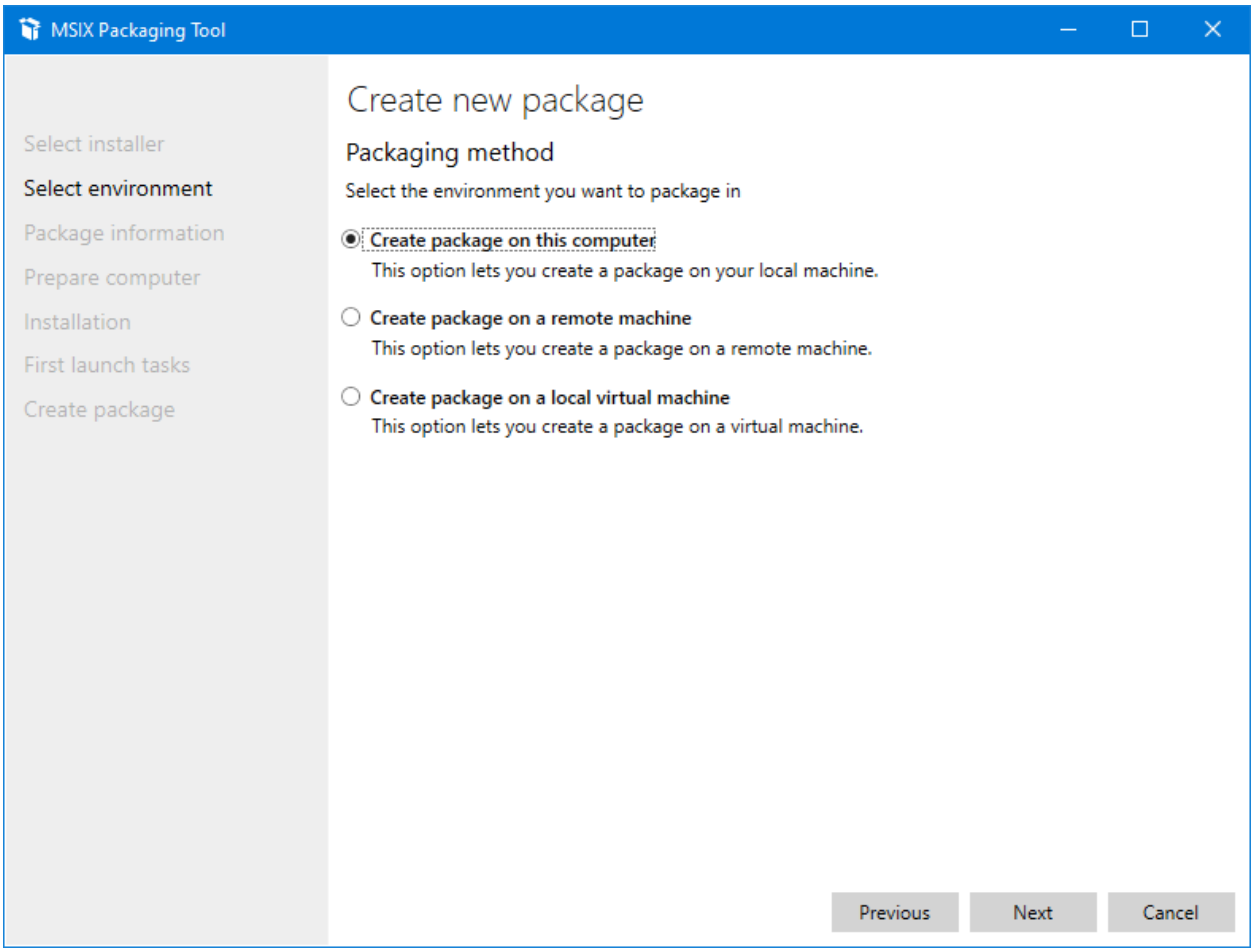


Figure 8: The screen of the "Select environment" step

In this step, you can choose if you want to repackage the application on your local machine, in an existing virtual machine, or on a remote machine. If you choose the second option, you will be able to specify the address and the credentials to log in to the computer. If you instead choose the third option, a drop-down menu will allow you to select one of the existing virtual machines configured in your local Hyper-V setup.

Note: Hyper-V is the Microsoft virtualization platform that, starting from Windows 8, has also been brought to the Pro and Enterprise editions of Windows. In the past, it was available only on the Server editions. You can enable it by searching in the Start menu for the "Turn Windows features on or off" section in the Control Panel and looking for Hyper-V in the list.

The suggested approach is to create a dedicated virtual machine with a clean Windows 10 installation, without any application or service installed other than the default ones. The main reason is that the monitoring process will capture all the operations happening on your machine while the installer is running. In a normal Windows computer, there may be lot of background activities: Windows services, minimized applications, updates, and so on. All these tasks can lead the package to contain many false positives, like files and registry keys that haven't been generated by the installer itself. A clean Windows 10 virtual machine can help you to reduce this eventuality.

Another reason is that, when the MSIX Packaging Tool installs the application, it's effectively performing the installation on the machine. Reverting the operating system to the state before the installation is a much easier task on a virtual machine, thanks to checkpoints. You can create a checkpoint before starting the packaging process, and then restore it once the process is ended to return your Windows 10 installation to a clean state.

Package information

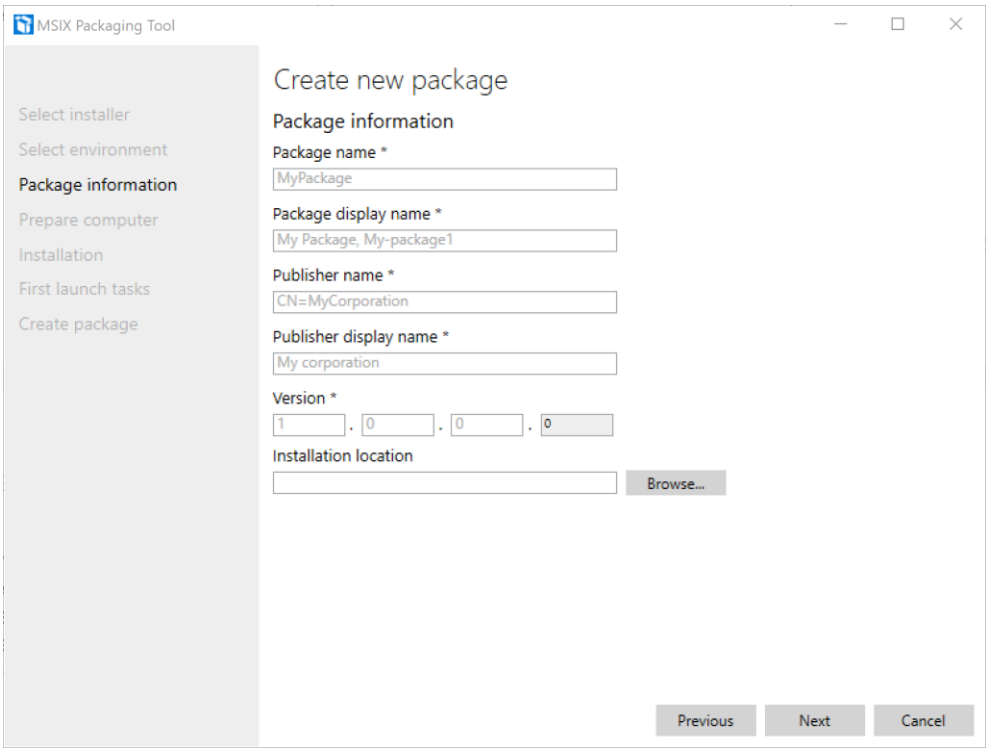


Figure 9: The screen of the "Package information" step

In this step, you can configure the identity of the application. All this information will be included in the manifest file of the package. By default, the tool will try to extract the information from the definition of the original installer. If it fails (like in the previous image), it will be up to you to manually fill all the fields. Let's see them in detail:

- **Package name**: The identifier of the application. If you're packaging an application for enterprise distribution, it's a free value. If you're planning to release the application on the Microsoft Store, it must match the identifier that the store has assigned to your application when you reserved the name for publishing.
- **Package display name**: The name of the application that will be displayed to the user in the Start screen. It's a free value if the package is meant for enterprise distribution; if it will be published on the Store, it must match the name you have reserved.
- **Publisher name**: This value will already be filled and read-only if you specified in the first step a certificate to use for signing the package. The requirement is that the subject of the certificate must match the publisher name. If you're planning to publish the application on the Store, this field must match the value assigned to your developer account when you registered it.
- **Publisher display name**: The name of the publisher, which will be visible to the user. The rules are the same for the package display name. If it's an enterprise application, it's a free value; if it's an application for the Store, it must match the publisher's name you chose when you opened your developer account.
- **Version:** The version number of the application. There isn't any rule here, except that the last digit must always be zero and, as such, it won't be editable. If you're creating an update, remember that this value must be higher than the version of the previous package you generated.

The last field, called **Installation location,** is dedicated to applications that don't have an installer or that require some manual extra steps after the installation process. All the files you're going to copy or modify inside this folder will be captured as part of the installation phase and included in the final package. This field is particularly important if you did not specify any installer in the first step. You can package a self-executable application by simply copying into this folder all the files that comprise it.

Click **Next** to move on to the next step.

Prepare computer

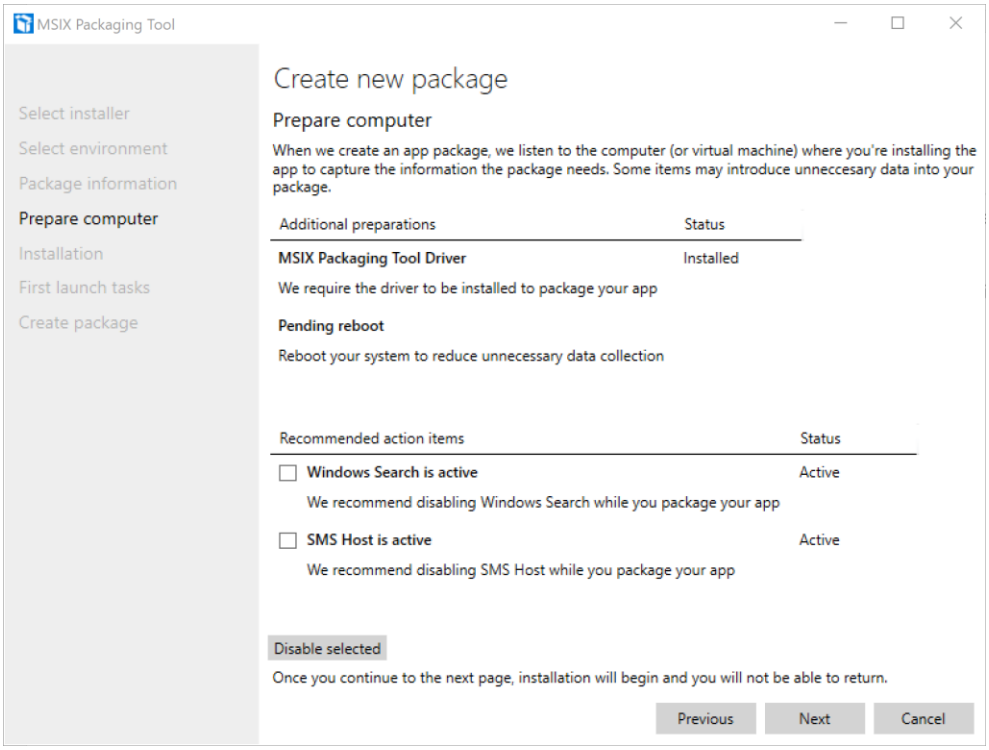


Figure 10: The "Prepare computer" step of the process

This is the last step before starting the real monitoring process, and it will make sure your machine is ready for it. This page will check for a series of factors that could introduce unnecessary data in your package, and it will propose some workarounds.

The only blocker is the **MSIX Packaging Tool Driver**. This special driver, which comes with Windows 10 version 1809 and later, is required to perform the installation monitoring. Without it, the tool can't work. If it isn't installed, the tool will take care of downloading and installing it from Windows Update for you.

If you are in an environment where Windows Update is not accessible, you can download the CAB, which contains the driver from this website (check the section titled, "Installing MSIX Packaging Tool driver FOD in WSUS"). Once you have downloaded it, you must open a PowerShell prompt with administrator rights and run the following command, where **`(path)`** is the full path of the CAB file you have downloaded.

Code Listing 3

```
Dism /Online /add-package /packagepath:(path)
```

The **Additional preparations** section may contain some other suggestions to improve the quality of the capturing process. For example, as you can see in Figure 4, if there's any update that has triggered a pending reboot, you will be invited to do it before moving on.

The second section, called **Recommended action items,** will highlight a series of services that may be running on your machine, like Windows Search, and it will give you the option to stop them in order to reduce the potential false data collection.

That's it. Now we can start the process by clicking **Next**.

Installation

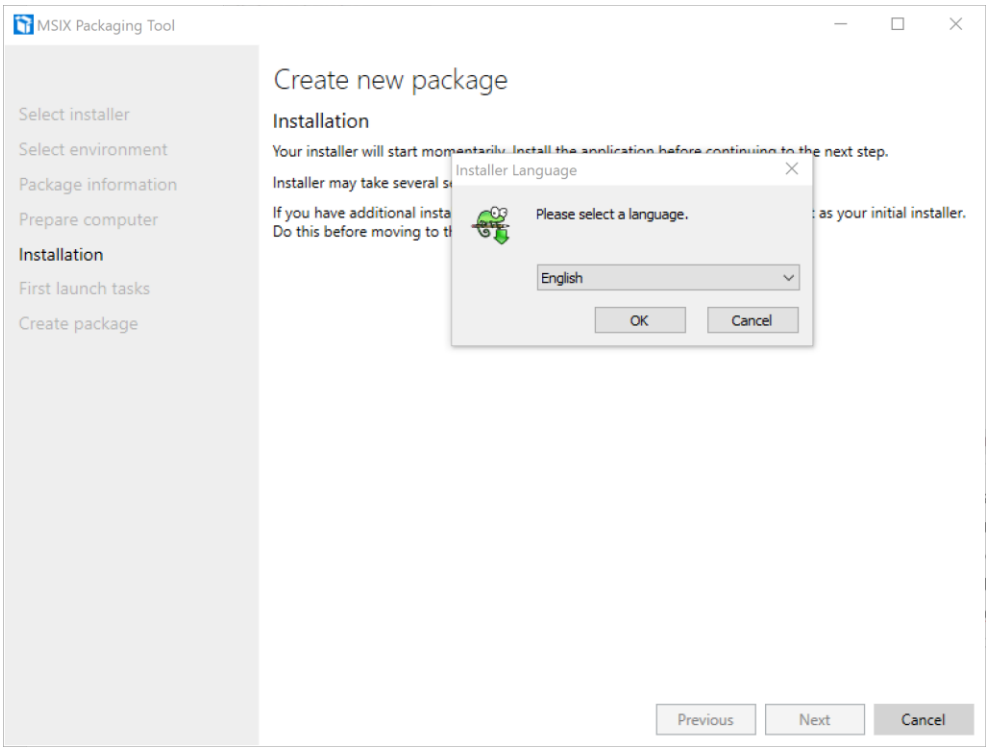


Figure 11: The "Installation" step of the process

The tool will now launch the installer you specified in the first step, and it will start the monitoring process. Now you just install the application, like you would do on a normal machine.

At the end of the setup, the tool will detect that the installation is completed and ask you if you want to terminate the process.

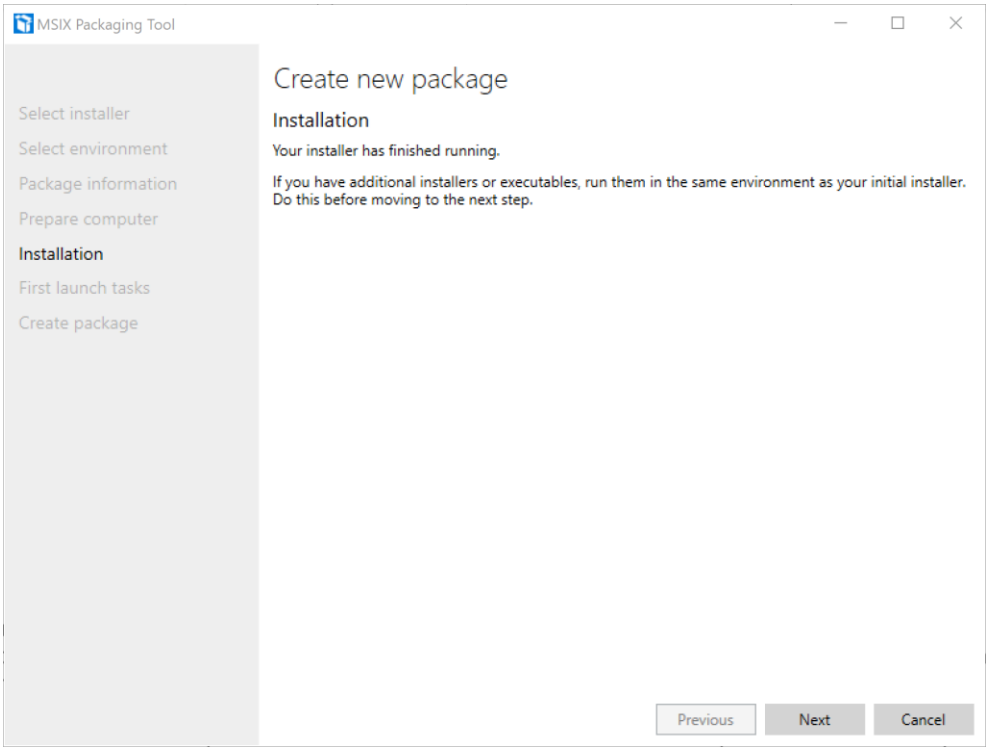


Figure 12: The tool has detected that the setup is completed

However, the monitoring will still be active. At this point, you have the chance to launch additional setups, like a dependent framework or another application. These files will be captured, too, and included in the same package. This feature helps you achieve the goal of building a self-contained package, which doesn't require the user to manually install any other dependency to use the packaged application.

First launch tasks

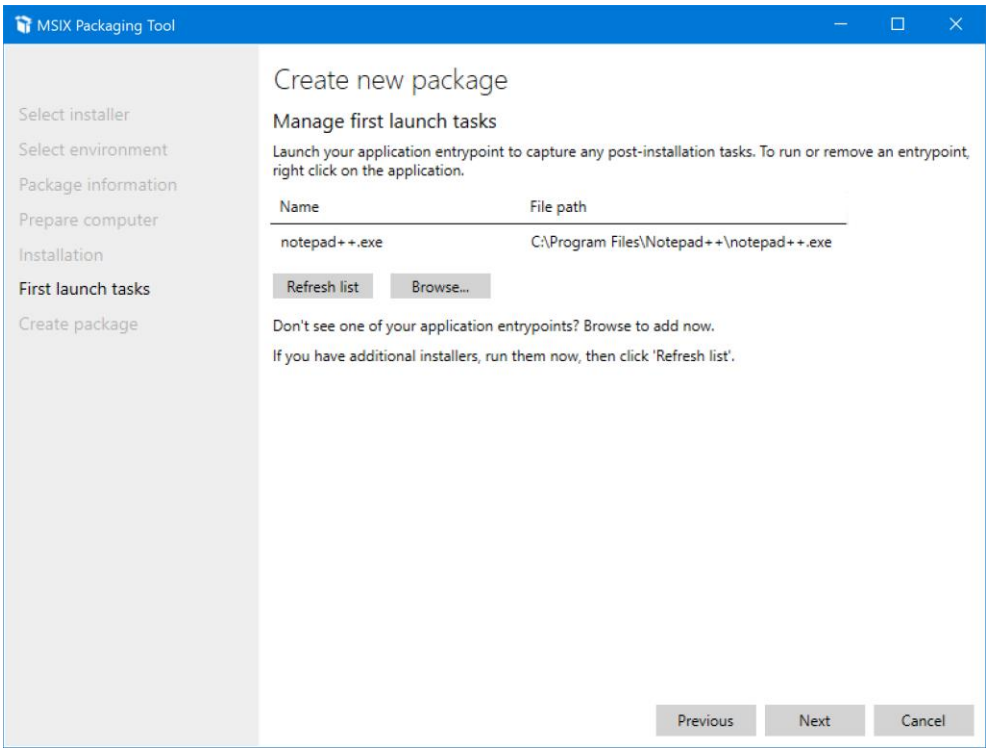


Figure 13: The first launch tasks

In this step, the tool will detect the main executable of the application you have packaged, based on the shortcut that the installer has created in the Start menu or on your desktop. If the tool can't detect it (for example, if you're packaging an application without an installer), you can manually choose it by clicking **Browse**.

The purpose of this step is also to help you capture any potential post-installation tasks of the application. The monitoring process, in fact, is still running. You're invited to launch the application you have just installed so that you can capture any additional configuration that might be performed at the first start. For example, there are applications that download some additional data the first time they are launched. Thanks to this option, you can include these files directly inside the package, making the deployment phase of the application smoother.

Once you have finished, click **Next.**

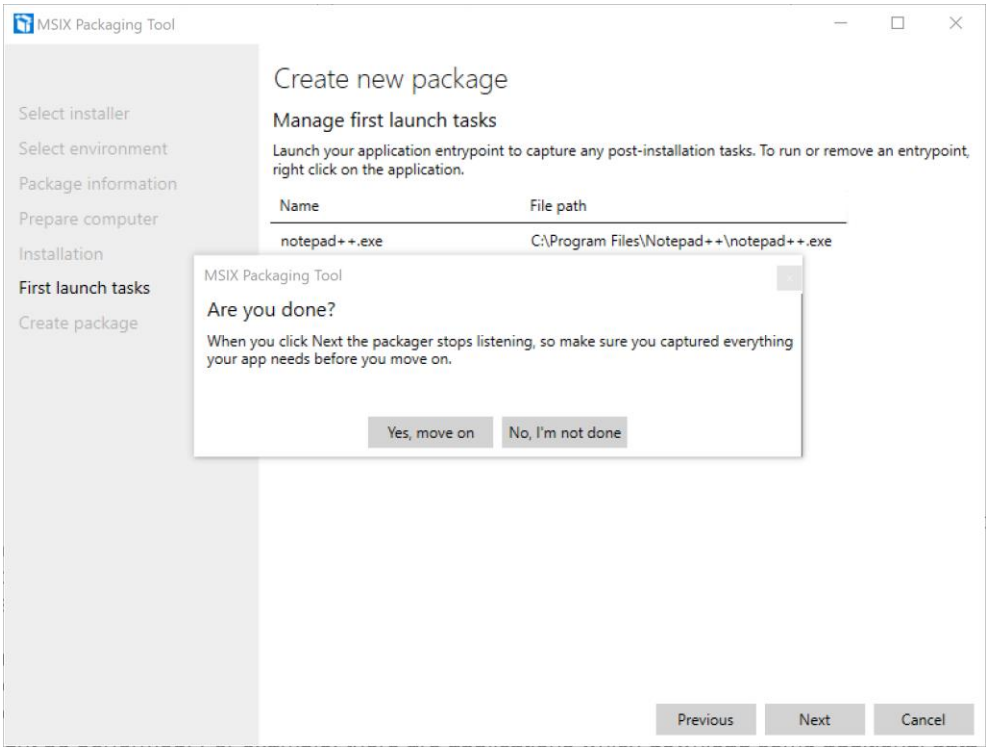


Figure 14: The tool is about to stop the monitoring process

This time, before moving to the next step, you'll be asked to confirm. The reason is that the tool will stop the monitoring process, and there's no way to resume it other than starting from scratch with a new packaging process.

Create package

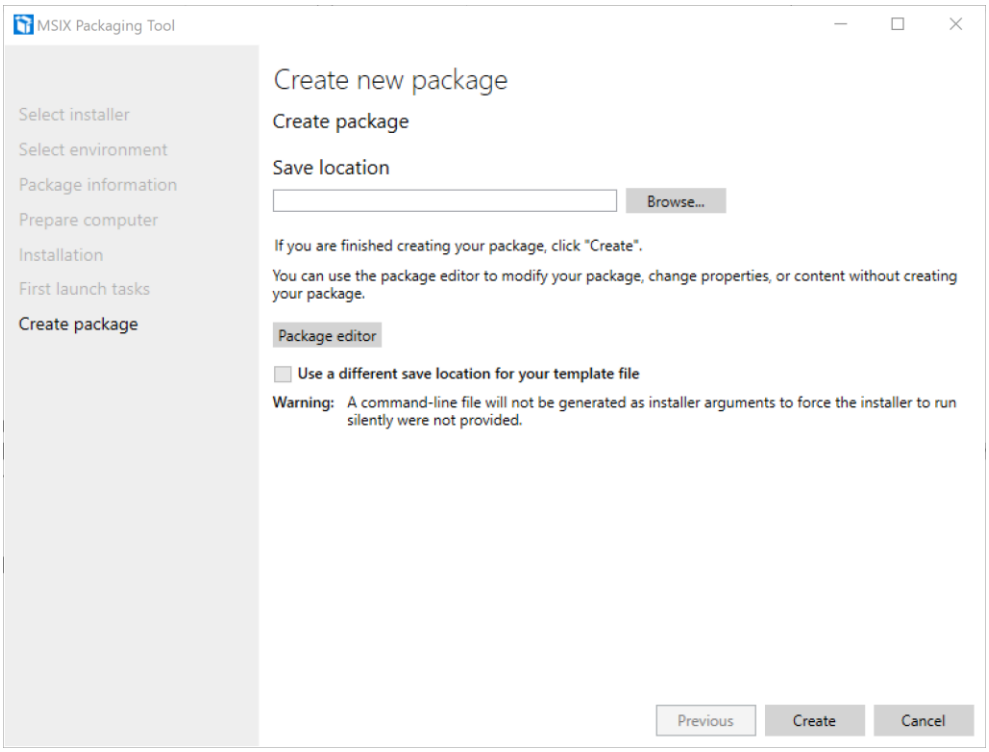


Figure 15: The last step of the packaging process

This is the last step of the process, which will give us the opportunity to choose the folder in which to save the package we have just created. Another option is to use the **Package editor** to inspect the content of the package to make sure that everything is in the right place. We'll explore the Package editor in more detail in another section of this chapter.

Click **Create** to complete the process. After a pop-up window shows a recap of the packaging process, you'll be redirected to the main screen of the application.

Testing the package

Before manually deploying an MSIX package, you will first to turn on the support for sideloaded apps, which allows Windows to install packages that are signed with any trusted certificate, and not just the ones coming from a public certification authority.

To do this, open the Windows settings and go to **Update & Security** > **For developers**. Select **Sideload apps** or **Developer mode** and wait for Windows to finalize the operation. The second option is suggested in case you want to execute more advanced tasks we're going to see in the next chapters, like deploying packages from Visual Studio or launching executables inside the application's container.

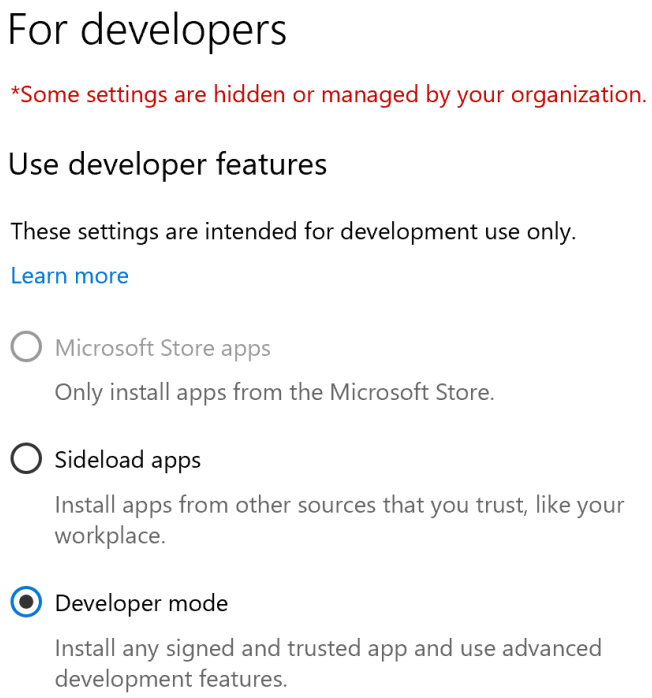


Figure 16: The Windows setting to enable the developer mode

The second step is to make sure that the certificate used to sign the package is trusted on your machine. The easiest way to check it is to right-click on the MSIX package in File Explorer, choose **Properties**, and move to the **Digital Signatures** tab. You will find a reference to the certificate used to sign the package. Click **Details** to see the information about the certificate.

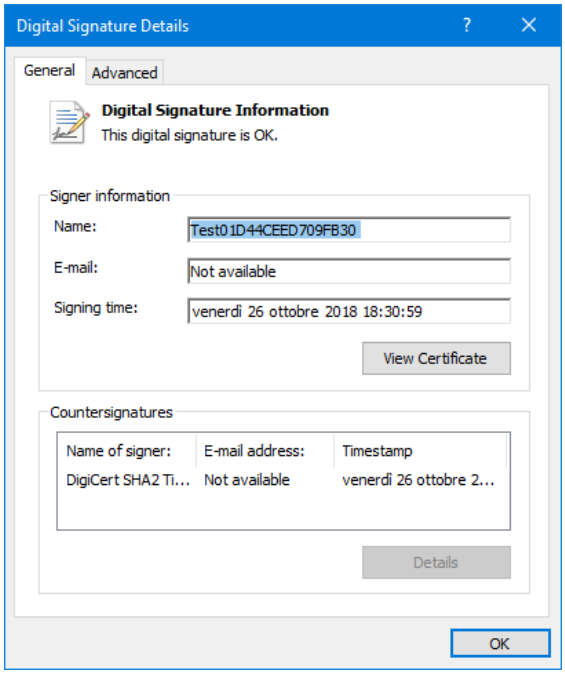


Figure 17: The details of the certificate used to sign the package

In Figure 17, you can see an example of a certificate that is already trusted on the machine. In such a case, you don't have to perform any extra step.

Otherwise, you need to install the certificate. Click **View Certificate**, then click **Install certificate** to start the wizard:

1. In the first step, choose to install the certificate for the **Local machine**. This will trigger the request of admin elevation when you'll press **Next**.
2. Choose the option **Place all certificates in the following store**. Then click **Browse** and, in the list of certificate stores, choose **Trusted People**. Click **OK** and then **Next**.
3. The last step is just a confirmation of the operation. Click **Finish** to complete the process.

Now that the certificate is installed, you can proceed to install the application. Windows 10 offers native support to MSIX packages, so you can just double-click on the file you have generated with the MSIX Packaging Tool.

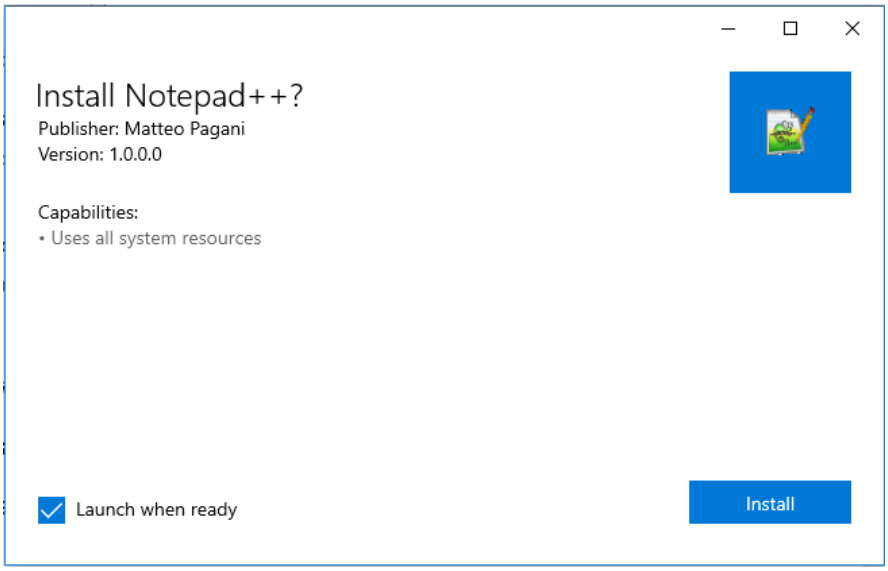


Figure 18: The Windows screen to manually deploy an MSIX package

If you click **Install**, the package will be deployed on the local machine, and if you leave the **Launch when ready** option turned on, the application will be executed at the end of the process. If the package isn't signed or the certificate isn't trusted, you will see an error appear inside the window.

Once the application has been deployed, you will find its shortcut in the Start menu and you'll be able to launch it by simply clicking on its icon. If the conversion has completed successfully and you are respecting all the requirements, you will see your original Windows application starting up.

Automating the packaging process

There are scenarios where you may need to automate the packaging process**.** In such a scenario, we can't leverage a user interface—all the operations must be executed without user intervention. For this purpose, the MSIX Packaging Tool can be leveraged from the command line, so that we can automate the full process we have previously seen.

The automation is handled with a **template file**, which is an XML file that describes the overall process: the installer's path, the location to store the output package, a set of files and registry keys to eventually exclude, and so on. The starting point for generating a template file is to perform a first conversion with the user interface of the MSIX Packaging Tool, like we previously did. At the end of the process, the tool will generate a template file that we can reuse.

However, the command-line packaging process is allowed only if the installer supports a silent installation, without any user intervention. Since the process is automated, we can't interact with the installer. If the starting point is an MSI installer, you're good to go, since it offers built-in support to silent installation, and the tool can automatically leverage it. If, instead, the application leverages another installer technology, you must provide a command argument to trigger the silent installation; otherwise the template file won't be generated. Let's take a look again at the first step of the packaging process with the Notepad++ installer.

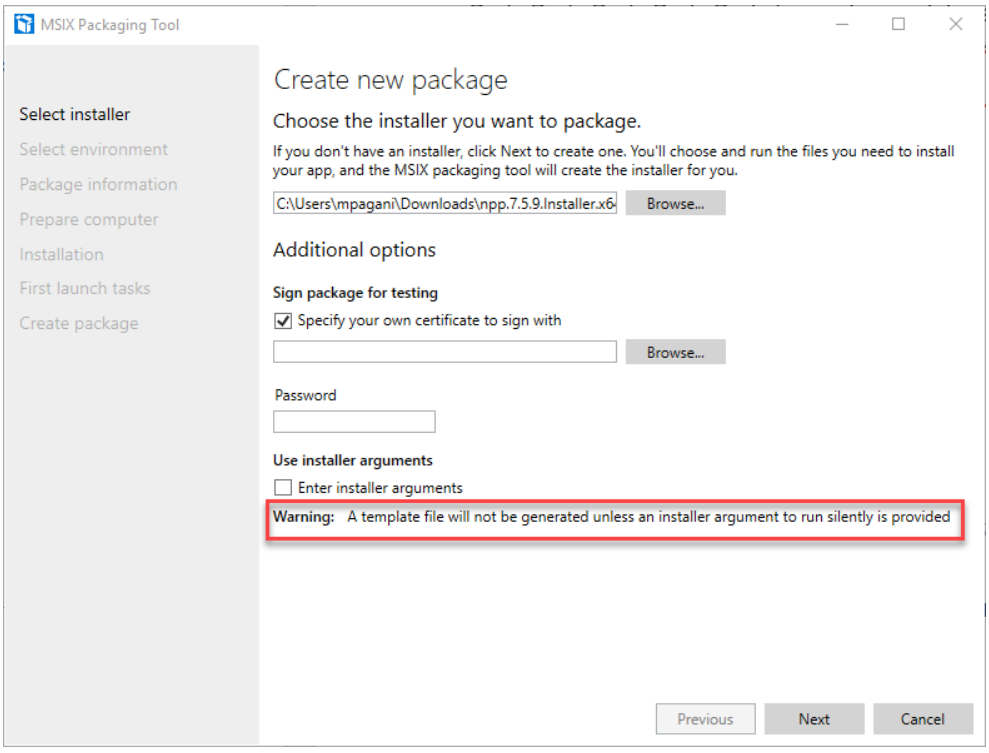


Figure 19: The MSIX Packaging Tool warns us that a template file can't be generated

As you can see from the image, the MSIX Packaging Tool has recognized that the Notepad++ installer isn't MSI-based and doesn't support, by default, a silent installation. As such, it's warning us that, unless we provide an installer argument to execute it in a silent way, a template file won't be generated at the end of the process.

There isn't a fixed rule for supporting a silent installation. Some of the most commonly used arguments are **/S**, **/SILENT**, and **/QUIET.** In the case of Notepad++, the installer argument to use is **/S**. If you specify it and you repeat the packaging process as we did before, you'll end up with another file inside the output folder: the template. Let's take a look at its content.

Code Listing 4

```
<?xml version="1.0" encoding="Windows-1252"?>

<MsixPackagingToolTemplate
xmlns="http://schemas.microsoft.com/appx/msixpackagingtool/template/2018">

  <Settings AllowTelemetry="true" ApplyAllPrepareComputerFixes="false"
GenerateCommandLineFile="true" AllowPromptForPassword="false"
EnforceMicrosoftStoreVersioningRequirements="true">

    <ExclusionItems>

      <FileExclusion ExcludePath="[{Local AppData}]\Temp" />

      . . .

      <RegistryExclusion
ExcludePath="REGISTRY\MACHINE\SOFTWARE\Wow6432Node\Microsoft\Cryptography"
/>

      . . .

    </ExclusionItems>

  </Settings>

  <PrepareComputer DisableWindowsSearchService="false"
DisableSmsHostService="false" DisableWindowsUpdateService="false" />

  <SaveLocation PackagePath="C:\
\Notepad++\NotepadPlusPlus_1.0.0.0_x64__e8f4dqfvn1be6.msix"
TemplatePath="C:\
\Notepad++\NotepadPlusPlus_1.0.0.0_x64__e8f4dqfvn1be6_template.xml" />

  <Installer Path="C:\Downloads\npp.7.5.9.Installer.x64.exe"
Arguments="/S" />

  <PackageInformation PackageName="NotepadPlusPlus"
PackageDisplayName="Notepad++" PublisherName="CN=mpagani"
PublisherDisplayName="Matteo Pagani" Version="1.0.0.0" />

</MsixPackagingToolTemplate>
```

The template is composed of different entries that define in a programmatic way all the steps we have previously performed with the user interface:

- The **`PrepareComputer`** entry defines which Windows services must be disabled or stopped before performing the packaging process.
- The **`SaveLocation`** entry defines where to save the output package and template.
- The **`Installer`** entry defines the path of the installer to the package and, eventually, the arguments. In this case, since the original installer isn't an MSIX, we have supplied the ***/S*** argument.
- The **`PackageInformation`** entry defines the identity of the application, and it includes the information that will be stored inside of the manifest.

The template also includes an **`ExclusionItems`** section, which contains the list of elements that could be captured during the packaging process, but that must be excluded because they aren't coming from the installer. You can have multiple **`FileExclusion`** entries for files and **`RegistryExclusion`** entries for registry keys. The standard template already includes a default set of exclusions, based on the most common false positives that Windows could generate during the process. However, it can be customized with additional ones.

Once you have a template file, you can use it with the command-line version of the MSIX Packaging Tool. To use it, you need to start a command prompt with administrator rights. The easiest way to do this is to right-click the **Start** menu and choose **Windows PowerShell (Admin).** Now you can launch the following command.

Code Listing 5

```
msixpackagingtool create-package --template "C:\
Notepad++\NotepadPlusPlus_1.0.0.0_x64__e8f4dqfvn1be6_template.xml"
```

When you install the MSIX Packaging Tool from the Microsoft Store, it registers a global alias called **`msixpackagingtool`**. As such, you can invoke it from any location without having to open a specific folder. The tool is very simple to use. You just need to specify the **`create-package`** command and the **`--template`** parameter, followed by the path of the template file you previously generated.

The tool will start, and it will execute the same exact packaging process we previously performed with the visual interface, but autonomously and without any user intervention. Once the process is finished, if no errors have happened, you will find in the output location an updated MSIX package and template file.

The package editor

The MSIX Packaging Tool includes a visual package editor, enabling you to modify the content of the package with a simple user interface. The editor can be leveraged as part of the packaging process, or independently.

In the first case, you will have access to the **package editor** option in the last step of the process. Thanks to this button, you will be able to immediately edit the output of the packaging process before saving the generated package. In the second case, instead, you can access the editor by choosing the **package editor** option in the main screen of the MSIX Packaging Tool. In this case, you will be asked to choose the package you want to edit.

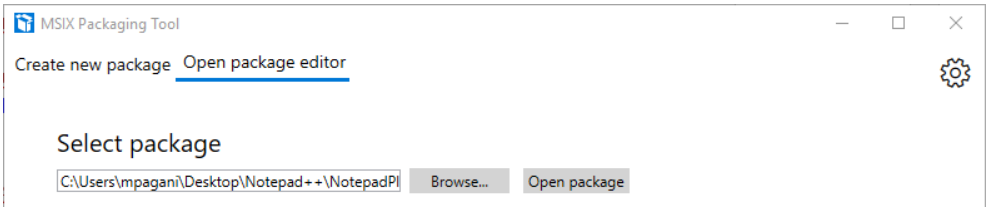


Figure 20: The option to access to the package editor in the MSIX Packaging Tool

Regardless of the choice, the tool will give you access to the same set of options. Let's see them in detail.

Package information

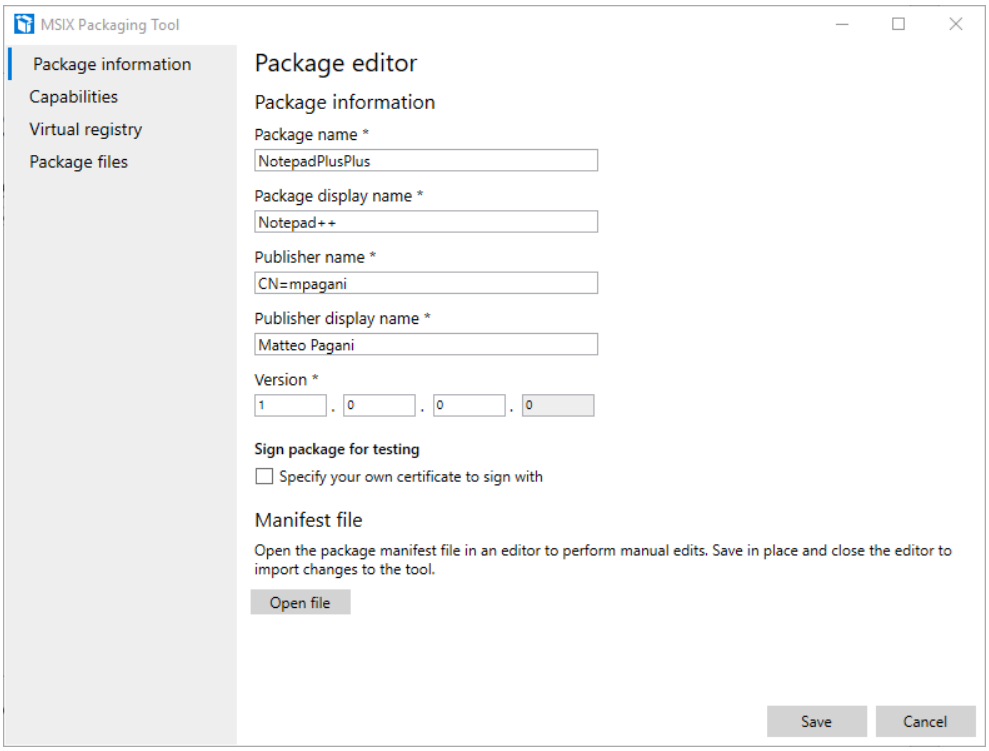


Figure 21: The Package information section of the package editor

This section can be used to change the identity of the application and define a different package name, publisher name, and so on. If you didn't sign the package when you created it, you can do it now by choosing the certificate on your disk.

You also have the opportunity to see the XML content of the manifest by pressing the **Open file** button. The file will be opened with a text editor. You will have the chance to make changes manually and save them. The updated manifest will be incorporated back inside the package.

Capabilities

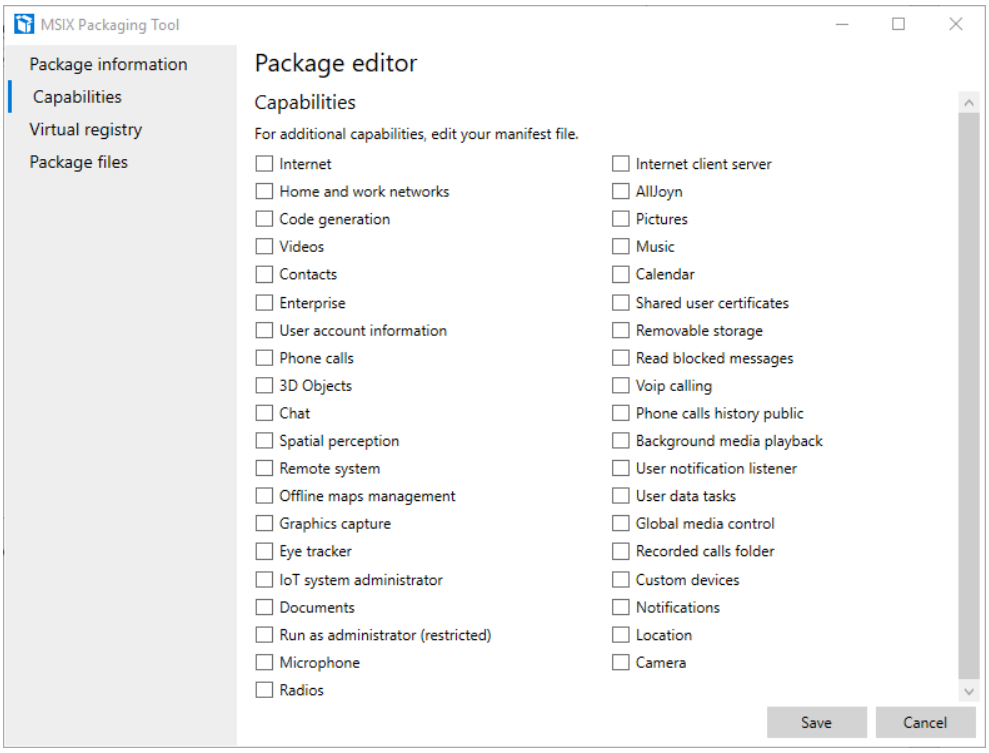


Figure 22: The Capabilities section of the package editor

This section lists all the available capabilities that can be accessed by an application. However, if you recall the section about the manifest in Chapter 1, you'll know that most of them apply only to Universal Windows Platform applications, and not to classic desktop applications. There may be some exceptions, however. For example, the **Run as administrator (restricted)** capability can also be leveraged by Win32 applications.

Virtual registry

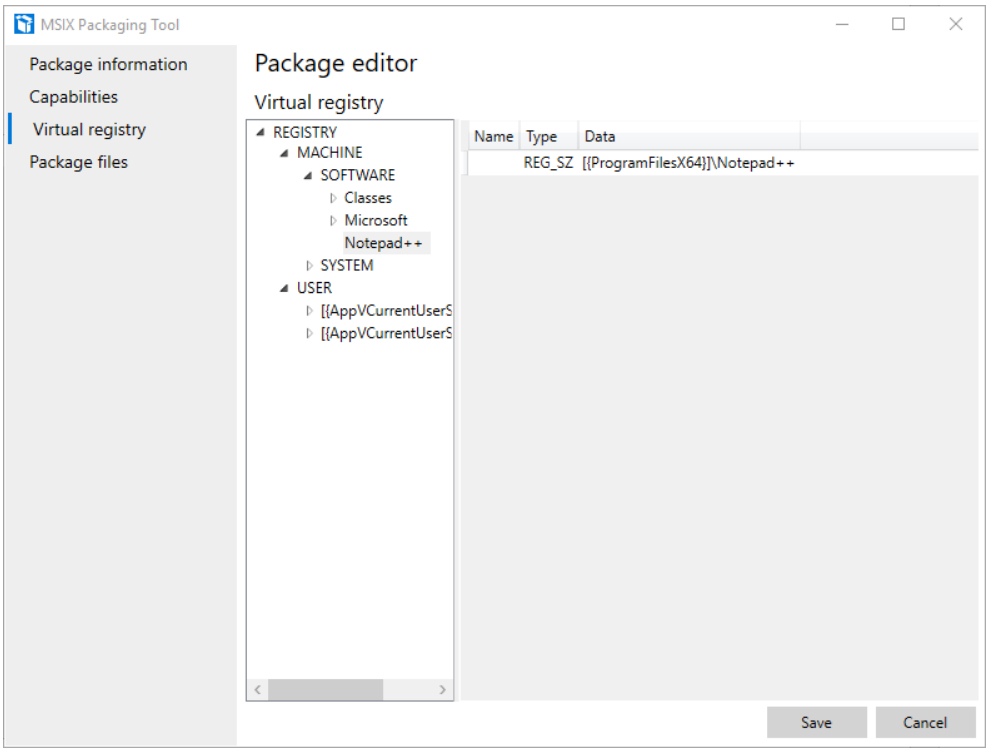


Figure 23: The Virtual registry section of the package editor

This section can be used to explore the .dat files that store the various registry keys created during the deployment process. Due to the virtualized registry approach provided by the MSIX container, these keys won't be visible in the system registry when the application is installed. As such, it's important to make any required changes in this section.

This section is also helpful to remove any unnecessary key that might have been captured by mistake or to add keys that might be missing. By right-clicking the various hives in the tree, you will have the option to add, edit, or delete keys and values.

Package files

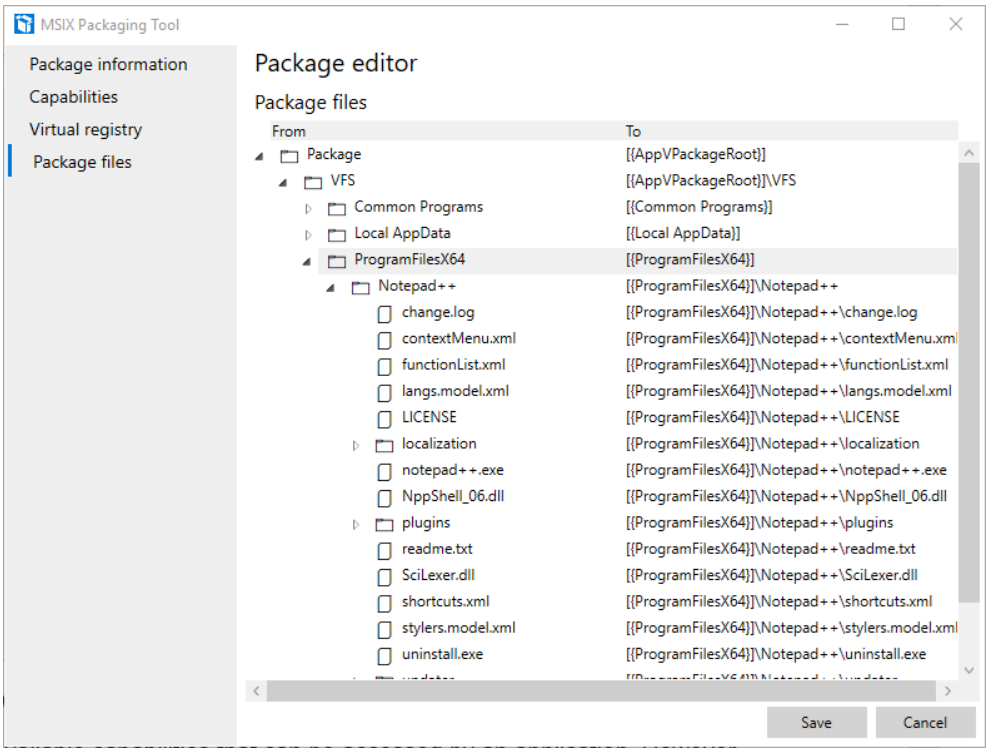


Figure 24: The Package files section of the package editor

You can use this section to explore all the files that are included inside the package. Like for the Virtual registry section, it can be helpful to remove files that aren't needed, or to add files that might be missing. By right-clicking on any element on the tree, you'll have the chance to delete it or to add a new file in the same location.

At any time, you can click **Save** at the bottom, which will generate an updated package with the changes you have applied.

MSIX Packaging Tool Insider Program

Thanks to an Insider Program, Microsoft offers a way to test new features of the MSIX Packaging Tool ahead of the public release. To join the program, you must follow the link included in the documentation, bringing you to a form in which you will be asked to share the Microsoft account you're using on the Microsoft Store. The Insider Program is managed through the flighting feature offered by the store. This means that the link used to get the application is the same as the standard version. However, if your Microsoft account is included in the program, you will get access to the Insider version, which will be automatically updated like a regular application.

Packaging your application with other tools

Microsoft has worked closely with many partners to bring MSIX support to the most popular authoring tools on the market, like Advanced Installer, InstallShield, Cloudhouse, and InstallAware. You can find the complete list in the official documentation.

Thanks to this integration, you'll be able to use the existing setup definition you're already using to generate a classic installer (like an MSI) to create an MSIX package.

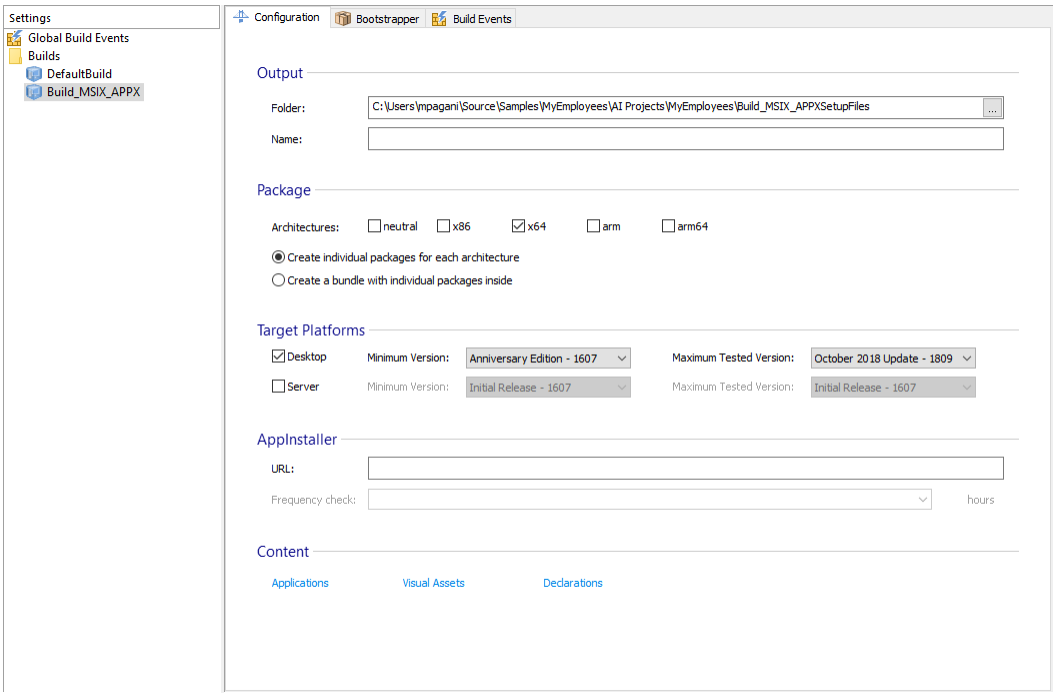


Figure 25: The option in Advanced Installer to generate an MSIX package

Additionally, many of these tools provide a visual front end to make it easier to integrate some of the features we're going to see in the next chapters, like modification packages and the Package Support Framework.

Chapter 3 Modification Packages

In Chapter 1, we learned that one of the goals of MSIX is to help enterprises be more agile by decoupling applications from customization from OS updates. One of the key features to achieve this goal is the **modification package**. Modification packages are special MSIX packages that don't contain a full application, but only files and registry keys. A modification package can't be installed on its own, it must target another application that is already installed on the system and deployed using MSIX.

When a modification package is installed, Windows will see its content as part of the same container that hosts the main application. This means every registry key and every file included inside the modification package will be treated like it's part of the main application.

Thanks to this approach, instead of having to customize the original MSI by creating a new installer from scratch, an IT pro can build a dedicated modification package with all the customizations. For example, it can include a configuration file, which the application can read to change its default settings; or a registry key, which is used to control a specific behavior. Because it's a separated package, the IT pro doesn't have to repeat the packaging process when the developer delivers a new update of the main application. The IT pro will just have to deploy the updated MSIX package, which won't affect the modification packages that are already applied.

Let's take a look at the manifest of a modification package.

Code Listing 6

```
<?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"
xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
xmlns:uap4="http://schemas.microsoft.com/appx/manifest/uap/windows10/4"
xmlns:uap6="http://schemas.microsoft.com/appx/manifest/uap/windows10/6"
xmlns:uap7="http://schemas.microsoft.com/appx/manifest/uap/windows10/7"
xmlns:mobile="http://schemas.microsoft.com/appx/manifest/mobile/windows10"
xmlns:iot="http://schemas.microsoft.com/appx/manifest/iot/windows10"
xmlns:desktop="http://schemas.microsoft.com/appx/manifest/desktop/windows10
"
xmlns:desktop2="http://schemas.microsoft.com/appx/manifest/desktop/windows1
0/2"
xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows
10/restrictedcapabilities"
```

```
xmlns:rescap3="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities/3"

xmlns:rescap6="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities/6"

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

  <Identity Name="MyEmployees-ExportPlugin" Publisher="CN=Contoso Software (FOR LAB USE ONLY), O=Contoso Corporation, C=US" Version="1.0.0.0" ProcessorArchitecture="x64" />

  <Properties>

    <DisplayName>MyEmployees (Export Plugin)</DisplayName>

    <PublisherDisplayName>Contoso</PublisherDisplayName>

    <Description>Reserved</Description>

    <Logo>Assets\StoreLogo.png</Logo>

    <rescap6:ModificationPackage>true</rescap6:ModificationPackage>

  </Properties>

  <Resources>

    <Resource Language="en-us" />

  </Resources>

  <Dependencies>

    <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.17701.0" MaxVersionTested="10.0.17763.0" />

    <uap4:MainPackageDependency Name="MyEmployees" Publisher="CN=Contoso" />

  </Dependencies>

</Package>
```

As you can see, this package has its own identity (different from the main package), but it doesn't have an **`Application`** entry. This package can't be installed as standalone, and it doesn't contain any application to launch.

However, inside the **`Dependencies`** section, in addition to the already known **`TargetDeviceFamily`** entry, we also find the **`MainPackageDependency`** entry, which supports two attributes:

- **`Name`**: the identity name of the main package we want to modify.
- **`Publisher`**: the publisher of the main package we want to modify.

The publisher doesn't have to match the one from the main application. This is the exact purpose of modification packages: IT pros must be able to modify applications that they don't own, or that have been built by a third-party vendor.

Another difference compared to a traditional package is the existence of the following entry inside the **`Properties`** section.

Code Listing 7

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

<Package

. . .

xmlns:rescap6="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities/6"
…. . .
  <Properties>

    <rescap6:ModificationPackage>true</rescap6:ModificationPackage>

  </Properties>

. . .

</Package>
```

This entry is required to support the new features added to MSIX in Windows 10 version 1903, like the ability to overwrite a file, or a registry key deployed by the main application with the same included in a modification package. This feature is useful for scenarios when the main application comes with a default configuration file and you want to override it with a custom configuration.

Using a modification package

Under the hood, modification packages leverage the same infrastructure of optional packages, which are part the Universal Windows Platform ecosystem. Optional packages provide a way to add content or native code to an existing application that is already deployed on the machine. Xbox games on the Microsoft Store heavily leverage this feature to handle add-ons.

The main difference between optional packages and modification packages is that the first mainly target developers. Once an optional package has been deployed on the machine, you must use the **`Windows.ApplicationModel.Package.Current.Dependencies`** API to iterate them, access the built-in files, or load code stored inside the package. Modification packages, instead, leverage the virtual file system we learned to use in Chapter 1. When you add a registry file or a VFS folder inside a modification package, they are merged with the content of the main application at runtime. As a consequence, you don't have to change the application's code in order to support them.

This means that, for example, if the main application tries to load a configuration file from the **C:\Program Files** folder, you can deploy this file using a modification package by storing it inside the special folder **VFS\ProgramFiles**. Since they share the same container, the main application will be able to pick this file as if it's included in the **VFS** folder of the main package.

Another example is if the application tries to read a registry key from the **HKEY_CURRENT_USER** hive. In this case, you can deploy a **User.dat** file with the key inside the modification package. Since they share the same container, the main application will be able to read the value of this registry key as if the file is included in the main package.

Handling modification packages

Modification packages can be installed only if the main application specified by the **`MainPackageDependency`** entry in the manifest is installed. Otherwise, you will experience the following error.

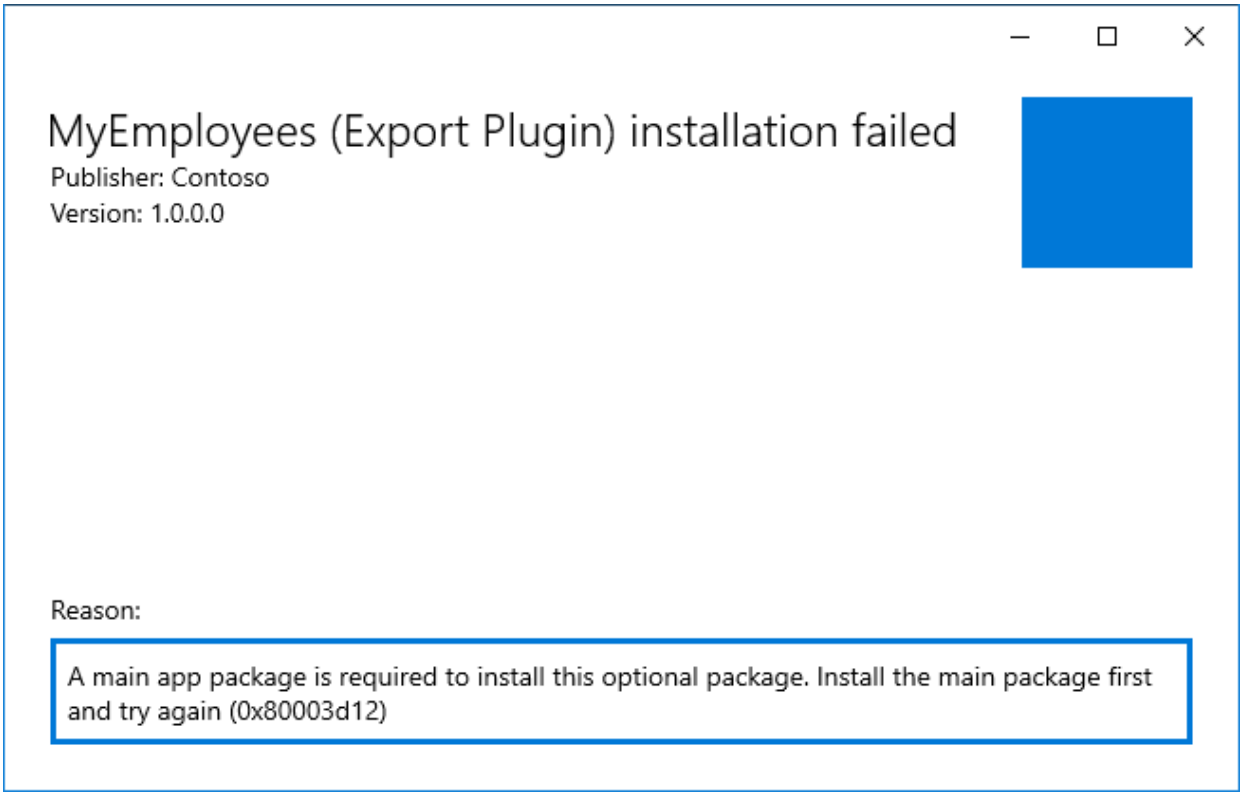


Figure 26: The error displayed when you try to install a modification package without having installed the main application that it's targeting

Once a modification package has been installed, it can be handled by using the application's settings in Windows. To see this section, right-click on the main application in the Start menu and choose **More** > **App settings**. You will find many options to reset the application and, at the bottom, a section titled **App add-ons & downloadable content**, which contains the list of installed modification packages. You will also have the opportunity to remove them.

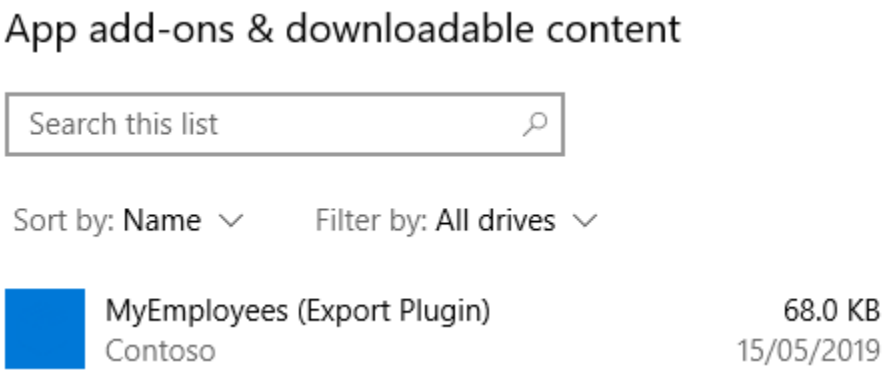


Figure 27: The list of modification packages installed for the selected application

Being separated, modification packages are completely independent from the main application. This means that, as an IT pro, you don't have to repackage the original application and deploy it from scratch when:

- You need to update the main application with a newer version.
- You need to update the modification package with an updated customization.
- You need to remove the modification package to restore the application to the original configuration.

Create a modification package

The MSIX Packaging Tool supports the creation of modification packages. The flow is the same as creating a main application. The tool will start monitoring all the changes that happen at the file system and at the registry and will include them inside the package. In order to start the process, choose the **Modification package** option when you launch the tool.

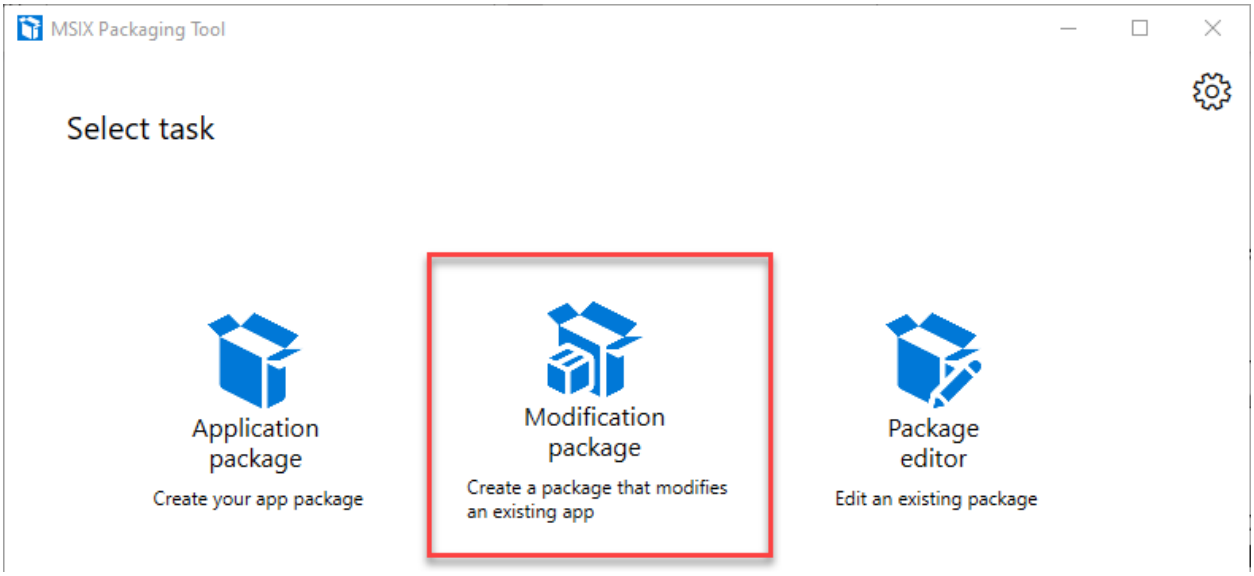


Figure 28: The option in the MSIX Packaging Tool to create a modification package

The process is the same as creating the package for a traditional application. You will be able to start from an installer that deploys the customization, or you can start the monitoring process and then create the files and the registry keys you want to include.

The only difference is in the first step, the one called **Select installer**.

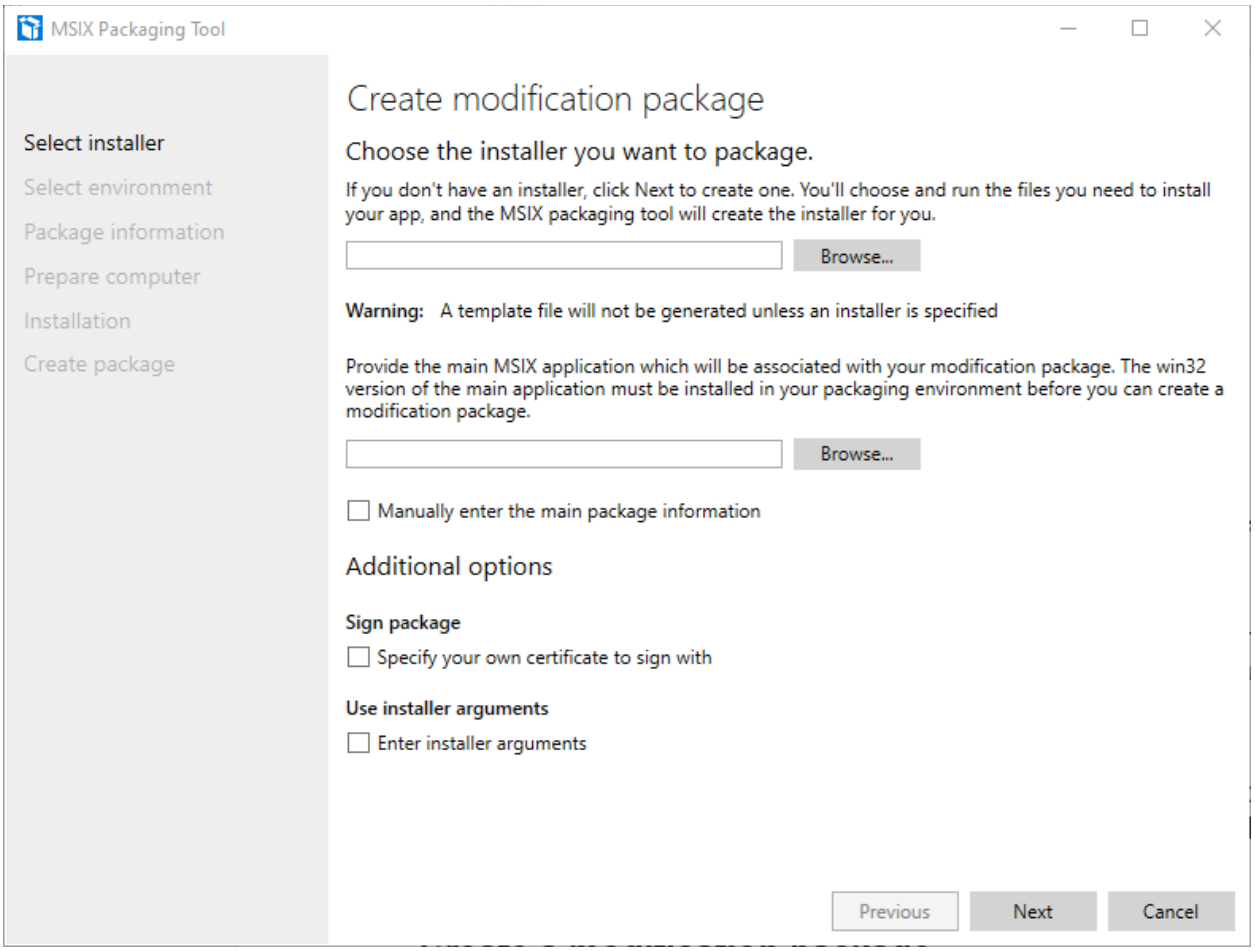


Figure 29: The starting point to create a modification package with the MSIX Packaging Tool

As you can see, unlike the process to create the package for an application, you are required to specify the main MSIX application you want to target. If you already have the MSIX package on your computer, you can click **Browse** and select it. The tool will automatically extract all the relevant information and add the required **`MainPackageDependency`** entry in the manifest. If you don't have it, you can select the **Manually enter the main package information** check box to enable two additional fields, where you'll be able to manually insert the name and the publisher of the package you want to target.

The rest of the process will be the same as the one you followed to create the main application's package, except that there will be fewer steps. For example, the **First launch tasks** step will be missing, since a modification package doesn't have an entry point.

A real example

Let's see a concrete example of a modification package to better understand how it works. The starting point is an application called **MyEmployees**, which is a Windows Forms application to handle the employees who work in a company. As you can see from the screenshot in Figure 30, the application has an About page, which comes as a customizable solution. The default text is just a placeholder.

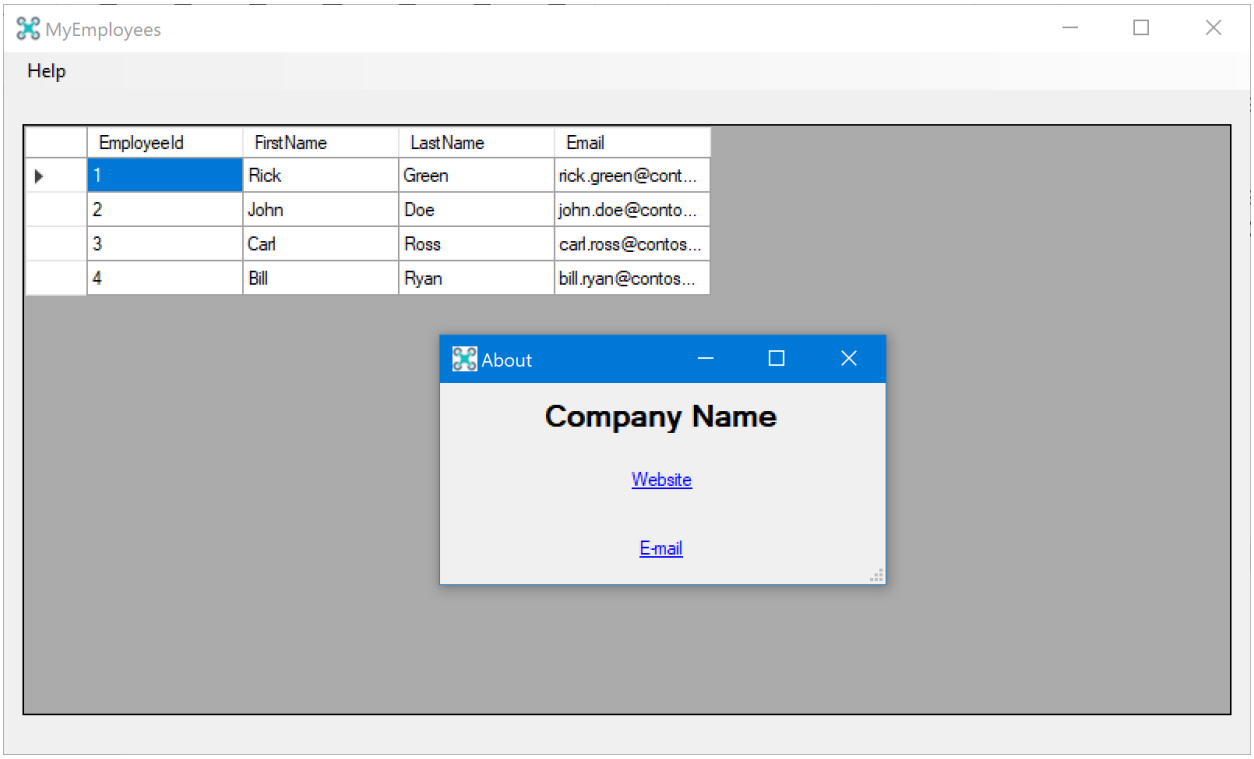


Figure 30: The MyEmployees application with an empty About page

The developer has built this application with the purpose of allowing IT pros to easily configure some settings, including the content of the About page. This is achieved by leveraging a configuration file, called **config.json**, which the application expects to find in the **C:\Program Files (x86)\Contoso\MyEmployees** folder. If the file isn't available, the application will retain the default settings.

This application has been packaged with MSIX, and it doesn't include a **config.json** file. As such, when you deploy it, you will see something like the screenshot shown in Figure 30.

The IT department has now built a modification package using the MSIX Packaging Tool, which includes a **config.json** file like the following one.

Code Listing 8

```
{
```

```
  "isCheckForUpdatesEnabled": "false",
  "about": {
    "companyName": "Contoso",
    "supportLink": "http://www.contoso.com",
    "supportMail": "support@contoso.com"
  }
}
```

Since the main application expects this file to be in the **C:\Program Files (x86)\Contoso\MyEmployees** folder, the IT engineer has placed it inside the **VFS\ProgramFileX86\Contoso\MyEmployees** folder of the modification package.

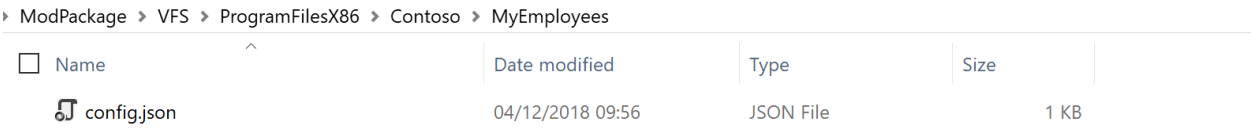


Figure 31: The config.json file inside the VFS folder of the modification package

Once you install the modification package, Windows will treat the VFS folder inside it like it's part of the main application. This means that the application will succeed when it tries to read the **config.json** file, even though it wasn't included in the original package. As such, when you launch the app and you open the About window, you will see the following outcome.

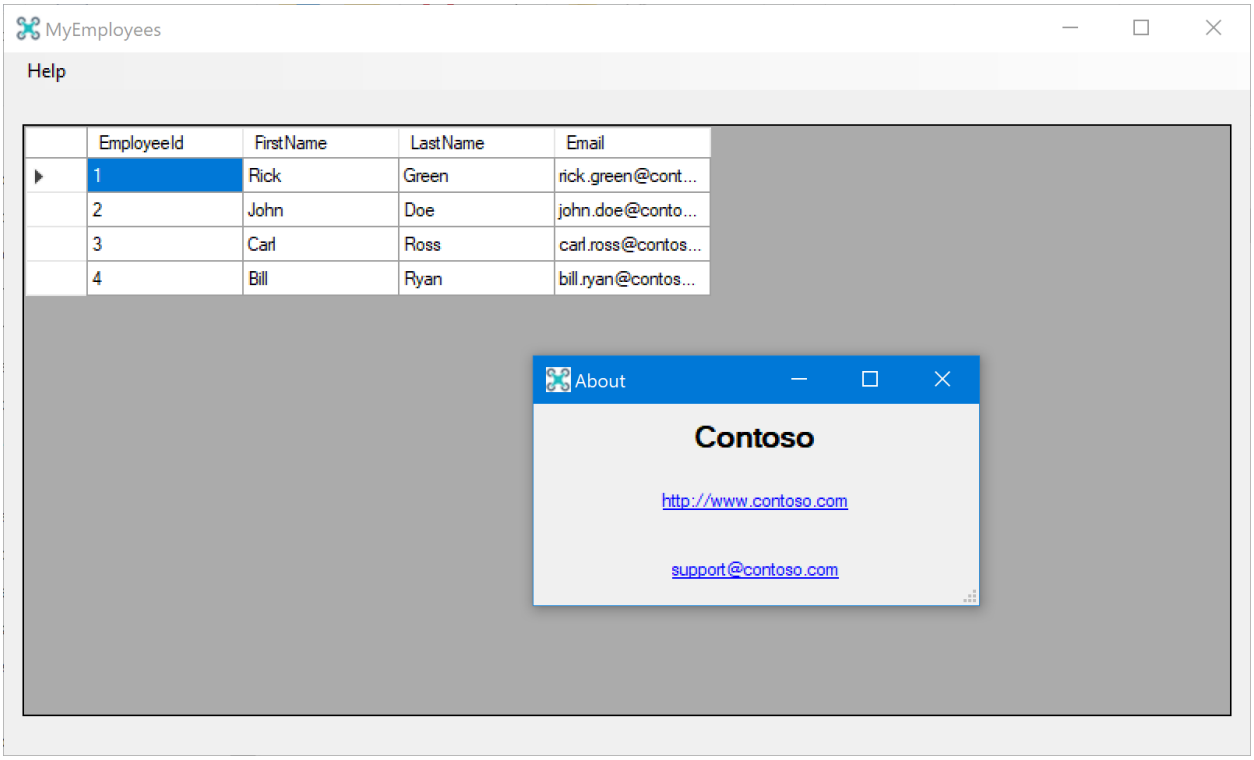


Figure 32: The MyEmployees application after the installation of the modification package

The content of the **config.json** file has been parsed and converted into the text displayed inside the About window.

Now let's assume that, after some months since the first deploy, the developer of the application releases a new version that adds a set of new features. The developer shares with the IT department a new MSIX package that contains the newer version of the main application. Since the customization is stored inside the modification package, the IT department can deploy it immediately. The existing customization will be retained, so after the deployment, users will enjoy the new features of the application with the same configuration they were using in the previous version.

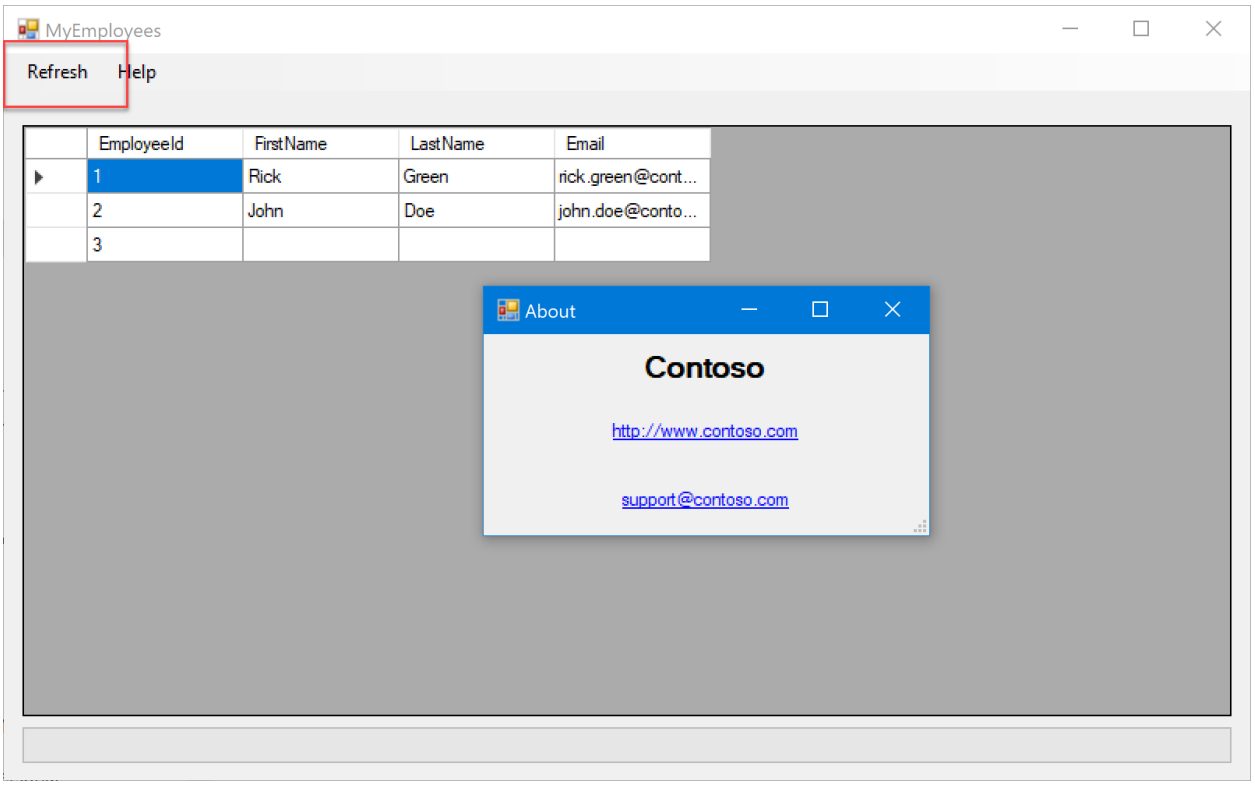


Figure 33: The updated version of the MyEmployees application, which leverages the existing modification package

As you can see in Figure 33, the application now has a new Refresh option in the menu, but the modification package that customizes the content of the About window is still applied.

Chapter 4 The Package Support Framework

In Chapter 1, we learned that even if the container enforced by MSIX is very lightweight, there are still some different behaviors between a traditional Win32 app and a packaged one. There are some requirements that we need to satisfy in order to make a successful packaging. For example, since MSIX packages don't create a traditional shortcut in the Start menu, which references the executable of the application, the APIs to retrieve the current working directory will return **C:\Windows\System32** or **C:\Windows\SysWOW64** (based on the app's architecture) instead of the folder where the application has been deployed. This behavior can create problems for applications that, at runtime, try to load files that are deployed together with the executable in the same folder, like a database or an image.

Another different behavior that we can experiment with is related to writing operations. MSIX packages are deployed inside the **C:\Program Files\WindowsApps** folder, which is system-protected. The application, which instead runs at the user level, doesn't have permission to write in that folder. However, some applications might use the installation folder as a location to write some content, like a log file or a configuration file. In such a scenario, you will experience an exception, since the writing operation will fail.

The best approach to solving this kind of problem is changing the code and fixing the wrong behavior. However, this isn't always possible. As an IT pro, you might need to deploy applications coming from third-party developers, so you don't have access to the source code. Or maybe it's a very old application that is still widely used, but no longer maintained.

The Package Support Framework is an open-source project, published by Microsoft on GitHub, that aims to solve these scenarios by allowing the IT pro to change the logic of the application without modifying the code. Let's take a look at the architecture of the Package Support Framework.

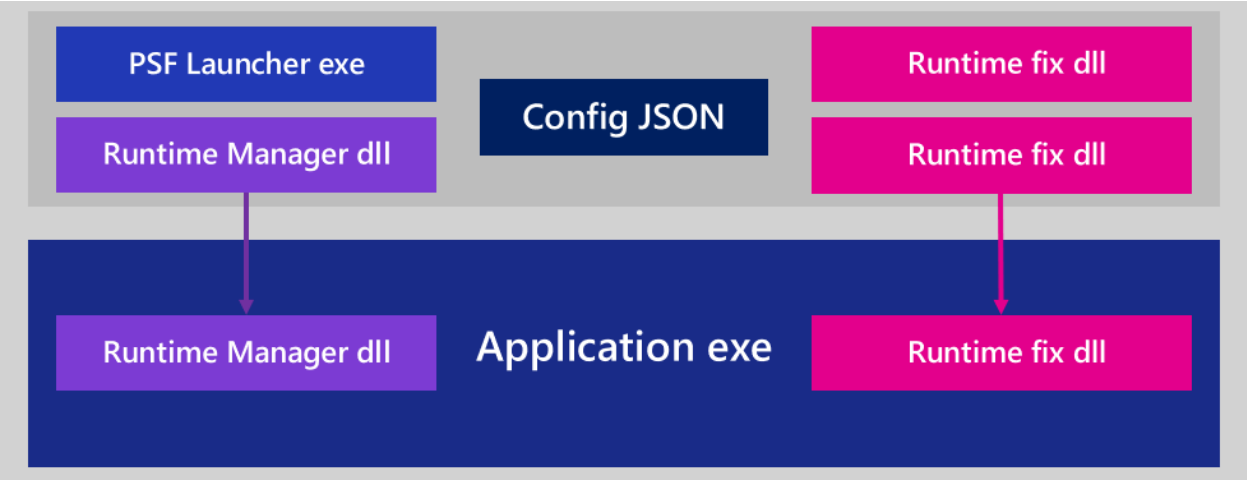


Figure 34: A packaged application which uses the Package Support Framework

The Package Support Framework is composed of four components:

- A launcher, which is an executable.
- A runtime manager, which is a DLL.
- A JSON configuration file.
- One or more runtime fixes, which are DLLs that contain the logic to change the behavior of the application.

When the Package Support Framework is applied to a packaged application, the launcher replaces the application's executable as the main entry point. The purpose of the launcher is to execute the main application with the injected runtime manager. Thanks to this implementation, the runtime manager is able to load runtime fixes that are able to change one or more behaviors of the app on the fly. The whole configuration is controlled by a JSON file, which specifies parameters like the current working directory to use, the main executable to launch, and which runtime fixes to apply.

The runtime fixes can change the workflow of Windows APIs by intercepting an operation and replacing it with a different behavior. For example, a fixup can intercept the writing operation happening on the installation folder and redirect it to another location where the application has write permission. This task is accomplished without changing the code of the original application.

Adding the Package Support Framework

The first step to add the Package Support Framework to an application is to download it. The framework is available on NuGet. However, in our scenario, we don't have the source code of the application, so we don't have a project in Visual Studio that we can open to install the NuGet package.

The easiest way to get it is to download the NuGet standalone executable to a folder of your choice. Once you have downloaded it on your machine, open a command prompt in that folder where you have saved it, and run the following command.

Code Listing 9

```
nuget.exe install Microsoft.PackageSupportFramework
```

Once the operation is completed, you will find in the same folder a new subfolder called **Microsoft.PackageSupportFramework.x.y.z**, where **x.y.z** is the version number of the most recent version. Inside the **bin** folder, you will find all the executables and DLLs that constitute the Package Support Framework, and which you will need to add to your packaged application.

However, at this point, you will probably already have a packaged application you have deployed, and you'll have verified that it suffers from an issue that can be fixed with the Package Support Framework. Before moving on, you will have to unpack the application so that you can add new files to it and change the configuration. You can achieve this goal with the **makeappx** tool, which is included in the Windows 10 SDK. If you don't already have it on your machine, you can install the standalone version.

Once you have installed the Windows 10 SDK, the tool will be available in the **C:\Program Files (x86)\Windows Kits\10\bin\10.0.xyz.0\x86** folder, where **xyz** is the version of the Windows 10 SDK you have downloaded. For example, at the time of writing, the most recent available Windows 10 SDK is **10.0.18362.0**, so you'll find the tool in the **C:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86** folder.

Once you have identified the right folder, open a command prompt on it and launch the following command.

Code Listing 10

```
makeappx unpack /p SamplePackage.msix /d PackageFiles
```

The **/p** parameter references the full path of the MSIX package, while the **/d** one points to the folder where you want to unpack the content of the MSIX package. After the operation is completed, the **PackageFiles** folder will contain all the files that comprise the application, plus the specific MSIX ones, like the app manifest and the virtual registry.

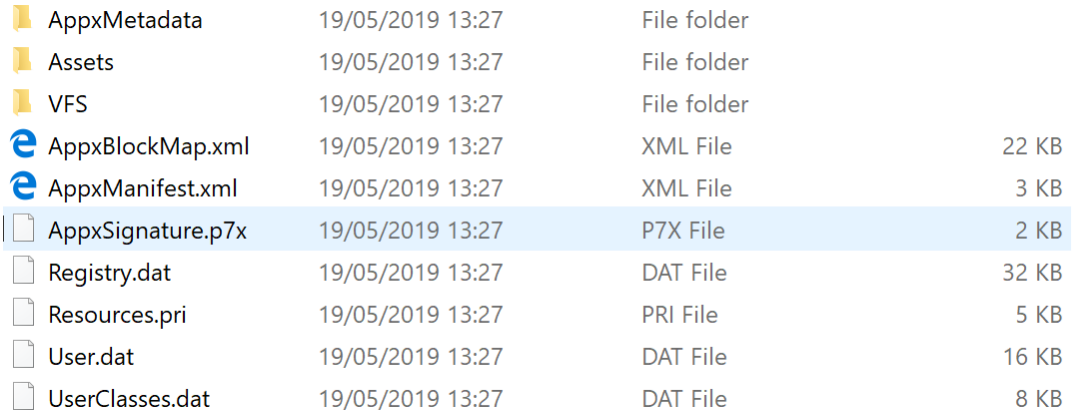


Figure 35: The content of a MSIX package once it has been unpackaged with the MSIX tool

The first step is to copy, inside this folder, the files that compose the Package Support Framework. You can find them in the **Microsoft.PackageSupportFramework.x.y.z\bin** folder, which you previously downloaded using NuGet. The minimum set of files to copy is:

- PsfLauncher.exe
- PsfRunDll.exe
- PsfRuntime.dll

The folder contains two variants of the files, one for each CPU architecture (x86 and x64). You need to choose the right version based on the CPU architecture for which the app has been compiled.

The folder also contains a set of fixups, which are ready to be used:

- **FileRedirectionFixup.dll**, to help you to solve the writing permission scenario we have previously described.

- **TraceFixup.dll**, to identify issues within the application. We're going to see later how to use it.
- **WaitForDebuggerFixup.dll**, to allow you to hold the process during startup until a debugger is attached.

If any of these fixups can be helpful to solve your scenario, just copy them inside the folder that contains the unpackaged version of your application.

Configuring the fixups

Once you have copied the required files, you must configure the framework using a JSON file called **config.json**, which must be included inside the root of the package.

This is what a typical configuration file looks like.

Code Listing 11

```
{
  "applications": [
    {
      "id": "PSFSample",
      "executable": "PSFSampleApp/PSFSample.exe",
      "workingDirectory": "PSFSampleApp/"
    }
  ],
  "processes": [
    {
      "executable": "PSFSample",
      "fixups": [
        {
          "dll": "FileRedirectionFixup.dll",
          "config": {
            "redirectedPaths": {
              "packageRelative": [
                {
```

```
                "base": "PSFSampleApp/",
                "patterns": [
                  ".*\\.log"
                ]
              }
            ]
          }
        }
      }
    ]
  }
]
}
```

The **applications** entry specifies the base configuration of the application by leveraging the following properties:

- **id** must match the application's identifier declared in the manifest. To find it, open the **AppxManifest.xml** file inside the package's folder with a text editor and look for the **id** attribute of the **Application** entry, as highlighted in the following sample.

Code Listing 12

```
<Applications>
  <Application Id="PSFSample" Executable="PSFSampleApp\PSFSample.exe"
EntryPoint="Windows.FullTrustApplication">


  </Application>
</Applications>
```

- **executable** is the main entry point of the application. Remember that when you use the Package Support Framework, the entry point becomes the launcher included in the tool, so we need to specify which is the real main process of the application.
- **workingDirectory** is the folder inside the package that contains the main executable. Thanks to this entry, the APIs to retrieve the current working directory will return the correct one instead of **C:\Windows\System32 or C:\Windows\SysWOW64**.

The **processes** section is used to configure the various fixups you have decided to include in the package:

- The **executable** property contains the name of the main executable of the application.
- **fixups** is a collection that contains one entry for each fixup.

Each fixup is defined by the **dll** property, containing the name of the DLL that implements it without any reference to the CPU architecture, and the **config** property, containing the fixup configuration. This last property doesn't have a fixed structure: it's up to the developer of the fixup to implement and document its own config. For example, on GitHub, you can find the documentation for the File Redirection Fixup, which is included in the previous sample. We'll see more details about the File Redirection Fixup later in this chapter.

Configuring the manifest

Once you have added the **config.json** file to your package, you must also update the manifest. Remember that to leverage the Package Support Framework, we need to change the entry point of the application. We must use the built-in launcher, which will take care of launching the real application and injecting the runtime and the various fixups.

To achieve this goal, open the **AppxManifest.xml** file and look, once again, for the **Application** entry, which will be something like the following.

Code Listing 13

```
<Applications>

  <Application Id="PSFSample" Executable="PSFSample\PSFSample.exe"
EntryPoint="Windows.FullTrustApplication">


  </Application>

</Applications>
```

As you can see, the **Executable** attribute will reference the main executable of the application. You must replace this value with the **PSFLauncher.exe** executable, which is part of the Package Support Framework. Remember to reference the correct version based on the application's architecture. For example, this is how the **Application** entry will look for a 32-bit application.

Code Listing 14

```
<Applications>

  <Application Id="PSFSample" Executable="PSFLauncher32.exe"
EntryPoint="Windows.FullTrustApplication">
```

```
  </Application>
</Applications>
```

Repackaging the application

The last step is to repackage the application in order to get back a MSIX package that we can deploy. We can again use the **makeappx** utility with the same parameters. We just need to change the action to perform from **unpack** to **pack**, as in the following example.

Code Listing 15

```
makeappx pack /p SamplePackage.msix /d PackageFiles
```

Once we have an MSIX package, we need to sign it before deploying it. Remember that unsigned MSIX packages can't be installed at all. For this task, we can use a tool included in the Windows 10 SDK, called **signtool.exe**. It's located in the same folder of the **makeappx** tool, which is **C:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86.**

Here is a sample command that you can use to sign the package.

Code Listing 16

```
signtool.exe sign /tr http://timestamp.digicert.com /a /v /fd SHA256 /f
"MyCertificate.pfx" /p "Password" SamplePackage.msix
```

The three relevant parameters that you must customize are:

- **/f**, containing the full path of the PFX certificate you want to use to sign the package.
- **/p**, containing the password of the certificate.
- The last parameter, which is the full path of the MSIX package you want to sign.

Now you're ready to deploy the updated version of your application leveraging the Package Support Framework.

Identifying potential issues

The first challenge of using the Package Support Framework is identifying the wrong behavior of the app, and which fixups might be helpful to fix it. There are cases where it's easy to identify the problem. For example, let's say you have an application that triggers the following problem at startup.

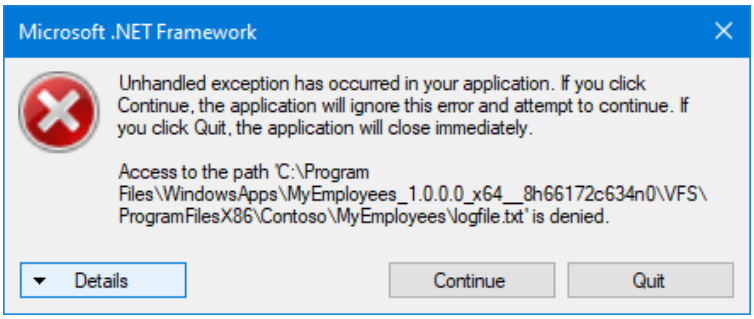


Figure 36: The error displayed when a packaged application tries to write inside the installation folder

In this case, it's easy to understand that the application is trying to write a file inside the installation folder. The error is reporting that access has been denied to a text file called **logfile.txt** stored in the folder where the package is deployed.

Another common scenario is when the application crashes with an error like the following one.

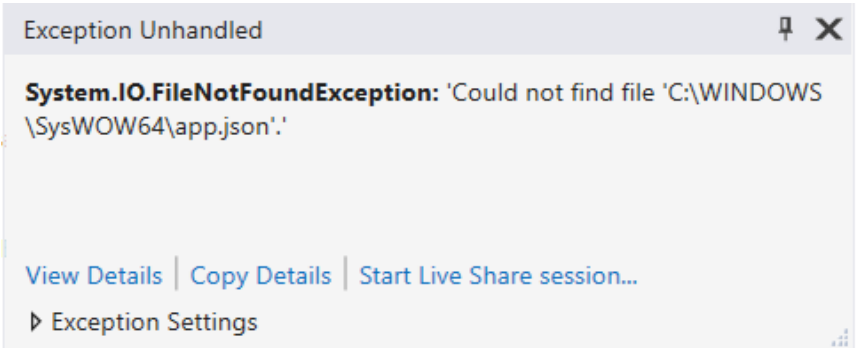


Figure 37: The error displayed when a packaged application tries to read from the current working directory

It's easy to understand what's happening. The application is trying to read a file called **app.json** from the **C:\Windows\SysWOW64** folder, which is the current working directory for a packaged app. It means that the application is using the Windows APIs to read a file from the installation folder, but since the Current Working Directory is different, it can't find it.

However, in some cases the behavior could be sneakier, and we might not have a clear error that helps us to understand what's happening. The application might just crash without logging any meaningful error. A good way to troubleshoot potential issues is by using one of the fixups provided by the Package Support Framework called Trace Fixup. Thanks to it, all the operations performed by the application that don't complete successfully (like reading a file from the hard disk or writing a registry key) will be logged and can be analyzed with different tools, based on the configuration.

The first step to using Trace Fixup is to make sure to include it inside the package. When you copy the Package Support Framework in the folder where you have unpackaged the application, make sure to copy also the **TraceFixup32.dll** or **TraceFixup64.dll** file, depending on the application's architecture.

The second step is to configure the fixup in the **config.json** file. This is what a typical configuration looks like.

Code Listing 17

```
{
  "dll": "TraceFixup.dll",
  "config": {
    "traceMethod": "outputDebugString",
    "traceLevels": {
      "default": "allFailures"
    },
    "breakOn": {
      "filesystem": "unexpectedFailures"
    }
  }
}
```

The **traceMethod** property can be used to specify where you want to output the traces recorded by the fixup. In the previous sample, we set to output the logs to the system trace.

Then we can specify two different kind of behaviors:

- **traceLevels** is used to define which kind of failures must be logged.
- **breakOn** is used to define which kind of failures should trigger a breakpoint if you have an attached debugger.

For each behavior, you can specify which activities and which kinds of failures you want to log. In the previous snippet, you can see two different samples:

- **default** and **allFailures** means that we want to log all the failures, regardless of the type.
- **filesystem** and **unexpectedFailures** means that we want to log only the unexpected failures that happen on the file system.

You can find the list of all the supported values in the documentation on GitHub.

Once you have included the Trace Fixup, you can use a debugger or a diagnostic tool like Debug View, which is part of the Sysinternals suite. For example, let's see what happens when you open Debug View and then launch an application with the Trace Fixup injected through the Package Support Framework.

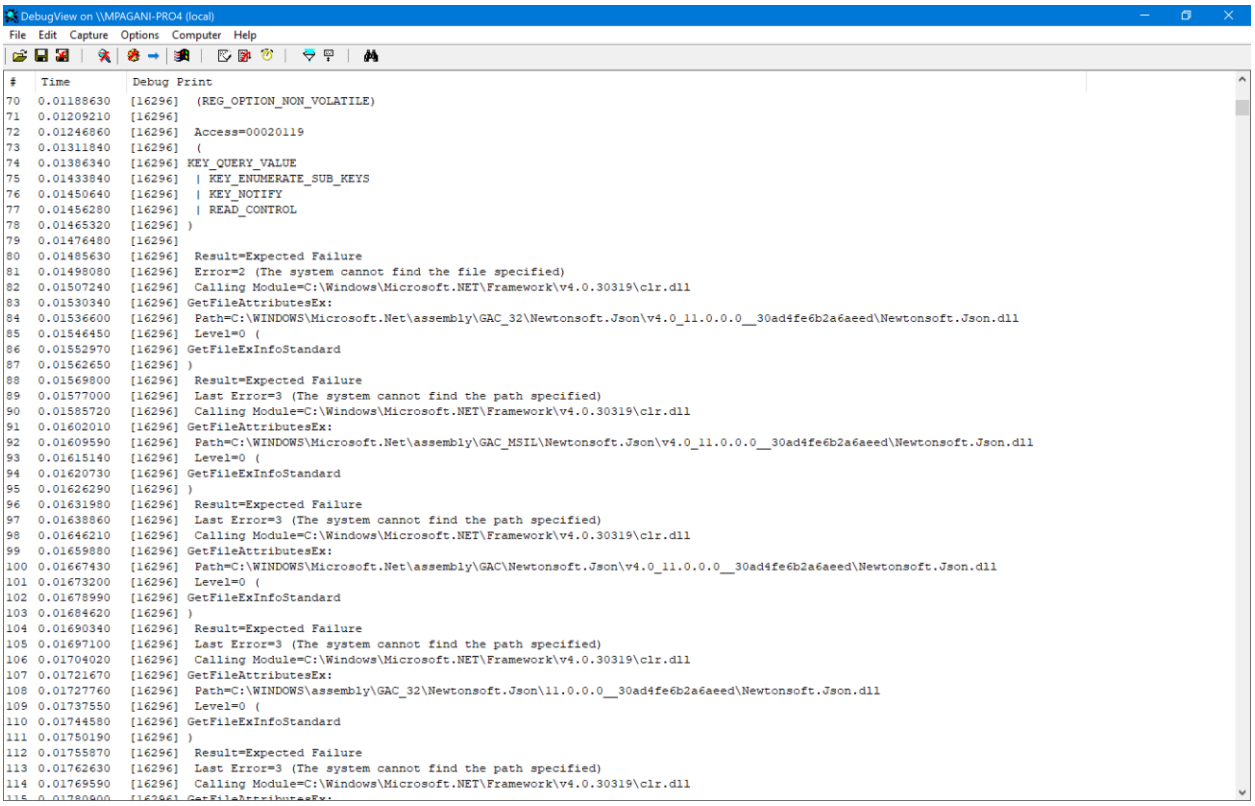


Figure 38: Debug View collects the trace output from an application that uses the Trace Fixup

As you can see from Figure 38, the Trace Fixup is sending to the system trace the output of all the operations performed by the application on the file system and on the registry, based on the configuration we have set. With some digging, we will be able to find out if the application is failing because it isn't able to file a file or a registry key, meaning that the lightweight container provided by MSIX could interfere with the regular flow of the application.

Enabling file redirection

As we learned at the beginning of this section, one of the most common issues that affects applications' MSIX packages is the inability to write in the installation folder. For this reason, the **File Redirection Fixup** is built into the Package Support Framework.

This fixup is implemented by the **FileRedirectionFixup32.dll** and **FileRedirectionFixup64.dll** files, so make sure to copy the right one (based on your application's architecture) inside the folder where you have unpackaged the MSIX.

Here is a sample configuration of the fixup.

Code Listing 18

```
{
```

```
  "dll": "FileRedirectionShim.dll",
  "config": {
    "redirectedPaths": {
      "packageRelative": [
        {
          "base": "PSFDemo/",
          "patterns": [
            ".*\\.json"
          ]
        }
      ]
    }
  }
}
```

We're using the **packageRelative** option, which means we want to redirect all the writing operations performed in a folder relative to the package. In this case, we are redirecting all the operations executed inside the installation folder, which is called **PSFDemo** and is specified using the **base** property.

The last step is to use the **patterns** option to specify which kind of operations we want to redirect. In this scenario, we're assuming that our application tries to write a file with the .json extension in the installation folder, so we specify **.*\\.json** as the pattern.

Another option we can use is called **redirectedPaths**, which is helpful when the application tries to write some content in a system folder. This is a sample configuration of this scenario.

Code Listing 19

```
{
    "dll": "FileRedirectionFixup.dll",
    "config": {
        "redirectedPaths": {
            "knownFolders": [
```

```
                {
                    "id": "ProgramFilesX86",
                    "relativePaths": [
                        {
                            "base": "Contoso\\MyEmployees",
                            "patterns": [
                                ".*\\.txt"
                            ]
                        }
                    ]
                }
            ]
        }
    }
}
```

In this case, the writing operation causing trouble for our application is performed inside the **C:\Program Files (x86)\Contoso\MyEmployees** folder, which is one of the folders redirected by Windows using the virtual file system. In this scenario, we use the **`knownFolders`** property to specify the identifier of the VFS folder where we want to enable the redirection, which is **`ProgramFilesX86`**. The rest of the configuration is the same as the previous one: we specify the base folder (**`Contoso\MyEmployees`**) and which file operations we want to redirect (the ones involving files with the .txt extension).

But where are these files being written? They are written inside the local storage of the application, which is located in **%LOCALAPPDATA%\Packages\<Package_Family_Name>\LocalCache**, where Package Family Name is the one assigned to the MSIX package.

This is, for example, the content of local storage of the application that is using the configuration file showcased in the previous snippet.

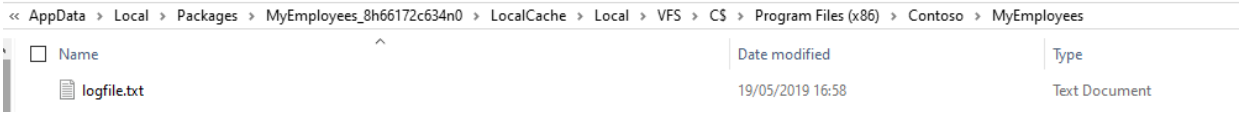


Figure 39: A file redirected to the local storage

As you can see, we have intercepted the creation of a .txt file, called **logfile.txt**. The file has been created inside the local storage of the application, in a folder that follows the same path of the original location of the file (**C:\Program Files (x86)\Contoso\MyEmployees**).

From now on, Windows will redirect all the operations, including the reading ones, to leverage this file. This means that, whenever the application tries to read the **logfile.txt** file, Windows will provide a reference to the one in the local storage, and not to the original one.

Chapter 5 Packaging Your Applications with Visual Studio

In Chapter 2, you learned about the MSIX Packaging Tool, a great tool for helping IT pros repackage applications. However, if you're a developer, you will probably feel more comfortable doing the packaging in an environment you probably already know very well and use every day for your development tasks: Visual Studio.

Visual Studio offers built-in MSIX packaging integration, which makes it easy to package an existing project built with WPF, Windows Forms, or Visual C++. In the most recent versions, Visual Studio has also gained the ability to package applications for which you don't own the source code. You won't have support for all the features (like debugging), but you will be able to leverage a better manifest editor and an integrated experience to generate an MSIX package.

To package an application with Visual Studio, you will need to install:

- Visual Studio 2019. The Community edition, which is free, is perfectly suitable for this task.
- The Windows 10 SDK. At the time of writing, the most recent version is 18362, which pairs with Windows 10 version 1903. It can be downloaded from here, but the simplest way to install it is to enable the **Universal Windows Platform development** workload during the setup process.

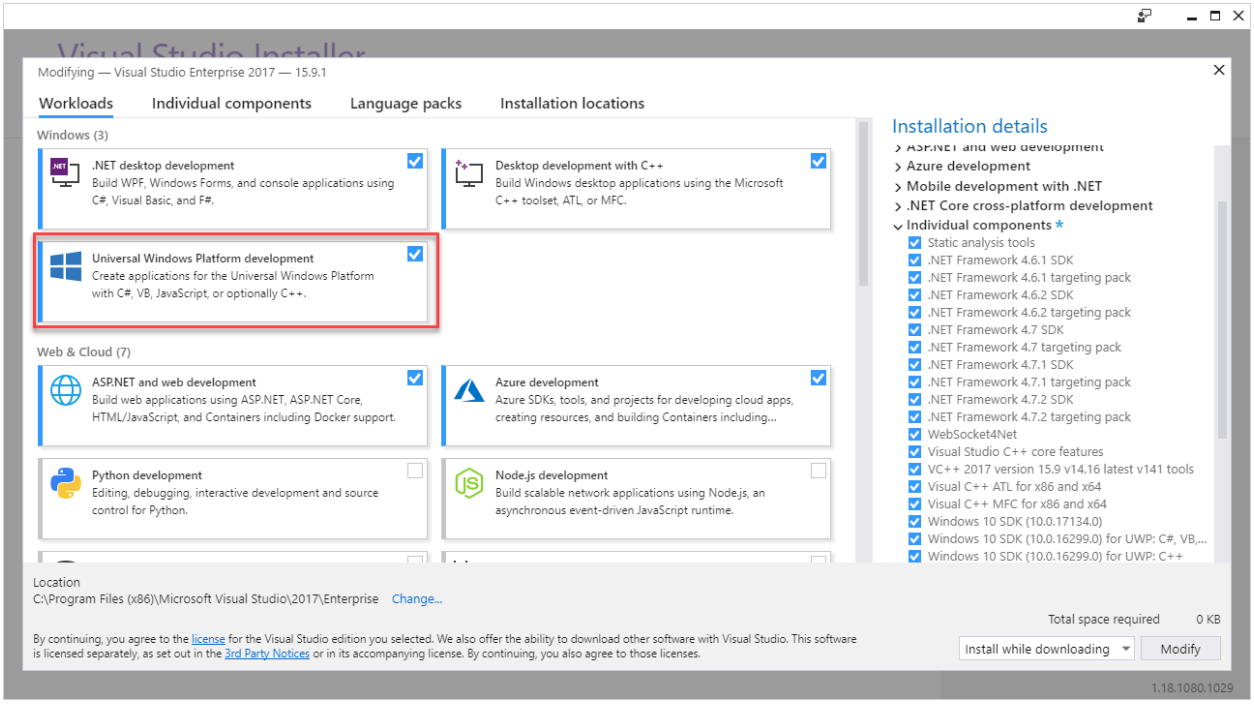


Figure 40: The workload to enable to leverage MSIX in Visual Studio

You will also need to make sure that the Developer Mode is turned on in Windows, as explained in Chapter 2.

Once you have met all the requirements, the first step is to open the Visual Studio solution that contains your desktop application's project. Right-click on the solution inside **Solution Explorer** and choose **Add** > **New project**. Search for the template called **Windows Application Packaging Project**, as displayed in the following image.

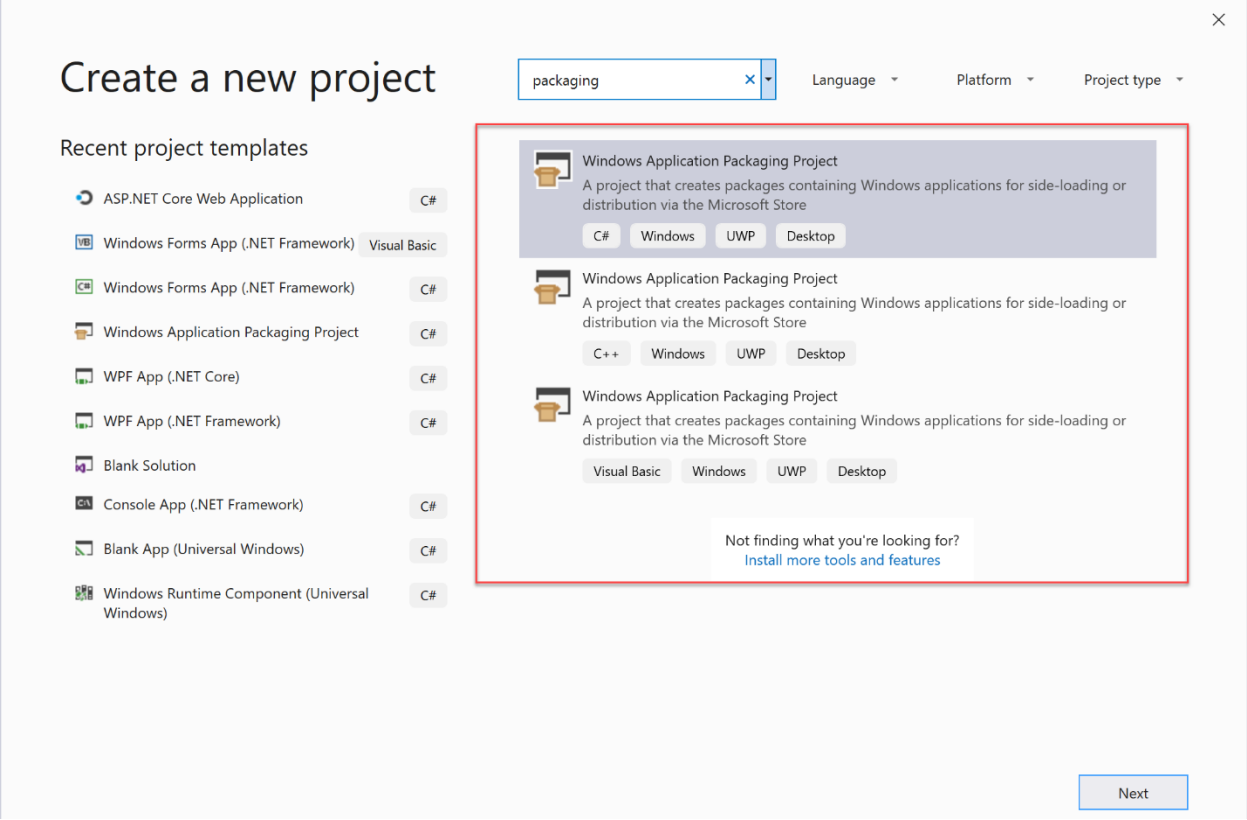


Figure 41: The Windows Application Packaging Project in Visual Studio 2017

You'll first be asked which target version and the minimum version of Windows 10 you want to leverage.

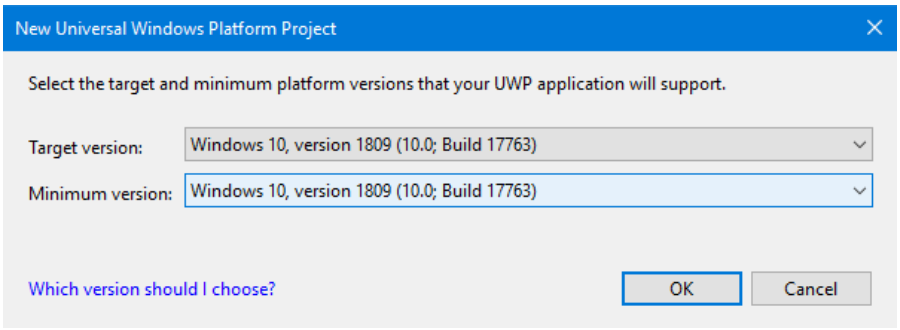


Figure 42: Setting up the target and minimum version for the project

The **target version** defines the API surface of the Universal Windows Platform the application will have access to. If your application doesn't leverage any feature from the Universal Windows Platform, so is a pure classic desktop application, this choice won't have any effect.

The **minimum version** defines the minimum version of Windows 10 required to run this application. To generate a MSIX package, you need to set both as target and minimum version at least **Windows 10 version 1809 (10.0; Build 17763)**. Setting a lower version will cause Visual Studio to generate a package using AppX, the previous packaging format.

The project will look like the one for the Universal Windows Platform application. You have a manifest file, called **Package.appxmanifest**, which you can edit with the visual editor by double-clicking on it. You also have a folder called **Images**, which contains the default icons used for the Start screen, the taskbar, and so on.

However, there's a big difference compared to a standard development project. Instead of the **References** section, which is typically used to add third-party libraries to your project, you will find one called **Applications**. The Windows Application Packaging Project, in fact, being meant for packaging existing apps, can't be used to include code; it must contain a reference to the application you want to package.

To select it, right-click the project and choose **Add references**. You will have the opportunity to choose which one of the projects included in the Visual Studio solution contains the main application. You can add multiple projects if the package contains multiple executables. In this case, you can choose the entry point of the package by right-clicking the correct application and choosing **Set As Entry Point**.

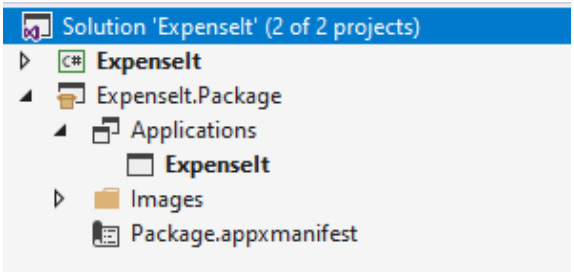


Figure 43: The structure of a Windows project that includes a Windows Application Packaging Project

Debugging

One of the advantages of the Windows Application Packaging Project is that it can be used to debug the packaged version of the application by leveraging the source code of the original desktop one. Right-click the Windows Application Packaging Project, choose **Set as Startup Project**, and then press **F5**. Visual Studio will launch the packaged version, but you'll still be able to put breakpoints in the source code of the Win32 project, add watches, and perform all the common debugging tasks.

Be aware, however, that when you deploy a package directly from Visual Studio, Windows leverages a deployment model called **loose registration**. The application will be executed directly from the build folder and not from the **C:\Program Files\WindowsApps** one, as we

learned in Chapter 1. Since the build folder isn't system-protected, some of the requirements enforced by MSIX packaging that we saw in Chapter 1 (like the inability to write in the installation folder) won't apply.

Before deploying an application packaged with Visual Studio in production, make sure to always test it with the real MSIX package generated using the built-in wizard, which we're going to see in a few moments.

The manifest editor

Visual Studio offers a built-in manifest editor, which will help you to set the identity, the capabilities, etc., of the application using a visual interface.

To access the manifest editor, you just need to double-click the **Package.appxmanifest** file included in the Windows Application Packaging project.

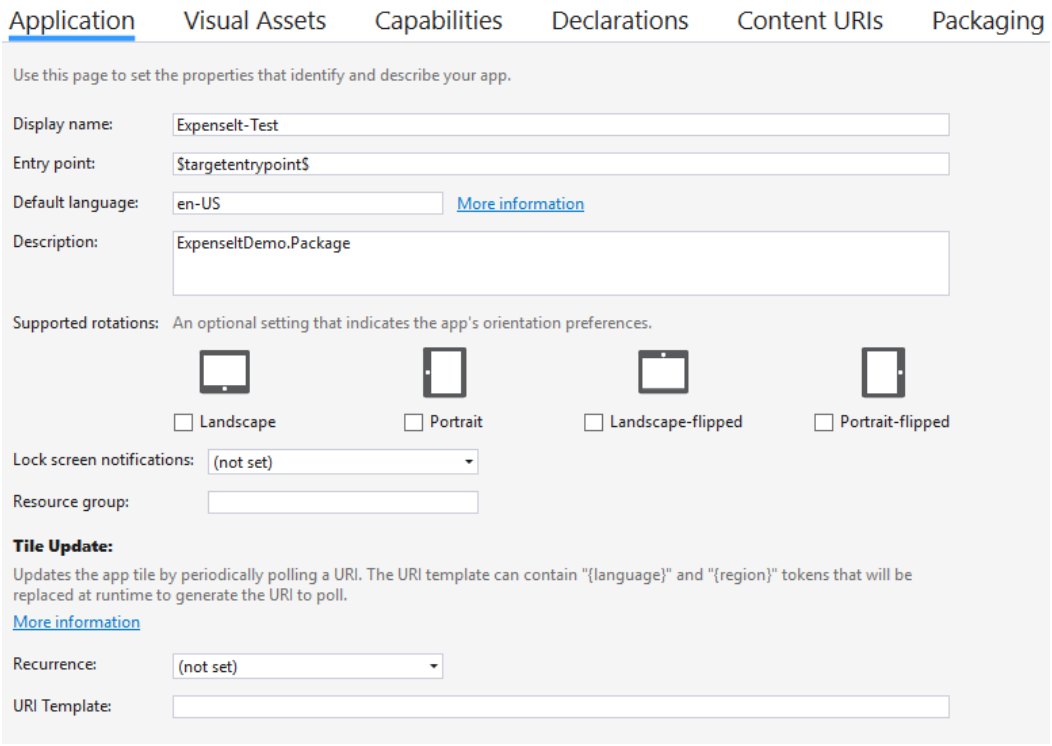


Figure 44: The Visual Studio manifest editor

If you're packaging a classic Windows application, the relevant sections are:

- **Visual Assets**, to help you generate the right icons to use in Windows 10.
- **Packaging,** used to set up the identity by defining the package name, the publisher's name, etc.
- **Declarations,** allowing you to add extension points to integrate your application with Windows 10.

Generating a package

To generate a package, right-click the Windows Application Packaging Project and choose **Store** > **Create app packages**.

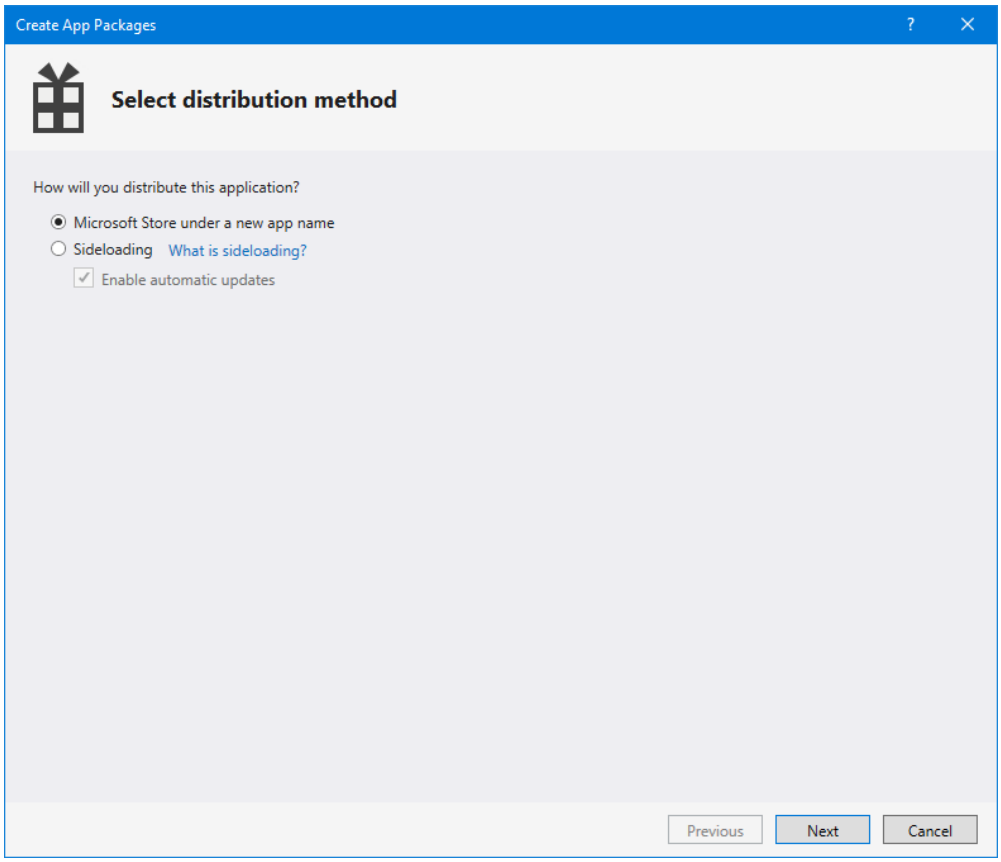


Figure 45: The "Create App Packages" wizard in Visual Studio

Before starting the process, you must choose whether you want to publish the application on the Microsoft Store, or if you're planning sideloading distribution for manual or enterprise deployment. In the first case, you'll be asked to log in with the Microsoft account linked to your developer account, so that Visual Studio can assign the right identity to the application. In the second case, you'll start the packaging process right away. However, you have the option to check the **Enable automatic updates** option, which will add an additional step during the process. We're going to see the distribution options in more detail in Chapter 7.

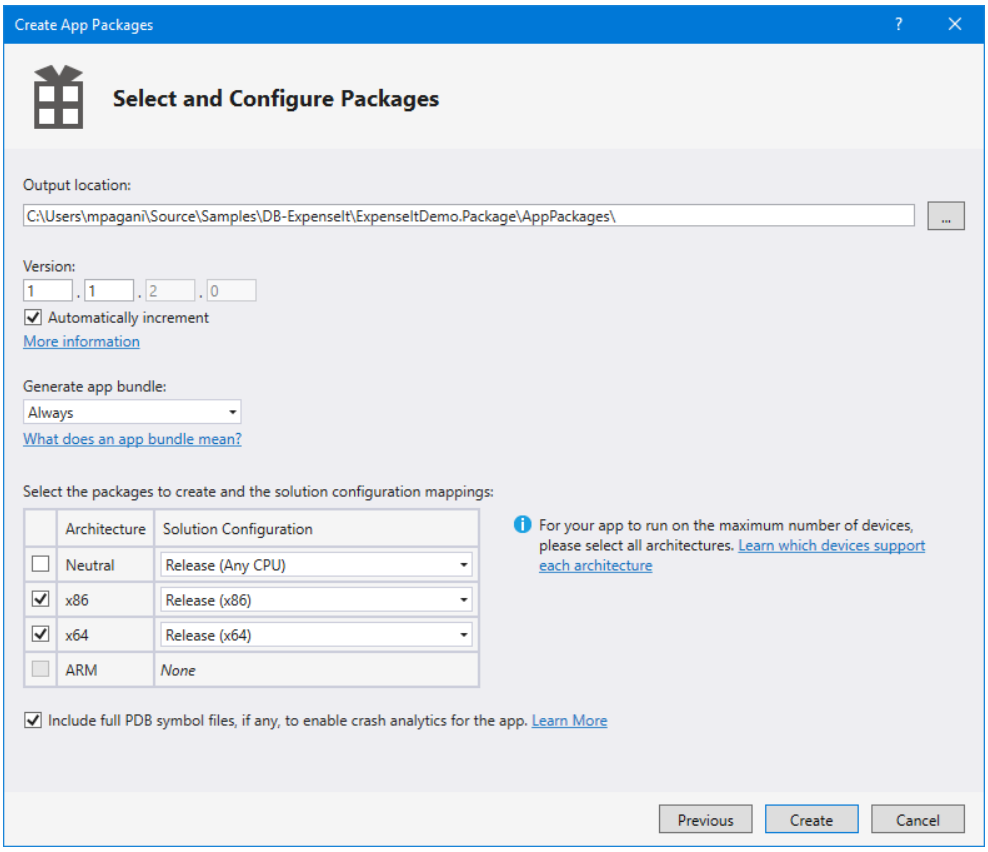


Figure 46: The window to configure the content of the package

In the next section, you will have the option to configure the package you're going to create. At first, you will need to define the folder where you want to generate the MSIX package. By default, it will be a folder called **AppPackages** inside the Windows Application Packaging Project.

Next, you can define the version number and choose whether you want to generate an app bundle. The suggested approach is to use **Always**.

> ***Note: An app bundle is a special package that includes multiple subpackages, one for each potential configuration of the user. With this approach, the tool will generate a main package with the basic app, plus a series of subpackages for the various CPU architectures, scale factors, languages, etc. Windows will install only the components that are the best match for the user's configuration.***

In the last section, you can define which packages you want to create, based on the different architectures you want to support. Since we are packaging a classic desktop application, we can support x86 and x64 architectures. Universal Windows Platform apps, instead, can be compiled for ARM. Using the **Solution Configuration** drop-down menu, we can also choose the configuration we want to use. If you're building a package for distribution, the suggested one to use is **Release**.

Click **Create**. Visual Studio will start building the applications, and it will generate the package. At the end, you will be notified with a confirmation window like the following one.

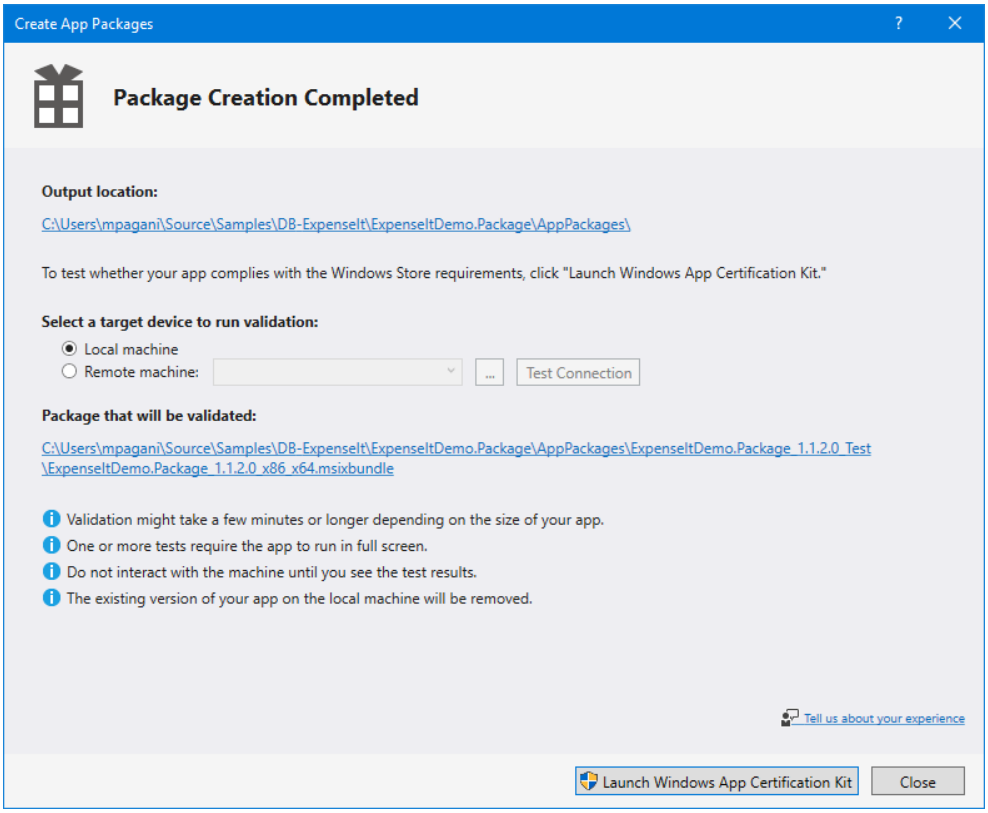


Figure 47: The creation of the package has been completed

By clicking the **Output location,** you will have access to the folder where the package has been created, which will look like the following image.

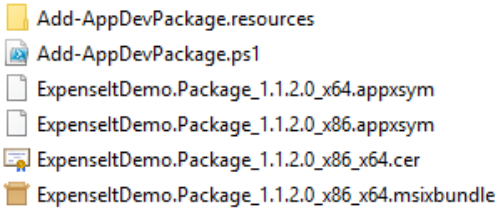


Figure 48: The output of the package creation process by Visual Studio

The main file is the one with the extension **.**msixbundle, which is the generated package. Since we have specified that we wanted to create an app bundle, Visual Studio has generated an .msixbundle file, which contains one .msix package for each supported architecture. Other important files are the test certificate used to sign the package and a PowerShell script, which we can use to automate the deployment of the package. The script will take care of installing the certificate and then deploying the package.

Packaging a standalone application

The Windows Application Packaging Project also supports packaging applications for which you don't own the source code. Thanks to this approach, you'll be able to retain the benefits of the Visual Studio environment, like the powerful manifest editor, or the ability to easily generate a MSIX package for the Store or for sideloading.

The first step to using the Windows Application Packaging Project with a standalone application is to copy, inside it, the binaries that comprise your application, like DLLs, assets, and configuration files.

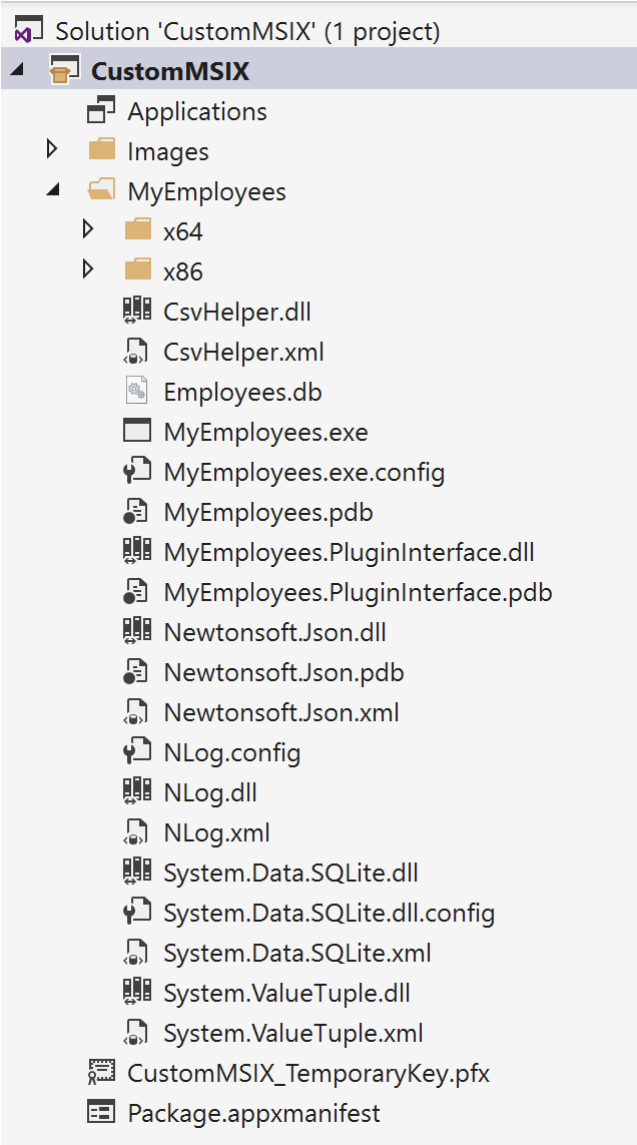


Figure 49: A Windows Application Packaging Project that contains the binaries of a standalone application

The next step is to configure the project to use, as a starting point, the main executable of the application you have just copied. However, in this case we can't use the **Add reference** option, because we don't have a Visual Studio project with the main application to reference. We already have the binaries included in the project itself. We must manually edit the project's file by right-clicking it and choosing **Edit Project.** Visual Studio will open the XML version of the project's file.

Now locate the first **`PropertyGroup`** element, containing information like the target SDK and the minimum supported Windows 10 version. You will need to add a new entry called **`EntryPointExe`**, which must include the relative path to the executable.

Code Listing 20

```
<PropertyGroup>
  <ProjectGuid>b2767d43-1ddb-4aa8-8dee-36d54055dcd2</ProjectGuid>
  <TargetPlatformVersion>10.0.17763.0</TargetPlatformVersion>
  <TargetPlatformMinVersion>10.0.17763.0</TargetPlatformMinVersion>
  <DefaultLanguage>en-US</DefaultLanguage>

<PackageCertificateKeyFile>CustomMSIX_TemporaryKey.pfx</PackageCertificateK
eyFile>
  <EntryPointExe>MyEmployees\MyEmployees.exe</EntryPointExe>
</PropertyGroup>
```

That's it. Now you can just press F5 to deploy the packaged version of the application. Alternatively, you can right-click the project and choose **Store** > **Create app package** to start the wizard, which will generate a MSIX package for distribution.

Creating a registry hive

One of the challenges of using the Windows Application Packaging Project, compared to the MSIX Packaging Tool, is that it doesn't support capturing and deploying a registry hive. The MSIX Packaging Tool is able to capture all the changes performed by the installer on the system, including the creation of keys in the system registry. These keys are then captured in one or more files with the .dat extension, and then deployed inside the virtual registry when the package is installed. This way, the application will be able to read these keys right away at the start.

The Windows Application Packaging Project lacks this capability, since it doesn't perform any capturing process. What if your packaged application needs to deploy one or more registry keys anyway? A solution is to use a library included in the Windows 10 SDK called Offline Registry Library, which is implemented by the **offreg.dll** file.

This library exposes classes and methods to create registry hives. However, compared to the traditional APIs that we find in other development environments (like the **`Registry`** class in the .NET Framework), these hives aren't stored in the system registry, but in an offline file. This file uses the same exact format leveraged by the .dat files created by the MSIX Packaging Tool. Thanks to this library, we can write a small application that creates the hive we need. Then we just need to include it inside the Windows Application Packaging Project to deploy it together with our package.

The Offline Registry Library is native, which means that you must leverage C or C++ to use it. If you're a C# developer, you can leverage an open-source wrapper developed by Michael Bisbjerg, called **OffregLib**. The library is available on GitHub and on NuGet.

You can leverage this library to build an application that generates a registry hive with the required configuration. In this case, we're going to use a console application.

> ***Note: The library shouldn't be installed in the main application you're packaging. It should be leveraged by an application built with the only goal of creating the registry hive you need. The .dat file, in fact, must be copied inside the Windows Application Packaging Project so that it can be deployed at startup. It can't be deployed at runtime.***

Once you have created your project in Visual Studio, just right-click your project, choose **Manage NuGet Packages**, look for the library called **OffregLib**, and install it.

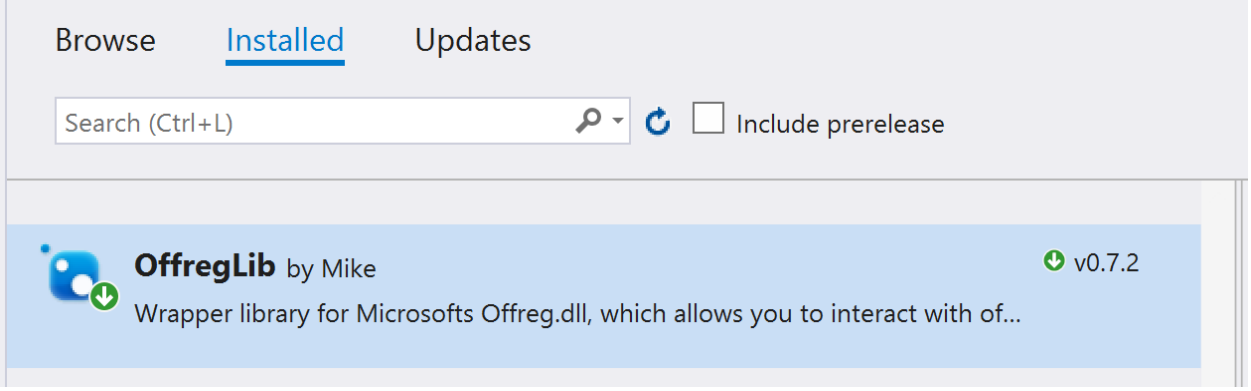


Figure 50: The OffregLib library in NuGet

Using this library is easy. The two relevant methods to use are:

- **`CreateSubKey()`**, which creates a key as child of the selected node.
- **`SetValue()`**, which creates a value inside the key.

For example, let's say that you want to create a virtual registry file that creates the value **FirstRun** inside the **HKLM/SOFTWARE/Contoso/ContosoExpenses** hive. This is the code you need to use.

Code Listing 21

```
using (OffregHive hive = OffregHive.Create())
```

```
{
    using (var registryEntry = hive.Root.CreateSubKey("REGISTRY"))
    {
        using (var machineEntry = registryEntry.CreateSubKey("MACHINE"))
        {
            using (OffregKey key = machineEntry.CreateSubKey("SOFTWARE"))
            {
                using (var subKey = key.CreateSubKey("Contoso"))
                {
                    using (var finalKey =
subKey.CreateSubKey("ContosoExpenses"))
                    {
                        // Set a value to a string.
                        finalKey.SetValue("FirstRun", "True");
                    }
                }
            }
        }
    }
}
```

You create the starting point of a hive using the static **`OffregHive.Create()`** method. From there, you can call the **`CreateSubKey()`** method to create a new item as child of the current one. Each child must be created by calling the **`CreateSubKey()`** method on the child created in the previous iteration. The library, in fact, doesn't support a way to create a full path with a single operation.

Note: In order to work properly as a virtual registry for an MSIX package, the starting node should be called REGISTRY.

Once you have reached the desired path, you can finally call the **`SetValue()`** method to create the desired value, passing as parameters the key's name and its value. As a last step, you can save the offline registry file by invoking the **`SaveHive()`** method on the starting **`OffregHive`** object. This is the full definition of the console application.

Code Listing 22

```
namespace RegistryHive.Writer
{
    class Program
    {
        static void Main(string[] args)
        {
            using (OffregHive hive = OffregHive.Create())
            {
                using (var registryEntry =
hive.Root.CreateSubKey("REGISTRY"))
```

```
                {
                    using (var machineEntry =
registryEntry.CreateSubKey("MACHINE"))
                    {
                        using (OffregKey key =
machineEntry.CreateSubKey("SOFTWARE"))
                        {
                            using (var subKey =
key.CreateSubKey("Contoso"))
                            {
                                using (var finalKey =
subKey.CreateSubKey("ContosoExpenses"))
                                {
                                    // Set a value to a string
                                    finalKey.SetValue("FirstRun",
"True");
                                }
                            }
                        }
                    }
                }

                // Delete the file if it exists - Offreg requires files
not to exist.
                if (File.Exists("Registry.dat"))
                    File.Delete("Registry.dat");

                // Save it to disk - version 5.1 is Windows XP. This is a
form of compatibility option.
                // Read more here: http://msdn.microsoft.com/en-
us/library/ee210773.aspx
                hive.SaveHive("Registry.dat", 5, 1);
            }
        }
    }
}
```

Now you can just launch your console application and, at the end of the process, you will find a file called **Registry.dat** in the build output folder of Visual Studio.

The final step is simply to copy this file inside the Windows Application Packaging Project of your solution so that it can be included in the generated MSIX package.

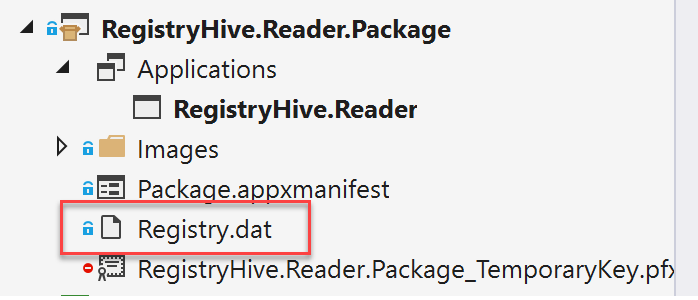


Figure 51: The Registry.dat file included in the Windows Application Packaging Project

Now your packaged application will be able to find the **FirstRun** key inside the registry hive **HKLM/SOFTWARE/Contoso/ContosoExpenses,** even if this key doesn't exist in your system registry.

Integrating with Windows 10

Many Windows applications have the requirement to integrate with the operating system. They may allow the creation of files, which should be opened with a double-click in File Explorer; or maybe they must be configured to run at startup; or again, they may need to set up some specific rules for the Windows firewall.

All these scenarios are typically controlled using the system registry and delegated to the installer of the application. The installer takes care of creating the keys required to register a specific file type association, or to add a context menu in File Explorer, or to expose a COM object to other applications. The challenge is that, most of the time, you must rely on a third-party tool to configure this integration, since the process isn't straightforward. You are required to create some special keys in the **HKEY_LOCAL_MACHINE\SOFTWARE\Classes** hive and to deal with concepts like ProgId and CLSID, which are typical of the COM world. Third-party installer authoring tools make this approach easier, since they typically provide a user interface that will create, under the hood, the required registry keys for you.

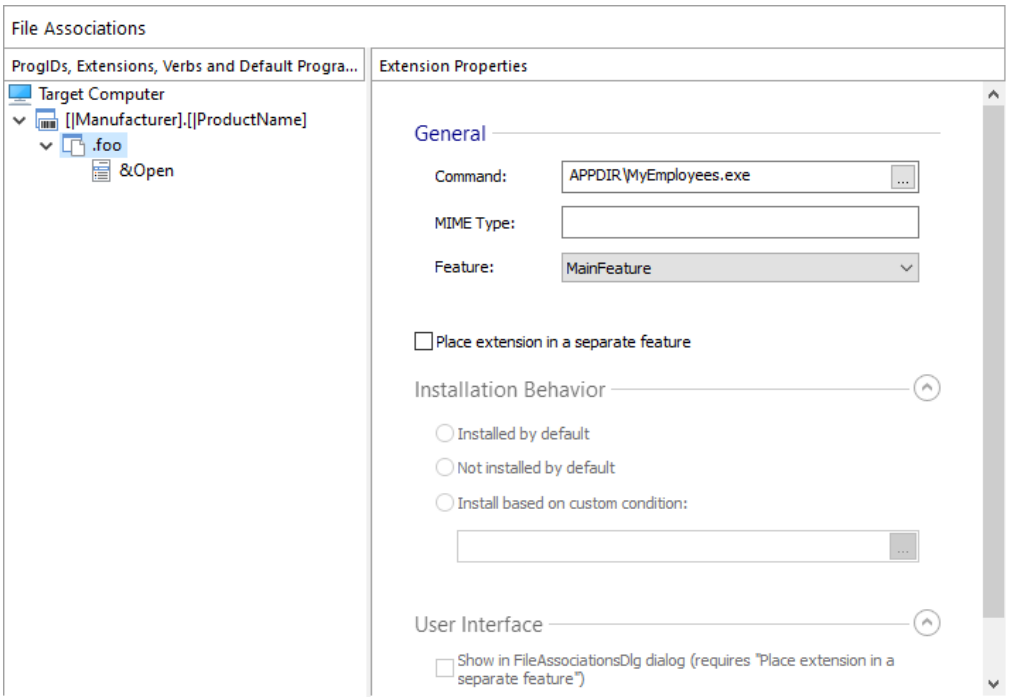


Figure 52: The dialog in Advanced Installer for creating a file-type association

MSIX packaging makes the integration even easier by moving the configuration of Windows 10 extensions from the registry to the manifest. Supporting a file type association or a startup task is nothing more than clicking a couple of options in the Visual Studio's manifest editor or, if you prefer a manual approach, adding an XML entry to the manifest.

Let's see some examples of the most common Windows integration options.

Registering a file-type association

File-type associations can be easily configured using the manifest editor included in Visual Studio. Just double-click the **Package.appxmanifest** file inside the Windows Application Packaging Project and move to the **Declarations** tab. A drop-down menu will list all the available integration options. Choose **File Type Associations** and click **Add.**

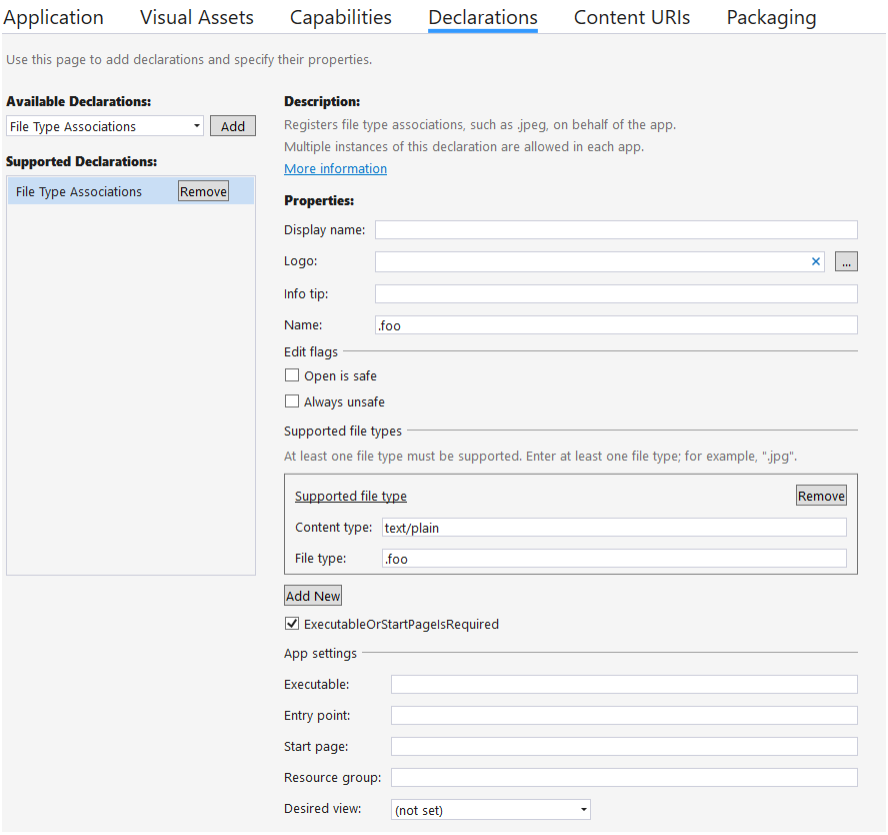


Figure 53: How to register a file-type association

The **Name** field is required and must match the extension you want to register. You can register as many extensions as you want by simply clicking **Add New**. By default, the template comes with a single association. For each of them, the required field to fill is **File type**, which must contain the extension you're registering, as well. **Content type** is optional, but it's helpful to let Windows understand which kind of content is stored inside the file.

By default, Windows will apply the main logo of the application to the files with the selected extensions. You can apply a custom logo, if you prefer, by choosing a different image in the **Logo** field.

If you right-click the **Package.appxmanifest** file and choose **View code**, you will be able to see how the options you have chosen have been translated into XML. Inside the **`Application`** entry you will find the following item.

Code Listing 23

```
<Extensions>
  <uap:Extension Category="windows.fileTypeAssociation">
    <uap:FileTypeAssociation Name=".foo">
      <uap:SupportedFileTypes>
        <uap:FileType ContentType="text/plain">.foo</uap:FileType>
      </uap:SupportedFileTypes>
    </uap:FileTypeAssociation>
```

```
  </uap:Extension>
</Extensions>
```

From a developer point of view, the activation works in the same way as for a traditional Win32 application. The application will be opened with, as first argument, the full path of the file selected by the user.

Enabling a context menu

File-type associations are often coupled with a context menu. When users right-click a file that is supported by your application, they will have access to a broader set of options. This feature can be configured using the same **`windows.fileTypeAssociation`** extension we saw in the previous section. However, the manifest editor doesn't support it, so we need to manually add it in the XML by right-clicking the **Package.appxmanifest** file and choosing **View code**. This is what the extension looks like.

Code Listing 24

```
<?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"
xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
IgnorableNamespaces="uap3 desktop">
  <Applications>
    <Application>
      <Extensions>
        <uap:Extension Category="windows.fileTypeAssociation">
          <uap:FileTypeAssociation Name=".foo">
            <uap:SupportedFileTypes>
              <uap:FileType ContentType="text/plain">.foo</uap:FileType>
            </uap:SupportedFileTypes>
           <uap2:SupportedVerbs>
             <uap3:Verb Id="Edit" Parameters="/e
&amp;quot;%1&amp;quot;">Edit</uap3:Verb>
             <uap3:Verb Id="Print" Parameters="/p
&amp;quot;%1&amp;quot;">Print</uap3:Verb>
            </uap2:SupportedVerbs>
          </uap:FileTypeAssociation>
        </uap3:Extension>
      </Extensions>
    </Application>
  </Applications>

</Package>
```

After the **SupportedFileTypes** entry, which contains the list of supported extensions, we add a new section called **SupportedVerbs**, which can contain one or more **Verb** items. Each **Verb** item is an option displayed in the context menu when you right-click a file with the target extension, such as .foo. Each entry has a unique **Id** and **Parameters**, which specify the set of arguments that will be passed to the application when you select this option. Thanks to these parameters, the application will be able to understand the context and perform the most appropriate action.

Register a global alias

Some applications need to register as a global alias so that you can invoke them from any File Explorer or command prompt instance. This extension isn't supported by the visual editor in Visual Studio, so you will need add the following entry to the XML. The extension is called **windows.appExecutionAlias**.

Code Listing 25

```
<?xml version="1.0" encoding="utf-8"?>
<Package
xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
xmlns:desktop="http://schemas.microsoft.com/appx/manifest/desktop/windows10
"
IgnorableNamespaces="uap3 desktop">
  <Applications>
    <Application>
      <Extensions>
        <uap3:Extension Category="windows.appExecutionAlias"
Executable="MyEmployees.exe" EntryPoint="Windows.FullTrustApplication">
          <uap3:AppExecutionAlias>
            <desktop:ExecutionAlias Alias="ME.exe" />
          </uap3:AppExecutionAlias>
        </uap3:Extension>
      </Extensions>
    </Application>
  </Applications>
</Package>
```

In this example, we're setting up an alias called **ME.exe**, which will launch the **MyEmployees.exe** executable included inside the package. After deploying this package, you would be able to just open a command prompt and, regardless of the folder in which you are located, type **ME** and press **Enter** to start the application.

Launch the application at startup

Another common requirement for applications is to start when the user logs in, so that they are immediately ready to handle tasks for the user. This goal can be achieved with the

windows.startupTask extension, which isn't supported by the visual editor. As such, you will need to edit the XML behind the **Package.appxmanifest** file and add the following snippet.

Code Listing 26

```
<Package

xmlns:desktop="http://schemas.microsoft.com/appx/manifest/desktop/windows10
"
  IgnorableNamespaces="desktop">
  <Applications>
    <Application>
      <Extensions>
        <desktop:Extension
          Category="windows.startupTask"
          Executable="MyEmployees.exe"
          EntryPoint="Windows.FullTrustApplication">
          <desktop:StartupTask
            TaskId="MyEmployees"
            Enabled="true"
            DisplayName="My Employees App" />
        </desktop:Extension>
      </Extensions>
    </Application>
  </Applications>
</Package>
```

The extension itself just needs an **Executable** attribute, which specifies the executable included inside the package you want to launch at startup. Then you need to define a **StartupTask** entry with the following properties:

- **TaskId**, the unique identifier of the task.
- **Enabled**, which defines the default status of the task.
- **DisplayName**, the name displayed in Task Manager in the list of startup services.

Note that this feature follows the original Universal Windows Platform philosophy, which is "user first." The goal is to make sure that all the operations that might jeopardize the user experience are made known to the user. As such, a startup task is initialized only the first time the application is launched. If the application is deployed but never started, the startup task won't be enabled.

Transition your users

When you start packaging your application as MSIX, there might be an unintended negative effect. Since Windows isn't able to determine the identity of a classic application, compared to one packaged with MSIX, users won't be blocked from installing the MSIX package of an application they already have deployed on their machine with a traditional installer. This can lead to multiple issues from a user experience point of view. In the best case, users will be confused because they will have two identical versions of the same application running side by

side; in the worst case, users can suffer data loss if both applications are reading and writing from the same data source.

Thanks to a set of special extensions, developers have the opportunity to facilitate this transition by making sure that all the existing shortcuts and file-type associations are redirected to the MSIX-packaged version instead of the classic one. Let's see these extensions in detail.

Shortcuts transition

This special extension allows Windows to automatically redirect all the existing shortcuts (in the Start menu, on the desktop, in the list of programs installed on your machine, etc.) to open the MSIX-packaged version of the application instead of the classic one. Let's take a look at a sample snippet.

Code Listing 27

```
<Package

xmlns:rescap3="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities/3"
  IgnorableNamespaces="rescap3">
  <Applications>
    <Application>
      <Extensions>
        <rescap3:Extension Category="windows.desktopAppMigration">
          <rescap3:DesktopAppMigration>
            <rescap3:DesktopApp
ShortcutPath="%USERPROFILE%\Desktop\[my_app].lnk" />
            <rescap3:DesktopApp
ShortcutPath="%APPDATA%\Microsoft\Windows\Start Menu\Programs\[my_app].lnk"
/>
          </rescap3:DesktopAppMigration>
        </rescap3:Extension>
      </Extensions>
    </Application>
  </Applications>
</Package>
```

The extension is called **windows.DesktopAppMigration**, and it includes a section called **DesktopAppMigration**. Inside it you can add as many **DesktopApp** entries as you need, one for each shortcut created by default by the traditional installer. The example in Code Listing 27 migrates two shortcuts: the one created on the desktop, and the one created in the Start menu. The full path of these shortcuts can be obtained by searching for the ones created by the traditional installer, right-clicking it, and choosing **More** > **Open file location**.

Once you have deployed the MSIX package with such a configuration, clicking the shortcuts in the Start menu, or the ones created on the desktop by the traditional installer, will open the MSIX-packaged version.

File type association transition

We have just seen how an application can be registered to open one or more file types based on their extensions. In such a scenario, you may already have the classic version of your Windows application registered for these file types. The goal of this special extension is to make sure that, once the MSIX package is deployed, double-clicking one of these files will open it instead of the classic version that was deployed with a traditional installer.

Here is the relevant snippet.

Code Listing 28

```
<Package
  xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
  xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"

xmlns:rescap3="http://schemas.microsoft.com/appx/manifest/foundation/window
s10/restrictedcapabilities/3"
  IgnorableNamespaces="uap3, rescap3">
  <Applications>
    <Application>
      <Extensions>
        <uap:Extension Category="windows.fileTypeAssociation">
          <uap3:FileTypeAssociation Name=".foo">
            <rescap3:MigrationProgIds>
              <rescap3:MigrationProgId>Foo.Bar.1</rescap3:MigrationProgId>
              <rescap3:MigrationProgId>Foo.Bar.2</rescap3:MigrationProgId>
            </rescap3:MigrationProgIds>
          </uap3:FileTypeAssociation>
        </uap:Extension>
      </Extensions>
    </Application>
  </Applications>
</Package>
```

To support the file-type transition, you need to find the programmatic identifier (ProgID), which is used by the classic application to register the file-type association. This is the special registry key we mentioned at the beginning of the section. You can locate it by analyzing the original installer of your application or by exploring **HKEY_LOCAL_MACHINE\SOFTWARE\Classes** in the Registry Editor. In the following screenshot, for example, you can see all the ProgIDs registered by Microsoft Excel.

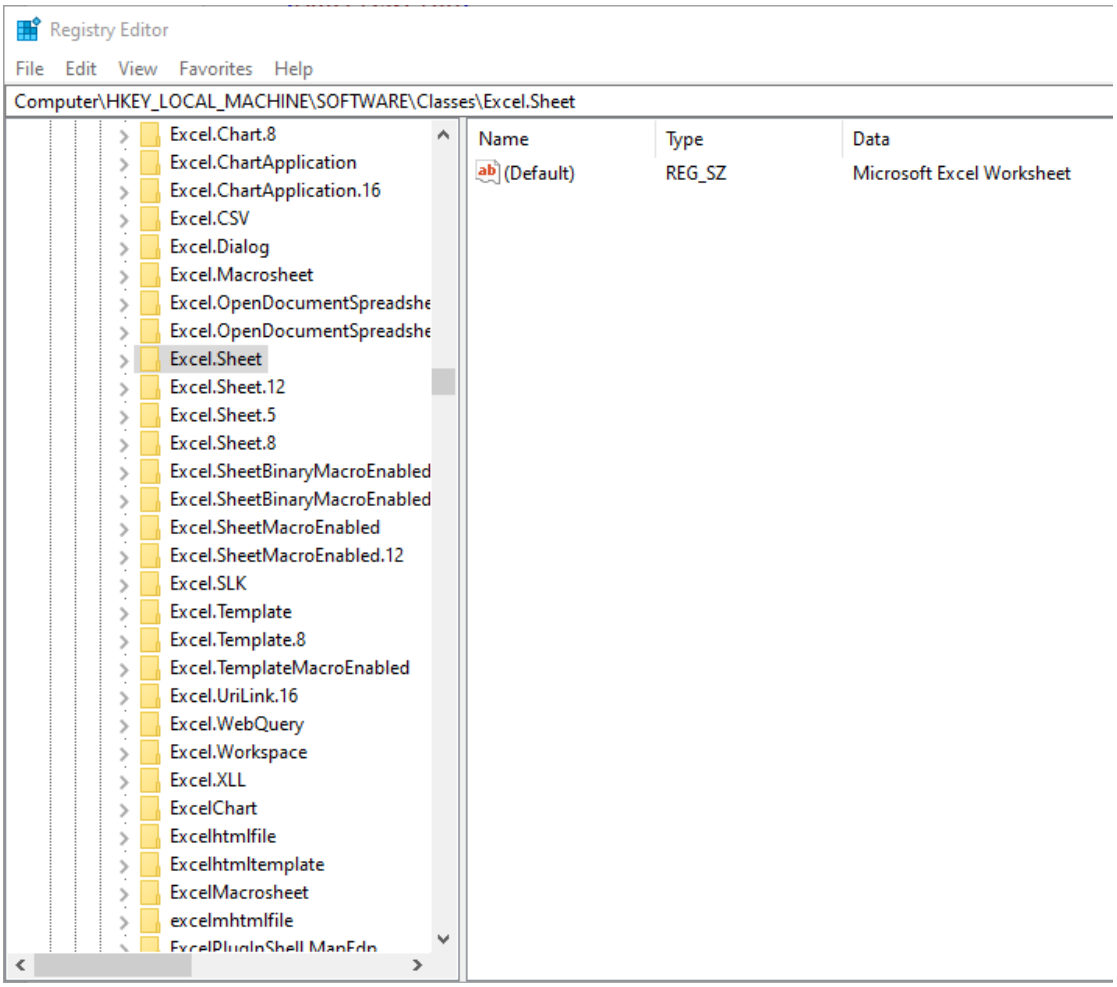


Figure 54: The ProgIDs registered by Microsoft Excel to handle different file types

Other extensions

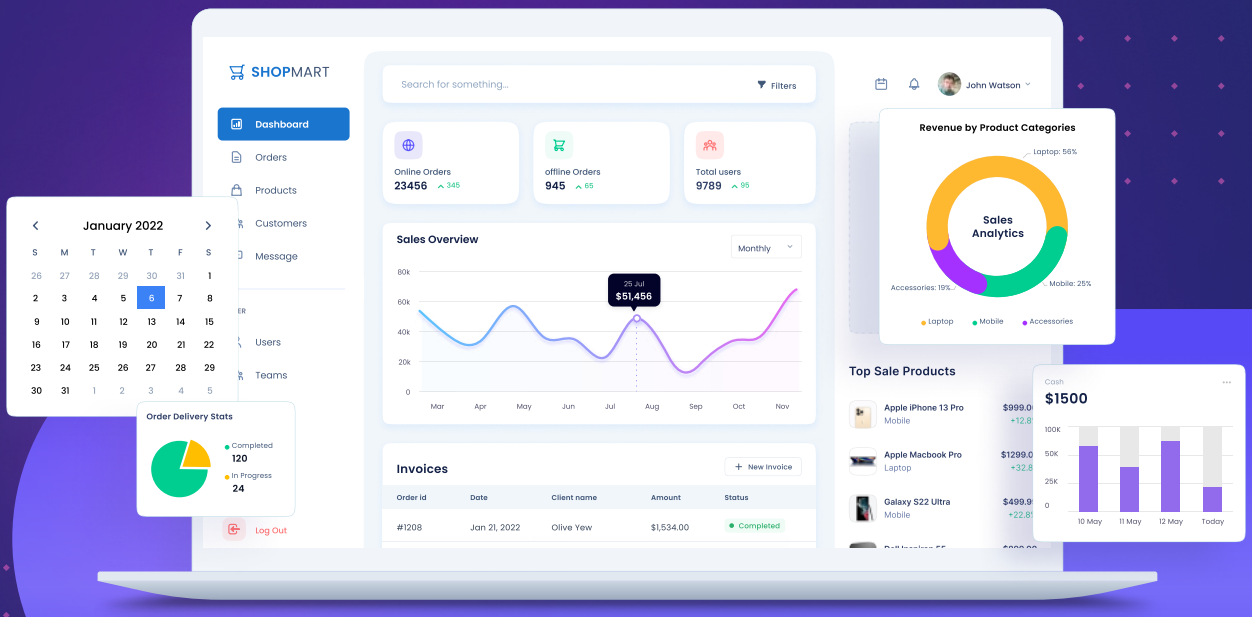
The manifest of a MSIX-packaged application supports many more extensions, like:

- Registering a protocol to launch the application.
- Setting one or more firewall rules.
- Exposing COM objects to other applications.
- Handling image preview in File Explorer.
- Exposing file properties in the details pane.

You can find the documentation on how to enable all these extensions here.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 6 Move Your Application Forward with MSIX

So far, we have learned how MSIX is a great way to improve the deployment of your existing Windows applications. However, if you're a developer, MSIX is a lot more than just a new deployment technology. It opens up many exciting opportunities to move your application forward, to enhance it, and to access all the new features that have been added to Windows 10 in the past years, like support for biometric authentication, toast notifications, Timeline, and new UI controls.

Evolving the ecosystem

When the Universal Windows Platform was launched, its focus was to provide a new modern framework to help developers build new experiences and compelling applications that can leverage all the latest enhancements in modern development, like touch and inking support, first-class support for the cloud, and speech recognition.

These features were immediately very compelling for developers, but there were many requirements to satisfy, especially in the enterprise space. These requirements typically belong to the traditional Windows world, like the ability to run at admin level and deploy applications using SSCM, Intune, or a website. Additionally, these applications usually have a long history and a big development codebase, so rewriting them using a new technology wasn't feasible in many scenarios.

With the latest enhancements in the Windows app-development story, this barrier between the modern and desktop worlds is becoming thinner and thinner. The Universal Windows Platform is still a great starting point if you want to build applications that target new experiences, but now it's also a framework that you can leverage from your existing desktop applications. You can easily add a reference to the Universal Windows Platform in your desktop application and use features that, in the past, were reserved for modern apps, like Windows Hello, Timeline, or Live Tiles. And thanks to a new technology called XAML Islands, you can add controls that belong to the Universal Windows Platform inside your existing Windows Forms or WPF UI. This way, you'll be able to leverage the new UI features (fluent design, enhanced accessibility, touch support, etc.) without rewriting your application from scratch.

MSIX is a great starting point for enhancing your Windows applications. Many APIs of the Universal Windows Platform can be consumed by a plain Win32 application, but some of them require the application to have an identity; otherwise, they won't work, or they will return exceptions. The identity is a concept that typically belongs to Universal Windows Platform applications, but thanks to MSIX packaging, you can also enable it for classic desktop applications.

Some of the features that require an identity are:

- Notifications

- Live Tiles and badges
- Background tasks
- Share target
- Storage and file pickers
- App services

Adding features from the Universal Windows Platform

In the past, adding a reference to the Universal Windows Platform in a Win32 app wasn't a straightforward task, since you needed to manually look for the Windows metadata files on the hard disk and add them. Now the process is really easy, thanks to a dedicated NuGet package, which you just need to install in your WPF or Windows Forms application.

The package is called **Microsoft.Windows.SDK.Contracts**, and it comes in multiple versions, based on the Windows 10 SDK version you want to target.

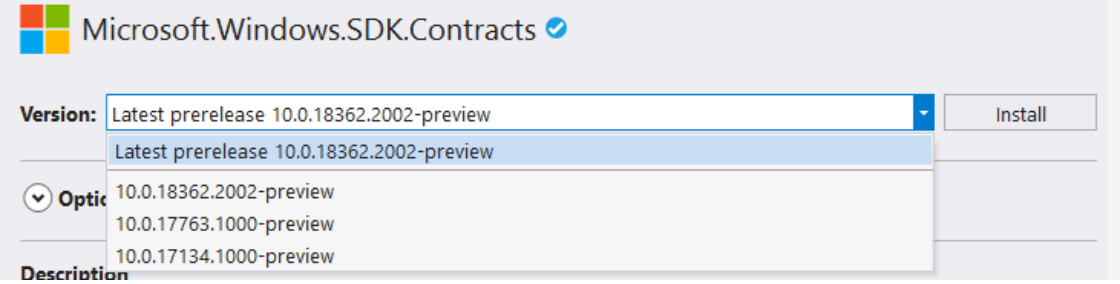


Figure 55: The various versions of the Microsoft.Windows.SDK.Contracts package

Once the package has been installed, you will be able to access the whole Universal Windows Platform surface in your application, which is offered through the **`Windows.*`** namespace.

In most cases, using APIs from the Universal Windows Platform in your classic desktop applications isn't any different from leveraging them in a full UWP app. For most of the features, you can just reference the standard documentation. You can also read my previous books published by Syncfusion, *UWP Succinctly* and *More UWP Succinctly*, if you want to learn more about the features offered by the Universal Windows Platform.

However, as already mentioned, some of the features require the application to have an identity before being used. Let's see an example.

Adding a toast notification

One of the features included in the Universal Windows Platform is the ability to send toast notifications. Windows offers a rich notification model that supports text, images, and actions.

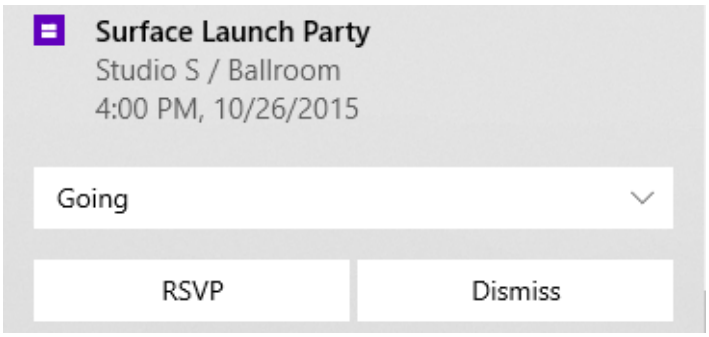


Figure 56: An actionable toast notification

Toast notifications are defined using an XML schema, which describes all the relevant properties like the title, text, icon, and actions. The following sample shows how to send a toast notification within an application.

Code Listing 29

```
public void ShowNotification(string description, double amount)
{
    string xml = $@"<toast>
    <visual>
        <binding template='ToastGeneric'>
            <text>Expense added</text>
            <text>Description: {description} - Amount: {amount} </text>
        </binding>
    </visual>
</toast>";

    XmlDocument doc = new XmlDocument();
    doc.LoadXml(xml);

    ToastNotification toast = new ToastNotification(doc);
    ToastNotificationManager.CreateToastNotifier().Show(toast);
}
```

In order to use these APIs, you will need to reference the following namespaces in your class.

Code Listing 30

```
using Windows.Data.Xml.Dom;
using Windows.UI.Notifications;
```

Once you have defined the XML schema, you need to load it inside a **`XmlDocument`** object by using the **`LoadXml()`** method. Then you can wrap it inside a **`ToastNotification`** object and, in the end, show it to the user by calling the **`Show()`** method exposed by the **`ToastNotificationManager.CreateToastNotifier()`** object.

Tip: If you prefer to use classes and C# to build your notification, you can use a library on NuGet called Microsoft.Toolkit.UWP.Notifications.

As you can see, these APIs belong to the two **`Windows.*`** namespaces declared in Code Listing 30, which means they are part of the Universal Windows Platform. If you invoke this code from an unpackaged Windows Forms or WPF application, you will see the following exception.

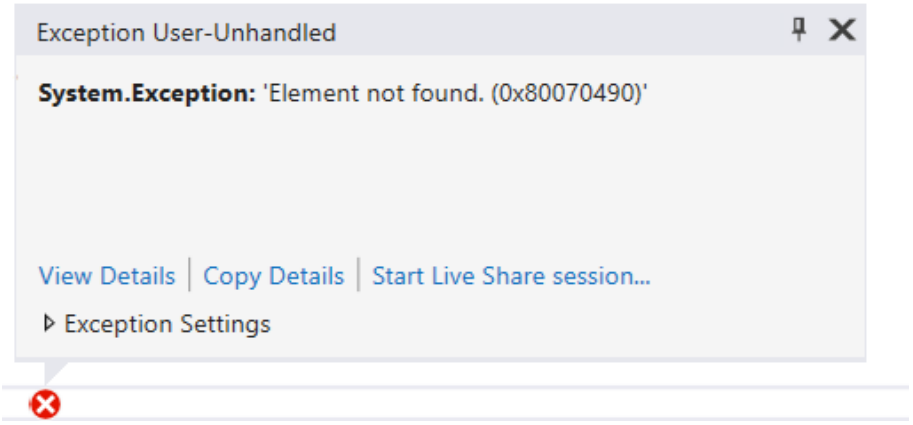


Figure 57: The exception raised when you send a toast notification from a Windows application that isn't packaged

This is a typical symptom of an API that requires an identity and, as such, it can't be used from a Windows application without packaging it first. The solution is quite straightforward: you need to add a Windows Application Packaging Project to your solution and, by following the guidance described in Chapter 5, package your application with MSIX.

Once the application has been packaged and deployed, the previous code will no longer trigger an exception. The notification will be correctly displayed.

Adding a UWP UI control

Windows 10 version 1809 introduced XAML Islands, a feature for hosting controls from the Universal Windows Platform in Win32 applications. However, the technology was just added in preview. Windows 10 version 1903 is marked the first official release.

The XAML Islands architecture

XAML Islands relies on a new set of APIs included in Windows 10 called **XAML Bridge Source WinRT APIs**. The “magic” is powered by two new system APIs: **`WindowsXamlManager`** and **`DesktopWindowXamlSource`**.

The **`WindowsXamlManager`** takes care of initializing the UWP XAML Framework inside the current thread of a non-Win32 Desktop app so that you can start adding UWP controls to it.

The **`DesktopWindowXamlSource`** is the actual instance of your Islands content. It has a **`Content`** property, which you can instantiate and set with the control you want to render.

With an instance of the **`DesktopWindowXamlSource`** class, you can attach its HWND (Win32 Windows Handle) to any parent HWND you want from your native Win32 app. This virtually enables any framework that exposes HWND to host a XAML Island, including third-party technologies like Java and Delphi. However, when it comes to WPF and Windows Forms applications, you don’t have to manually do that, thanks to the **Windows Community Toolkit**, which already wraps these classes into ready-to-be-used controls.

The Windows Community Toolkit is an open-source project maintained by Microsoft and driven by the community, and includes many custom controls, helpers, and services to speed up the development of Windows applications. Starting from version 5.0, the toolkit includes four packages to enable XAML Island:

- One called **WindowsXamlHost**. It's a generic control that can host any UWP control, either custom or native. It comes in two variants: **Microsoft.Toolkit.Wpf.UI.XamlHost** for WPF and **Microsoft.Toolkit.Forms.UI.XamlHost** for Windows Forms.

- One called **Controls**. It includes wrappers for first-party controls like **`Map`** or **`InkCanvas`**. Thanks to these controls, you'll be able to leverage them as if they're native WPF or Windows Forms controls, including direct access to the exposed properties and binding support. Also in this case, it comes in two variants: **Microsoft.Toolkit.Wpf.UI.Controls** for WPF, and **Microsoft.Toolkit.Forms.UI.Controls** for Windows Forms.

All these packages are available on NuGet.

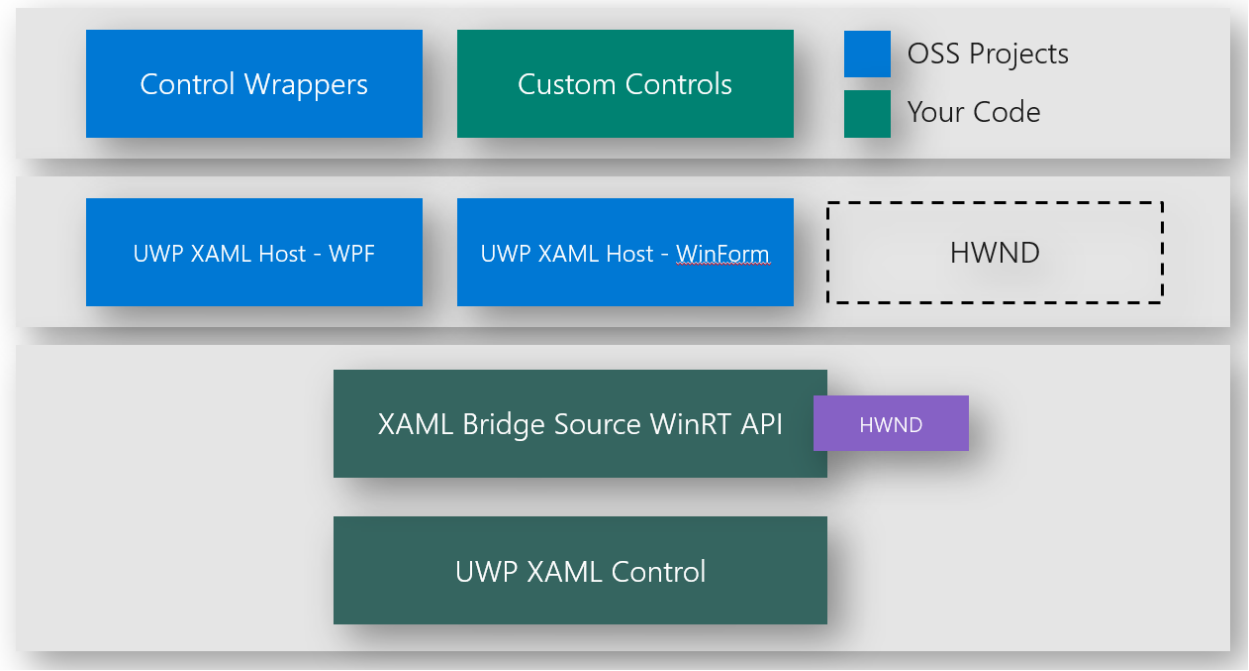


Figure 58: The XAML Islands architecture

Using one of the available wrappers

The purpose of the wrappers is to make it easier for WPF and Windows Forms developers to include some of the most commonly requested controls. These are the ones currently available in the Windows Community Toolkit:

- **`WebView`**: A modern WebView that you can use to embed HTML content in your application. It's based on the Edge rendering engine, as opposed to the built-in .NET Framework **`WebBrowser`** control, which is still based on the Internet Explorer engine.
- **`MapControl`**: A map control with full support for touch, inking, offline capabilities, traffic, etc.
- **`MediaPlayerElement`**: An advanced control to play media items with support for captions, smooth streaming, modern codecs, etc.
- **`SwapChainPanel`**: A hosting surface to render DirectX content as XAML UI.
- **`InkCanvas`**: A canvas with full support for inking, with text and shape recognition.

Thanks to these wrappers, you can add one of these controls from the Universal Windows Platform without using any Windows Runtime concept or type. You will just leverage properties and events like you would do with a native .NET Framework control.

Let's see an example by adding an **`InkCanvas`** control to a WPF application. The first step is to add the relevant NuGet package, which, in the case of a WPF application, is Microsoft.Toolkit.Wpf.UI.Controls**.**

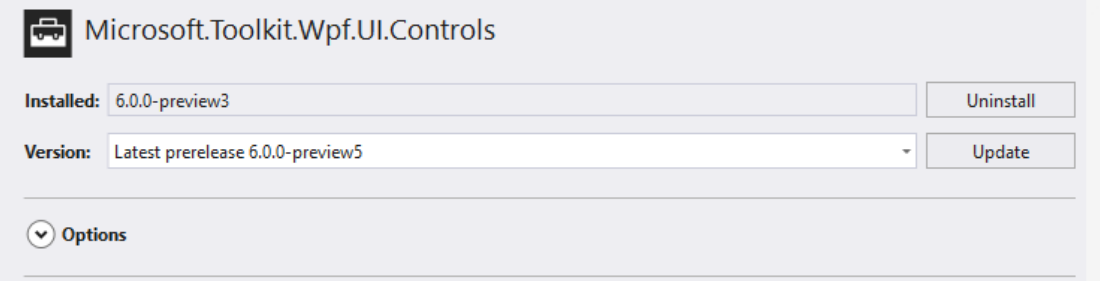


Figure 59: The NuGet package to add XAML Islands wrapped controls to a WPF application

Once the package is installed, you will need to add a namespace to the XAML page where you want to add the control. Being a third-party control, it can't be referenced like you would do with a native control, like **`TextBox`** or **`Grid`**. This is the namespace to add.

Code Listing 31

```
xmlns:toolkit="clr-
namespace:Microsoft.Toolkit.Wpf.UI.Controls;assembly=Microsoft.Toolkit.Wpf.
UI.Controls"
```

Thanks to this namespace, you can reference all the wrapped controls included in the toolkit. For example, this is how you can add the **`InkCanvas`** control to your page.

Code Listing 32

```
<toolkit:InkCanvas x:Name="Signature" />
```

Once the control has been added, you can interact with it in your code like you would do with a native WPF control. For example, let's say you want to change the interaction type so that the user can also interact with the canvas using the mouse or the touch screen. By default, in fact, the **`InkCanvas`** control works only with the pen. It's enough to add the following snippet when the WPF window is initialized.

Code Listing 33

```
public partial class ExpenseDetail : Window
{
    public ExpenseDetail()
    {
        InitializeComponent();
```

```
        Signature.InkPresenter.InputDeviceTypes =
CoreInputDeviceTypes.Mouse | CoreInputDeviceTypes.Pen;
    }
}
```

InkPresenter is a wrapped property exposed by the **InkCanvas** control, which accepts one or more values of the **CoreInputDeviceTypes** enumerator. They're both types coming from the Universal Windows Platform, but, being wrapped for you, they can be used without any specific knowledge about this framework.

In the following figure, you can see the actual outcome.

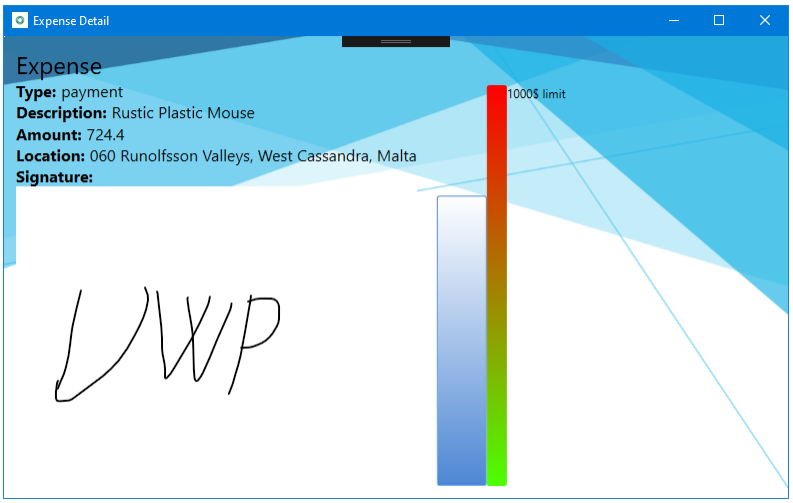


Figure 60: A WPF window with an InkCanvas control

Hosting a generic control

The XAML Islands feature also offers the opportunity to host any control from the Universal Windows Platform by using the **WindowsXamlHost** control included in the Windows Community Toolkit. To add this control to your application, you will need to install another NuGet package, called Microsoft.Toolkit.Wpf.UI.XamlHost.

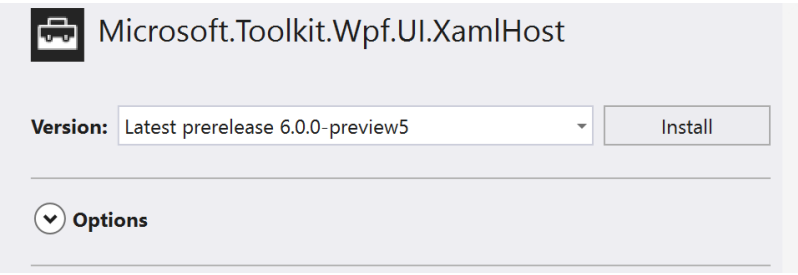


Figure 61: The NuGet package to add the generic UWP control host in a WPF application

Once the library has been added, you need to add to the WPF XAML the declaration of the namespace that contains the control.

Code Listing 34

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

Then you can add the **WindowsXamlHost** to your page, like in the following sample.

Code Listing 35

```
<xamlhost:WindowsXamlHost
InitialTypeName="Windows.UI.Xaml.Controls.CalendarView"
x:Name="CalendarUwp" ChildChanged="CalendarUwp_ChildChanged" />
```

The key property of the control is **InitialTypeName**, which must be set with the full signature of the control you want to host. In the previous sample, we're hosting a control built-in with the Universal Windows Platform, the CalendarView. However, you aren't limited to built-in controls. You can also reference a custom control or a control coming from a third-party vendor.

The second important part is the usage of the **ChildChanged** event. The main difference between this control and a wrapped control is that **WindowsXamlHost** can virtually host any control. As such, it can't expose specific properties or events. Thanks to the **ChildChanged** event, you can intercept the rendering of the hosted control and set properties or subscribe to events that are specific to the control you want to display. Here is an example implementation of this event.

Code Listing 36

```
private void CalendarUwp_ChildChanged(object sender, System.EventArgs e)
{
    WindowsXamlHost windowsXamlHost = (WindowsXamlHost)sender;

    Windows.UI.Xaml.Controls.CalendarView calendarView =
        (Windows.UI.Xaml.Controls.CalendarView)windowsXamlHost.Child;

    if (calendarView != null)
    {
```

```
        calendarView.SelectedDatesChanged += (obj, args) =>
        {
            if (args.AddedDates.Count > 0)
            {
                //do something
            }
        };

        calendarView.MaxDate = DateTimeOffset.Now;
    }
}
```

The **WindowsXamlHost** control offers a property called **Child** that contains a reference to the control being hosted. This control matches the one you have defined with the **InitialTypeName** property, so you can cast it using the same type. In the previous example, we're casting the **Child** property to the **Windows.UI.Xaml.Controls.CalendarView** control, which was set as **InitialTypeName** in the XAML.

Once we have a reference to the hosted control, we can access its events and properties. In the previous snippet, we subscribe to the **SelectedDatesChanged** event (which is triggered when the user selects a date from the calendar), and we set the **MaxDate** property (to set a maximum selectable date).

This is the final result in the WPF application.

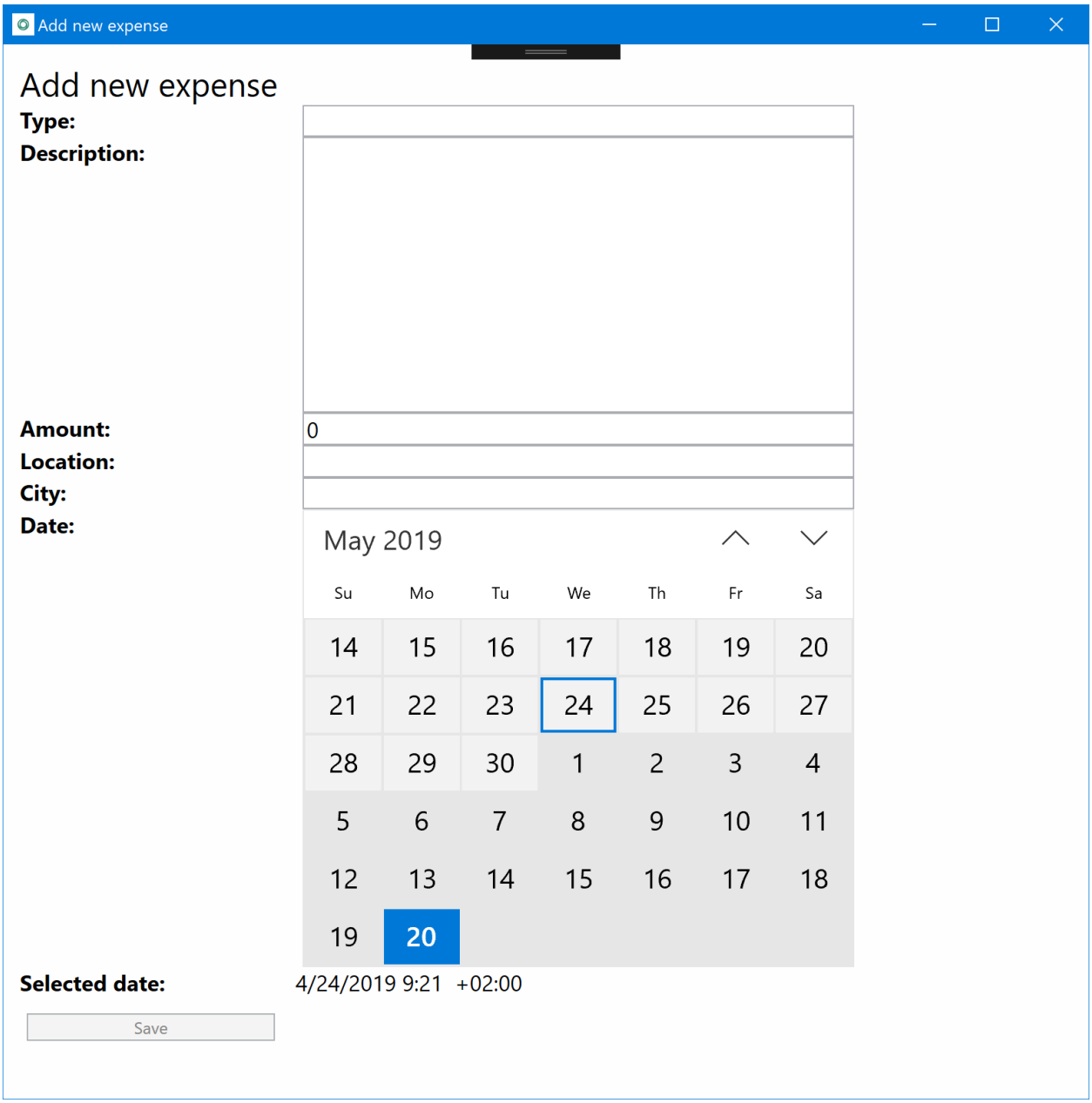


Figure 62: A UWP CalendarView control included inside a WPF window

As you can see in Figure 62, thanks to the events that we have subscribed to inside the **`ChildChanged`** event, we are able to interact with the control from our WPF application. The selected date is displayed in the window inside a WPF **`TextBlock`**.

XAML Islands and MSIX

You might be wondering what the relationship is between MSIX and XAML Islands. This technology is available starting from Windows 10 version 1903, so it requires your application to target this version or later as minimum. If you don't satisfy this requirement, your application will

crash with the following exception as soon as you try to load a window that contains one or more UWP controls.

Figure 63: The exception raised when you use XAML Islands without a manifest

The best and quickest solution is to package your application with MSIX. In the MSIX manifest, you can declare the minimum Windows 10 version the app supports, so you can make sure that your Windows application using XAML Islands is targeting 1903 as minimum version.

Alternatively, you can also add an application manifest, which is an XML that describes and identifies the shared and private side-by-side assemblies that an application should bind to at runtime. Starting from Windows 10 version 1903, the manifest includes a property called **`maxVersionTested`**, which you can use to leverage some Windows 10 features in an unpackaged application. You can learn more about this approach in this article.

The future of XAML Islands: WinUI 3.0

As already mentioned, every Windows 10 feature update brings new features for developers. This includes new UI controls, like the teaching tip introduced in Windows 10 version 1903. The limitation of this approach is that, in order to use these controls, your customer must be on the same or later version, which is not always possible in the enterprise space. For this reason, Microsoft has introduced WinUI, an open source library that includes most of the inbox UWP controls and that offers backward compatibility back to Windows 10 version 1607. Since it's a NuGet package, you just need to install it in your Windows project to take advantage of the latest UI controls.

However, WinUI doesn't help to solve this problem when it comes to XAML Islands, since it leverages some features of the XAML framework that are deeply integrated with Windows. Currently, your customers must use Windows 10 1903 or later to launch a classic Windows application that uses XAML Islands to host UWP controls. However, at Build 2019, Microsoft announced that the next release of WinUI, 3.0, will solve this problem by detaching the whole XAML Framework from the operating system and moving it directly inside the library. As a consequence, you will be able to support XAML Islands in your applications starting from Windows 10, version 1607, just by installing the WinUI 3.0 package from NuGet in your WPF or Windows Forms applications.

The first preview of WinUI 3.0 is scheduled for the end of 2019. You can find the complete roadmap on GitHub.

Extend your Windows application with Universal Windows Platform components

The Universal Windows Platform offers many powerful APIs to add new features to your existing Windows application. However, there's more! UWP also comes with many components that allow you to perform tasks that, in the past, were reserved for full Universal Windows Platform applications.

Features like background tasks, App Services, and contracts can help your application to integrate even better with Windows 10 and add more value for your users.

In this chapter, we're going to explore one of these components: background tasks.

Background tasks

The Universal Windows Platform has introduced a new way to perform operations in the background through the introduction of background tasks. This new approach was required due to the different lifecycle of Universal Windows Platform applications compared to the classic Win32 ones. Once an application is minimized in the taskbar, it gets suspended, which means that it can't leverage CPU, network, or storage. As a consequence, it isn't able to perform any operations in background. Background tasks have been created to overcome this limitation. They are small components, separate from the main application, that contain code that can be executed at any stage of the lifecycle. Whether the application is opened, closed, or suspended, Windows can run a background task if the conditions for its execution are satisfied.

Thanks to this approach, background tasks are much more friendly in terms of CPU, memory, and consumption compared to the traditional Win32 approaches, like using Windows Services or keeping an entire application in the background.

A background task is controlled by two factors:

- A **trigger**, the event that triggers the execution of the task.
- A **condition**, a way to optimize resources and to make sure the task is executed only if all the conditions to perform the operation are satisfied.

However, a traditional Windows application, even when it's packaged as MSIX, doesn't follow the same lifecycle as a full UWP application. It can run in the background indefinitely, and it can access all the computer's resources. Which are the features that make background tasks interesting for classic Windows applications?

The first one is the flexibility of the various triggers available in the system. When you develop a Windows Service or a background process, you need to manually intercept the event that you want to handle. Some of them can be hard to pin down, and you may consume lots of resources to listen to them continuously. Triggers in background tasks, on the other hand, will allow you to focus only on the operation you want to execute. Windows will trigger the task for you when the

event happens, even the most complex ones, like a time zone change or the connection with a Bluetooth device.

The second one is the better usage of system resources. With Windows 10, we have seen the advent of many new form factors that go beyond the classic desktop computer, such as 2-in-1s, detachable devices, and tablets. These devices may be optimized for portability and have less power compared to a full desktop PC, or they may be used for long period of times without a power source. Having applications that don't waste CPU or battery life is critical for all these scenarios. Background tasks include a lot of optimizations to make sure their impact on the system resources is minimum. Additionally, Windows can track them and limit their execution in case the system's conditions aren't optimal, like when there is low battery power or too much memory in use.

A background task is defined using a Windows Runtime component, a class library built on top of the Windows Runtime. The main difference from a traditional library is that, through the generation of metadata, it can be consumed using any language supported by the Universal Windows Platform, regardless of the language used to build the component.

The following image shows you the general architecture of a background task.

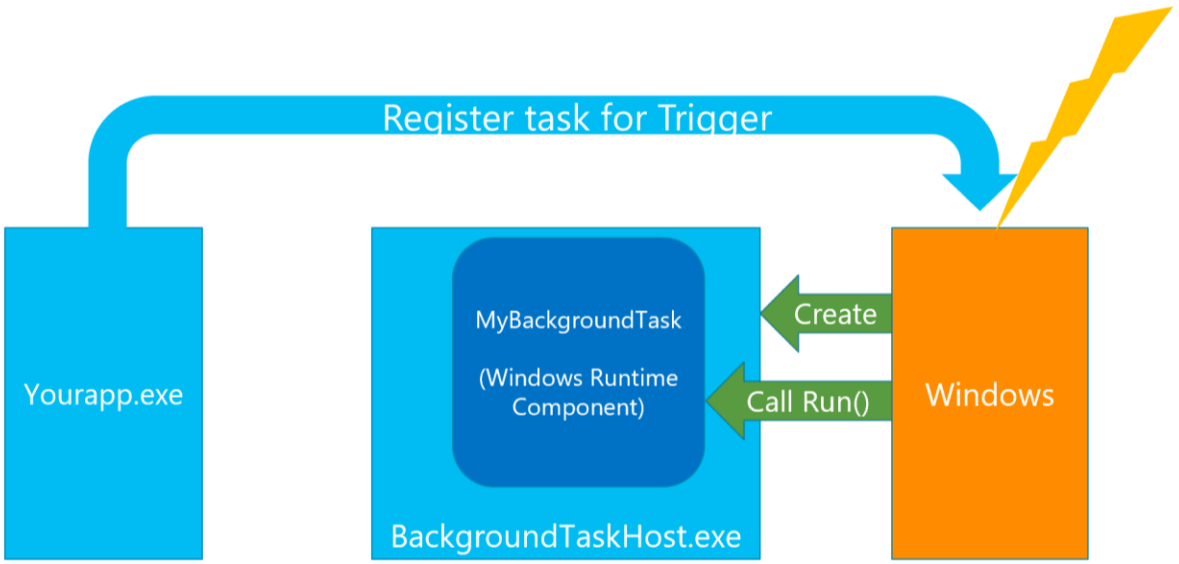


Figure 64: The architecture of a background task

The task is registered by the application. During the registration process, the application defines which trigger is connected to the execution of this task. Once it's registered, the application hands off the execution to Windows. When the registered trigger occurs, Windows will start a new process called **BackgroundTaskHost.exe**, which will create a new instance of the Windows Runtime Component. Then it will look for a method called **`Run()`**, implemented by the component, and it will execute it. As we're going to see in the rest of the section, as a developer you will be required to implement inside this method the logic you want to execute when the task is invoked.

Windows 10 also supports another background task model, called **in-process**. In this scenario, the task isn't implemented with a separate background task, but directly inside the main

application. However, this implementation relies on the lifecycle of a full Universal Windows Platform application and can't be leveraged in a packaged Win32 application scenario.

Let's see how to implement a background task for the following scenario: we want to display a notification to the user every time the status of the internet connection changes.

Create a background task

The first step is to add a Windows Runtime component to your WPF or Windows Forms solution. You can find this template when you create a new project.

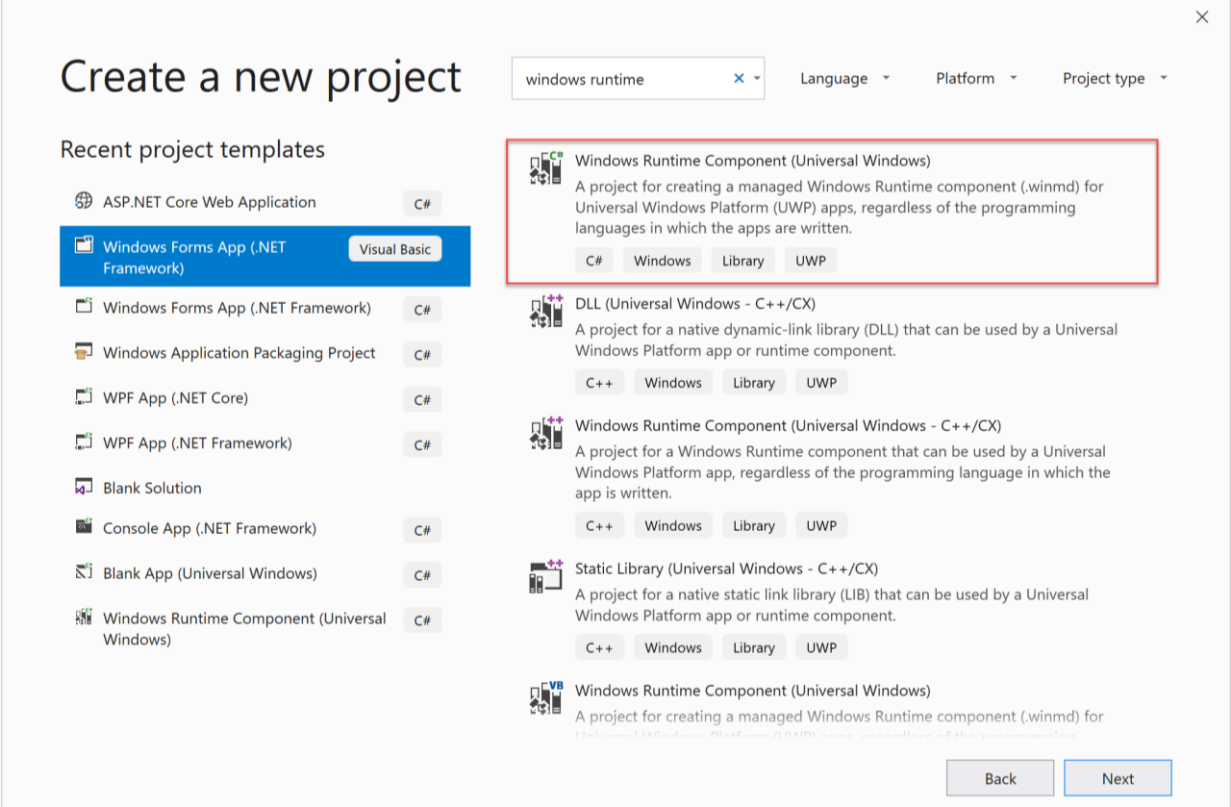


Figure 65: The template to create a Windows Runtime Component in Visual Studio

By default, the new project will contain just one empty class. Feel free to rename it with a more meaningful name. The next step is to inherit your class from the **`IBackgroundTask`** interface, which will require you to implement a method called **`Run()`**.

Code Listing 37

```
public sealed class OfflineTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
```

```
    {
        var deferral = taskInstance.GetDeferral();

        //background task implementation.

        deferral.Complete();
    }
}
```

Inside this method, we're going to add the code that we want to execute when the task is triggered. Inside a task, it's important to leverage the **deferral**, which is a special object that keeps the task alive during the execution of asynchronous operations. Typically, when the task is executed, you get a reference to the deferral by calling the **GetDeferral()** method, and then you release it once the task is ended by calling the **Complete()** one. Otherwise, as soon as you start an asynchronous operation, which is executed on a different thread, Windows will think that the task has completed its work and will terminate it.

The benefit of using a background task is that you don't have to focus on the event you want to handle, but rather on the code you want to execute. Let's see the implementation of our task.

Code Listing 38

```
public sealed class OfflineTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        var deferral = taskInstance.GetDeferral();

        var profile = NetworkInformation.GetInternetConnectionProfile();

        string title;
        string description;

        if (profile != null)
```

```
        {
            title = "You're online!";
            description = "Make sure to sync all the expenses for your
trip!";
        }
        else
        {
            title = "You're offline!";
            description = "Your expenses won't be synced until you're
back online.";
        }



        string xml = $@"<toast>
        <visual>
            <binding template='ToastGeneric'>
                <text>{title}</text>
                <text>{description}</text>
            </binding>
        </visual>
    </toast>";

        XmlDocument doc = new XmlDocument();
        doc.LoadXml(xml);

        ToastNotification toast = new ToastNotification(doc);
        ToastNotificationManager.CreateToastNotifier().Show(toast);

        deferral.Complete();
```

```
    }
}
```

As you can see, the task doesn't contain any reference to the event that is going to trigger it. It just includes the code that we want to execute when the trigger happens. In our scenario, we use the **`NetworkInformation`** class and the **`GetInternetConnectionProfile()`** method to discover the status of the internet connection (available or not available). Then we create a new **`ToastNotification`** object using an XML schema to display the network status to the user.

Registering the background task in code

Once we have defined the background task, we need to register it inside the application so that we can specify which trigger and which conditions we want to handle. Where to add this code is up to the requirements of your application. In most of the cases, a background task is automatically registered when the application is launched, so it's a good practice to place it when the application starts or when the first window is loaded. However, there are scenarios where you might want to keep the background task off by default and register it only as a user's choice, for example, with a button in the Settings page.

Regardless of your choice, here is the code snippet to register a background task.

Code Listing 39

```
public void RegisterBackgroundTask()
{
    string triggerName = "NetworkTrigger";
    bool taskRegistered = false;
    // Check if the task is already registered.
    foreach (var cur in BackgroundTaskRegistration.AllTasks)
    {
        if (cur.Value.Name == triggerName)
        {
            taskRegistered = true;
            break;
        }
    }
```

```
    if (!taskRegistered)
    {
        BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
        builder.Name = triggerName;
        builder.SetTrigger(new
SystemTrigger(SystemTriggerType.NetworkStateChange, false));
        builder.AddCondition(new
SystemCondition(SystemConditionType.UserPresent));
        builder.TaskEntryPoint = "ContosoExpenses.Task.OfflineTask";
        builder.Register();
    }
}
```

These APIs are part of the Universal Windows Platform, and they require the declaration of the following namespace.

Code Listing 40

```
using Windows.ApplicationModel.Background;
```

The first step is to assign a unique name to the trigger, in this case **`NetworkTrigger`**. Then, by iterating through the **`BackgroundTaskRegistration.AllTasks`** collection, we check if the task has already been registered during a previous execution of the application. If that's the case, we can skip the rest of the code, since we don't have to register it again.

If it isn't registered, instead, we kick-off the registration process by using the **`BackgroundTaskBuilder`** class. We need to define the following information:

- The unique name, by using the **`Name`** property.
- The **`TaskEntryPoint`**, which is the full signature of the class that implements the **`Run()`** method. It's the class we have previously implemented in the Windows Runtime component.
- The trigger that will execute the task. We add it by calling the **`SetTrigger()`** method and passing one of the available triggers. In this case, we're using one of the system triggers, specifically the **`NetworkStateChange`** one. By passing **`false`** as second parameter, we're telling Windows that we want to execute the task every time the selected event is triggered. By passing **`true`**, we would register a one-shot background task.
- One or more conditions that need to be satisfied in order to run the background task. This isn't a requirement, but it can be helpful to save resources. In this sample, we're using the system condition **`UserPresent`** to make sure that the task runs only if the user

is actively using the computer. If the user is away from the screen, we don't need to alert them with a notification.

As a last step, we call the **`Register()`** method to complete the registration.

Registering the background task in the manifest

Out-of-process background tasks must be declared in the manifest of the application; otherwise, they can't be registered.

You must open the manifest by double-clicking the **Package.appxmanifest** file included in the Windows Application Packaging Project of your solution and move to the **Declarations** tab. From the drop-down list, choose **Background Tasks** and click **Add.**

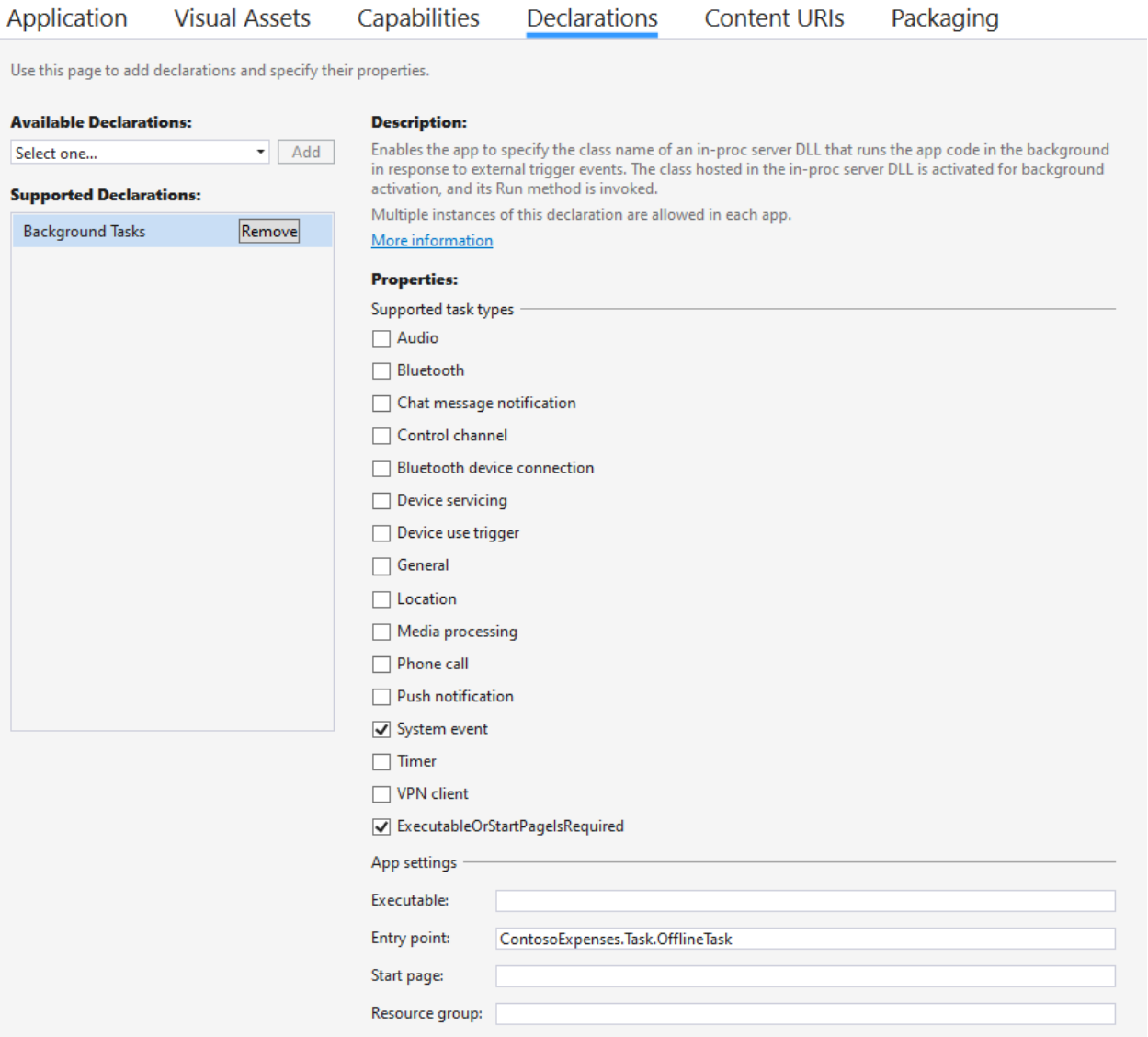


Figure 66: The background task declared in the manifest of the packaged application

First you need to choose from the list which type of background task you're registering. In our case, we're using the **`NetworkStateChange`** trigger, which belongs to the **System event** category. In the **Entry point** field, you need to add the full namespace of the class in the Windows Runtime component that implements the **`Run()`** method. It's the same signature you specified in the **`TaskEntryPoint`** property when you registered the task in code.

Packaging everything together

If you try to package the whole solution, you may face some challenges. The first one is that the background task is implemented using a Windows Runtime component, which belongs to a different ecosystem than the different .NET Framework. As such, Visual Studio won't allow you to directly reference the background task from your WPF or Windows Forms application. A potential solution would be to reference the Windows Runtime component directly from the Windows Application Packaging Project. However, this isn't a supported scenario. The project is meant to package applications and, as such, it won't allow referencing a project that doesn't output an executable, like a library.

So, are we stuck? Luckily, there's a workaround, which is using an empty UWP project as container for our background task. Universal Windows Platform apps can reference a background task without issues since they belong to the same ecosystem. However, the application will be empty. It will be used to make sure that the MSIX package contains all the required assemblies and files, but it will never be launched and made visible to the user.

As a first step, right-click your Visual Studio solution and add a new project of type **Blank App (Universal Windows)**.

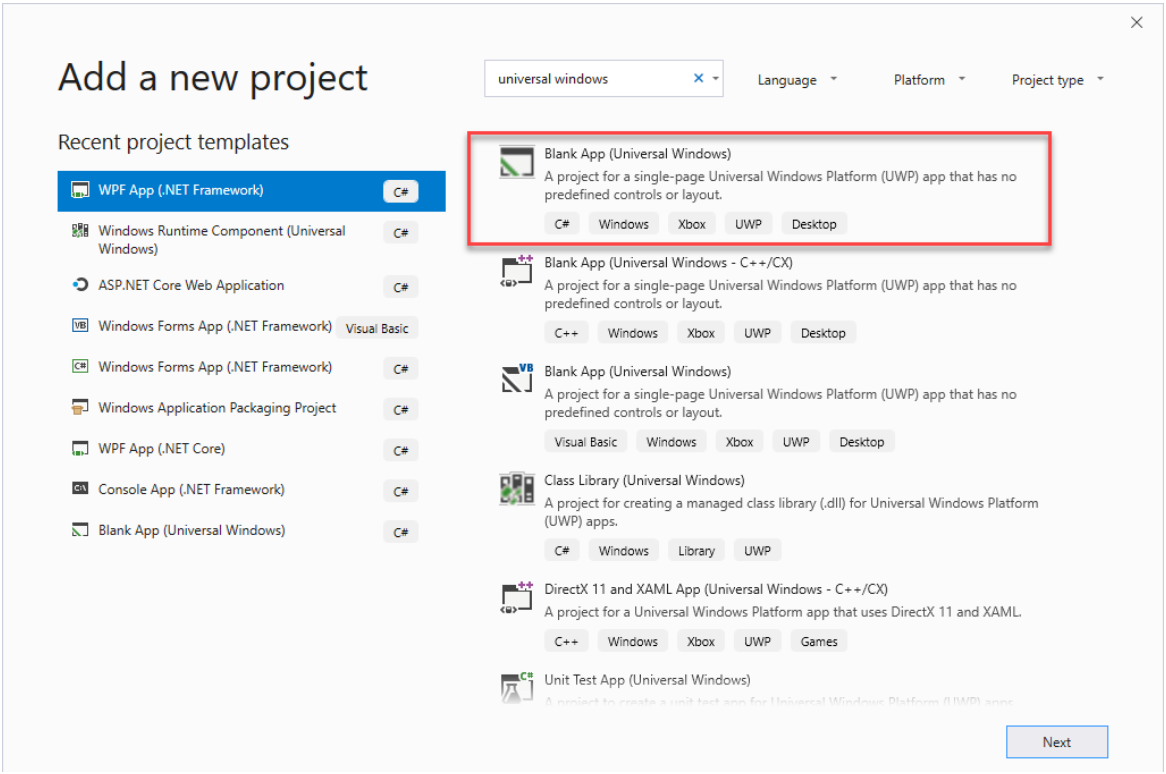


Figure 67: A project to create a blank Universal Windows Platform application

Once the project has been created, right-click it and choose **Add reference**. From the **Project -> Solutions** section, choose the Windows Runtime component that includes your background task.

In the end, right-click the Windows Application Packaging Project, choose **Add reference** and add the UWP project you have just created. Then expand the **Applications** section and make sure the project that contains your WPF or Windows Forms app is in bold, which means that it will be the entry point of the package.

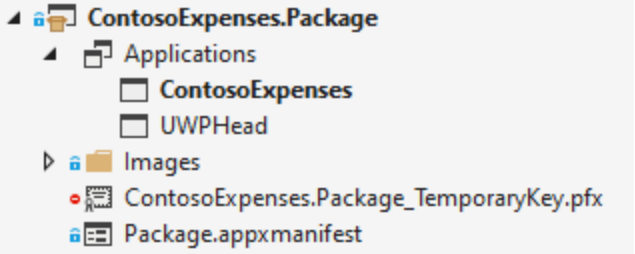


Figure 68: A Windows Application Packaging Project that contains two applications: a WPF one and a UWP one

Testing the application

That's it. Now you must set the Windows Application Packaging Project as startup and launch it. You just have to trigger the event that will cause the execution of the background task. In some scenarios, like ours, it's easy. You just need to disable and enable your internet connection, and you should see a notification popping out.

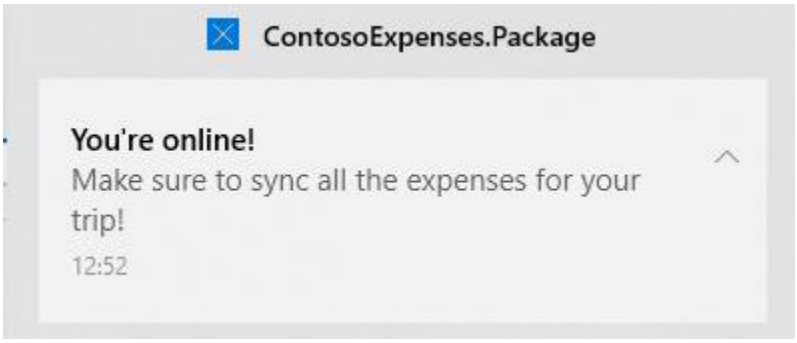


Figure 69: A toast notification sent by a background task

In some other cases, it might be more complicated, like for a time-based trigger, so Visual Studio provides a convenient way to force the execution of the task. Once the debugger is attached, look in the toolbar for a drop-down menu called **Lifecycle Events**. By expanding it, you will see a set of default events, plus a list with all the background tasks registered by your application. Just click the one you want to test to force the execution.

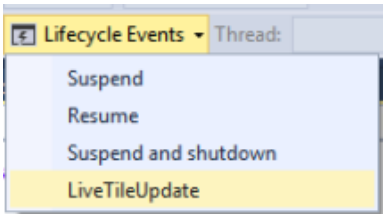


Figure 70: The option in Visual Studio to force the execution of a background task

As a side effect of the new configuration of your solution, you might see two applications being deployed in your Start menu: your Win32 packaged application, and the blank UWP application. This happens only when you debug in Visual Studio since, by default, it deploys all the projects included in the package that output an executable. When you generate an MSIX package for the Microsoft Store or for sideloading, you won't face this problem, since your package has only a single entry point. However, if you want to remove this behavior from Visual Studio, as well, you just need to navigate to **Build > Configuration Manager** and uncheck the **Deploy** option for the blank UWP project.

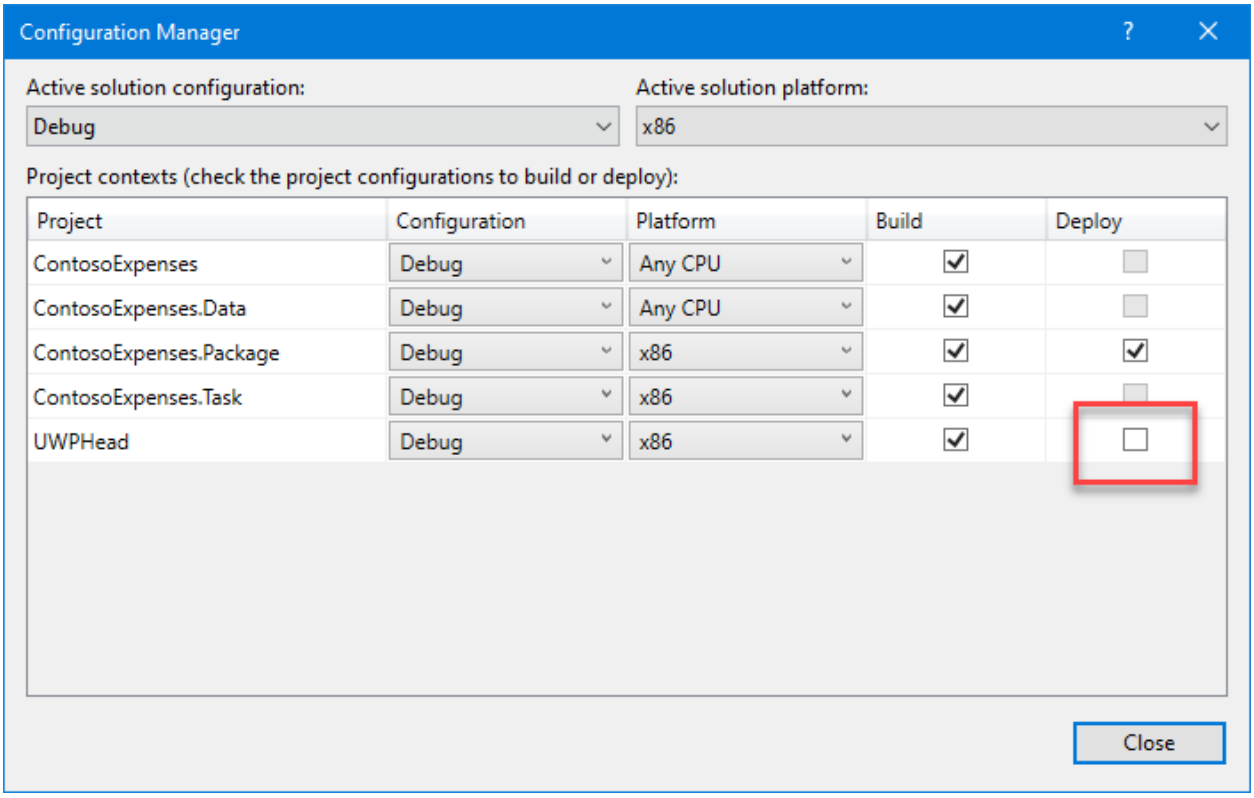


Figure 71: The option to uncheck to avoid the deployment of the blank UWP project

Use a Win32 process in a Universal Windows Platform application

So far, we have seen scenarios where you can integrate features from the Universal Windows Platform inside a classic .NET application. What about the other way around? What if I have a Universal Windows Platform application and I want to leverage features from the Win32 ecosystem?

There are two compelling scenarios where this approach can be useful:

- You want to leverage a library or a process that is targeting the .NET Framework, but it doesn't provide support to the Universal Windows Platform, or you aren't ready to migrate it.
- You want to leverage a feature that isn't supported by the Universal Windows Platform.

Thanks to the Windows Application Packaging Project and a set of extensions included in the Universal Windows Platform, you can achieve this goal. But first, let's introduce app services.

App services

Previously in this chapter, we learned about background tasks. App services are built on top of the same concept, but they can be leveraged from multiple applications. Once an application deploys an app service, all the other applications installed on the computer can interact with it. Applications can establish a communication channel with the app service and, using a key-value pairs collection, exchange data.

App services are a great way to offload operations that must be shared across a suite of applications. For example, an app service could implement the logic to extract information from a bar code. Then, other applications belonging to the same suite can just send the bar code image to the app service and get back the decoded text. You can think of app services as a way to build a set of REST services, but which run offline and are managed by Windows.

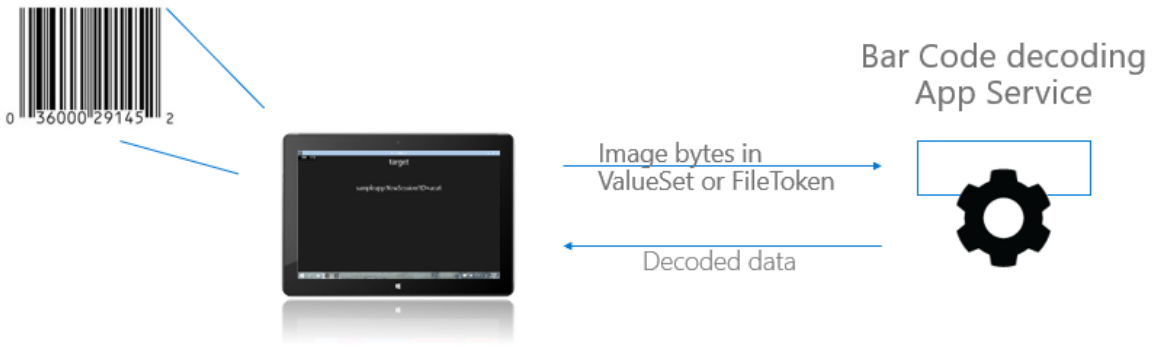


Figure 72: A sample usage of an app service

From a technical point of view, app services and background tasks share the same implementation. They both implement the **`IBackgroundTask`** interface and the code to execute when the task is invoked is defined inside the **`Run()`** method. The main difference is the trigger that causes the execution. For background tasks, we have seen that triggers are events that happen inside the system: a network connection change, a Bluetooth device connected, etc. For app services, the trigger is the incoming request from another application.

A client application can connect to an app service by using the **`AppServiceConnection`** class, included in the **`Windows.ApplicationModel.AppService`** namespace of the Universal Windows Platform. The following sample shows you a connection.

Code Listing 41

```
public async Task Connect()
{
    //we create a connection with the app service defined by the UWP app.
    var connection = new AppServiceConnection();
    connection.AppServiceName = "DatabaseService";
    connection.PackageFamilyName = "Contoso.MyEmployees_8rkfj2ay7vd1w";
    connection.RequestReceived += Connection_RequestReceived;
    connection.ServiceClosed += Connection_ServiceClosed;

    //we open the connection.
    AppServiceConnectionStatus status = await connection.OpenAsync();

    if (status == AppServiceConnectionStatus.Success)
    {
        ValueSet initialStatus = new ValueSet();
        initialStatus.Add("Status", "Ready");
        await connection.SendMessageAsync(initialStatus);
    }
}
```

The two relevant pieces of information to establish a connection are:

- The **`AppServiceName`**, defined by the application that exposes the app service.
- The **`PackageFamilyName`** of the application that exposes the app service.

Once you have established a connection using the **OpenAsync()** method, you can start to exchange data between the client application and the app service using a **ValueSet**, which is a key-value pairs collection. Messages can be exchanged via the **SendMessageAsync()** method.

After the connection is established, the app service will receive a reference to the channel, and it will be able to use it as well as receive and send messages to the client application. Like for background tasks, you can implement an app service using the out-of-process model (a dedicated Windows Runtime component) or the in-process model (directly in the UWP app). The following sample shows the in-process implementation, which is done in the **App** class of the Universal Windows Platform application.

Code Listing 42

```
protected override void
OnBackgroundActivated(BackgroundActivatedEventArgs args)

{

    base.OnBackgroundActivated(args);

    if (args.TaskInstance.TriggerDetails is AppServiceTriggerDetails)

    {

        AppServiceTriggerDetails details =
args.TaskInstance.TriggerDetails as AppServiceTriggerDetails;

        Connection = details.AppServiceConnection;

        Connection.RequestReceived += Connection_RequestReceived;

    }

}


private async void Connection_RequestReceived(AppServiceConnection
sender, AppServiceRequestReceivedEventArgs args)

{

    if (args.Request.Message.ContainsKey("Status"))

    {

        string currentStatus = args.Request.Message["Status"].ToString();

        ValueSet set = new ValueSet();

        set.Add("StatusUpdated", "Status has been updated");

        await args.Request.SendResponseAsync(set);
```

```
    }
}
```

In the in-process implementation, the **`Run()`** method of the Windows Runtime component is replaced by the **`OnBackgroundActivated()`** method. The Universal Windows Platform lifecycle, in fact, allows the application to be activated by an event without having to actually launch the whole application. In this scenario, the method will be triggered when another application calls the app service. Windows will execute it in background, and then it will terminate the process.

The arguments of the event handler come with a property called **`TaskInstance.TriggerDetails`**, which contains all the information about the channel. The most important one is **`AppServiceConnection`**, which is the actual connection established with the calling application. Once we have the connection, we can use it like we did in the calling application. We can start listening for incoming requests by subscribing to the **`RequestReceived`** event or we can send a message using the **`SendMessageAsync()`** method.

In the previous snippet, you can see what happens when the app service has received a message. The **`RequestReceived`** event handler is triggered and, inside it, thanks to the **`Request`** property, you can access the **`Message`** collection, which is the **`ValueSet`** object sent by the calling application. Then you can craft a response and send it back to the calling application by calling the **`SendResponseAsync().`**

Exactly like a background task, an app service must be declared in the manifest of the application through the **Declarations** section.

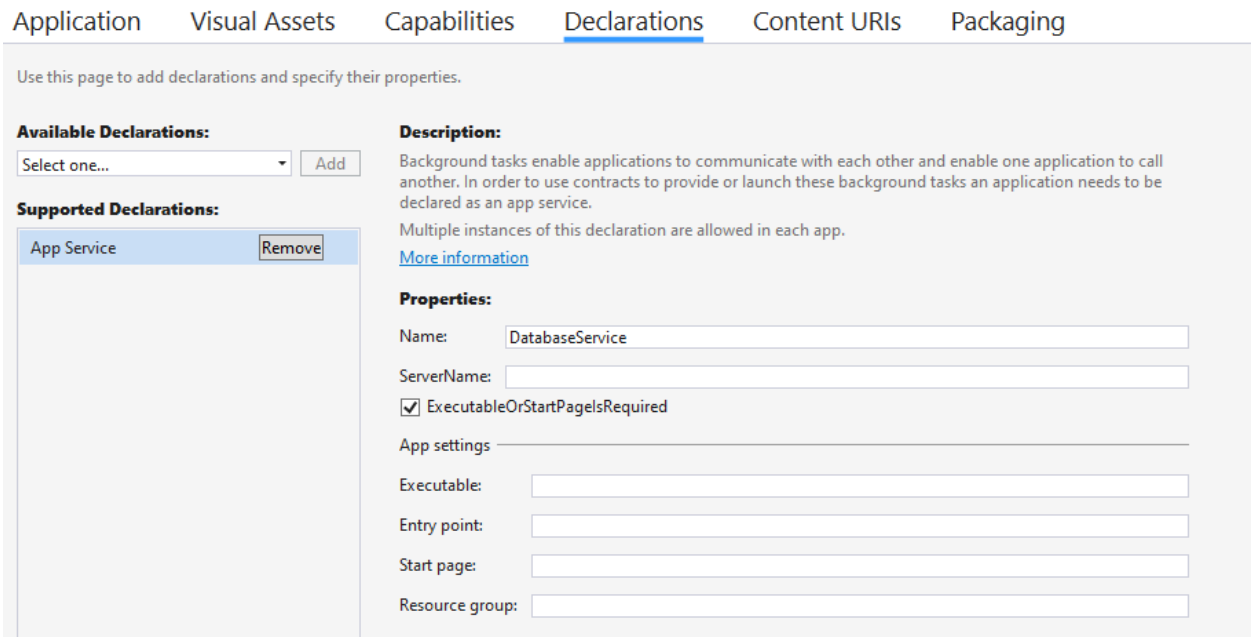


Figure 73: An manifest declaration to expose an app service

The relevant information to set is the **Name**, which is the one that the other application will need to use (in combination with the Package Family Name of the app) to connect to it. If your app service is declared out-of-process, you will also have to set the **Entry point** with the full signature of the class that implements it. If, instead, like in the previous sample, the app service is handled in-process, there's no need to specify an entry point.

Why are app services important in the context of using Win32 processes from a Universal Windows Platform application? Because they are the way the two processes can communicate. The Universal Windows Platform application will expose an app service, and the Win32 process included in the same package will connect to it. By using **`ValueSet`** objects, the two processes will be able to exchange data. This way, the Win32 process can perform one or more operations and then return the result to the Universal Windows Platform application, making it possible to perform tasks that are supported only by the .NET Framework.

The only limitation is that this scenario is supported only by in-process app services. We can't build a communication channel using a separated Windows Runtime component, like we did in the previous section when we implemented a background task.

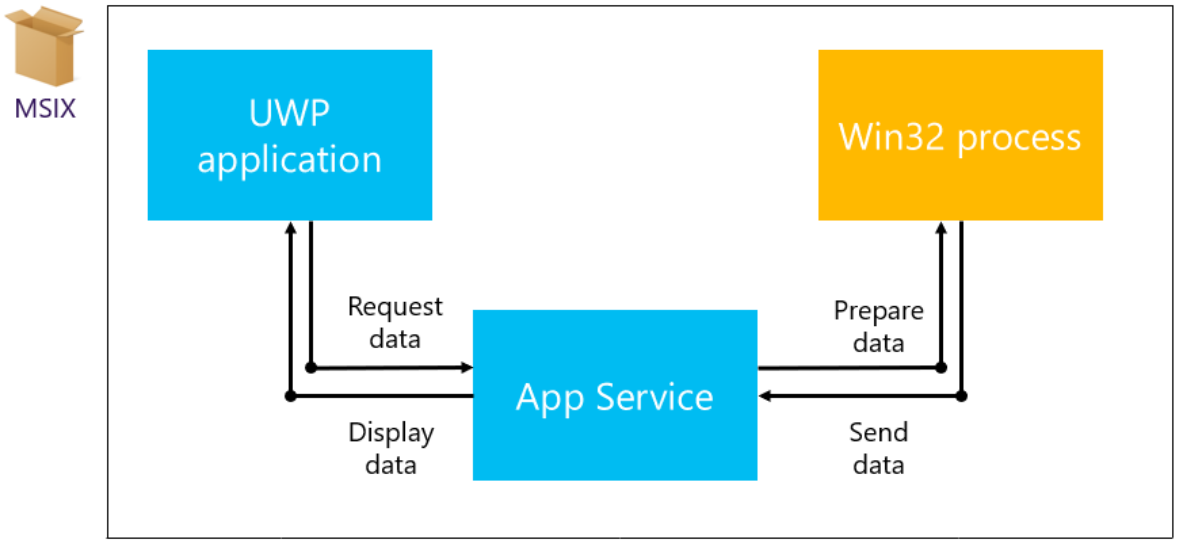


Figure 74: An app service used as a communication channel between a UWP application and a Win32 process included in the same package

A real example

To demonstrate the usage of this feature, let's build a Universal Windows Platform application that displays some data to the user. The challenge is that this data is coming from a Microsoft Access database, which isn't supported by the Universal Windows Platform. We don't have any APIs that we can use to query an Access database. The .NET Framework, on the other hand, supports this functionality, so we can delegate this operation to a Win32 process. In Figure 75, you can see the look of the final application.

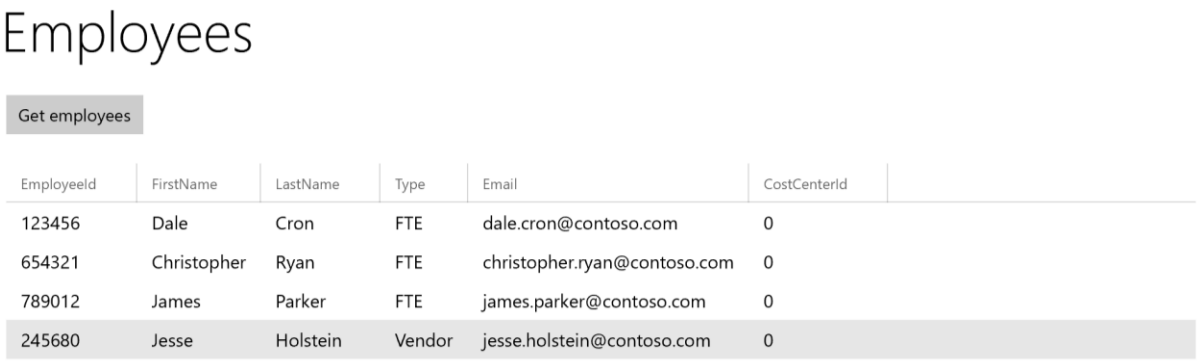


Figure 75: The Employees application built with UWP, but powered by a Win32 process

The main application is built with the Universal Windows Platform, and it leverages a modern UI. However, when the user clicks the **Get employees** button, the operation will be performed by the Win32 process.

This is how our solution looks.

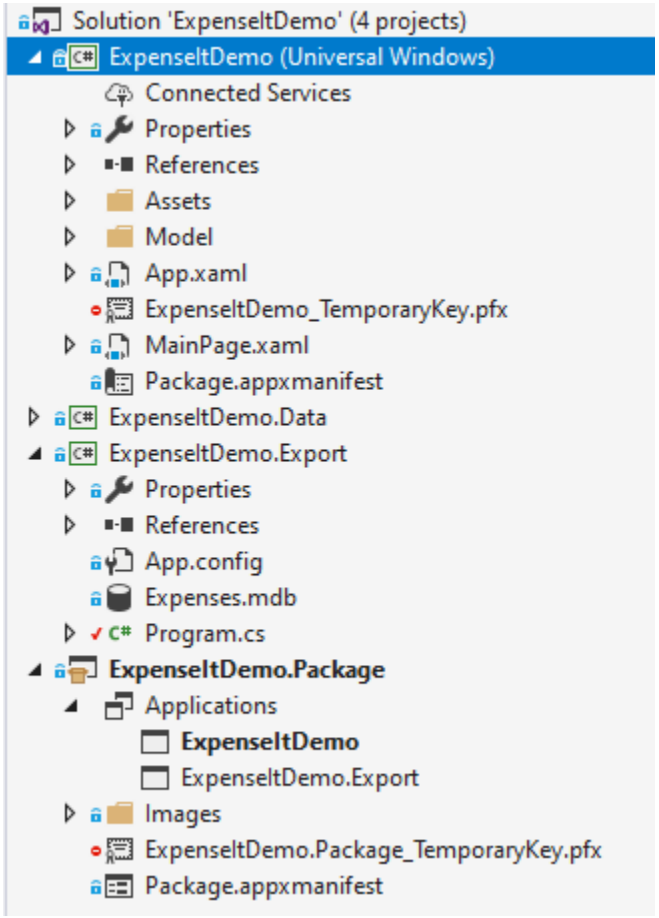


Figure 76: The solution that includes a UWP application and a Win32 process

We have a UWP application (**ExpenseItDemo**), a Win32 process **(ExpenseItDemo.Export**), and a Windows Application Packaging Project **(ExpenseItDemo.Package**). The last one is required because it helps us to combine both applications in a single MSIX package. As you can see, they are both added as applications inside the project, but the UWP one is set as the entry point.

The first step to building our solution is to declare the app service in the manifest. However, there's an important thing to highlight. Once you create a package with the Windows Application Packaging Project, its manifest becomes the main one that defines the identity. As such, even if at first glance you would be tempted to declare the app service in the application's manifest of the Universal Windows Platform application, you must do it in the Packaging Project's one; otherwise, it won't be recognized. You must replicate the configuration you see in Figure 76 in the **Package.appxmanifest** file included in the **ExpenseItDemo.Package** project.

The Win32 process

In this scenario, the Win32 process is completely transparent to the user. It doesn't provide any UI, but it just takes care of retrieving the data from the Access database and returning it back to the UWP application. The process is a simple console application, built on top of the .NET Framework.

> ***Tip: If you right-click the Console project and change the Output Type from Console Application to Windows Application, you will get an executable that runs in background without opening a console window. It's the best choice for this scenario, because the users won't see any application running other than the main UWP one unless they open Task Manager.***

Let's see the full implementation and explain it step by step.

Code Listing 43

```
class Program
{
    static AppServiceConnection connection = null;
    static AutoResetEvent appServiceExit;

    static void Main(string[] args)
    {
        //we use an AutoResetEvent to keep the process alive until the
communication channel established by the app service is open.
        appServiceExit = new AutoResetEvent(false);
```

```
        Thread appServiceThread = new Thread(new
ThreadStart(ThreadProc));
        appServiceThread.Start();
        appServiceExit.WaitOne();
    }

    static async void ThreadProc()
    {
        //we create a connection with the app service defined by the UWP
app.
        connection = new AppServiceConnection();
        connection.AppServiceName = "DatabaseService";
        connection.PackageFamilyName =
Windows.ApplicationModel.Package.Current.Id.FamilyName;
        connection.RequestReceived += Connection_RequestReceived;
        connection.ServiceClosed += Connection_ServiceClosed;

        //we open the connection.
        AppServiceConnectionStatus status = await connection.OpenAsync();

        if (status != AppServiceConnectionStatus.Success)
        {
            //if the connection fails, we terminate the Win32 process.
            appServiceExit.Set();
        }
        else
        {
            //if the connection is successful, we communicate to the UWP
app that the channel has been established.
            ValueSet initialStatus = new ValueSet();
```

```
            initialStatus.Add("Status", "Ready");
            await connection.SendMessageAsync(initialStatus);
        }
    }

    private static void Connection_ServiceClosed(AppServiceConnection
sender, AppServiceClosedEventArgs args)
    {
        //when the connection with the app service is closed, we
terminate the Win32 process.
        appServiceExit.Set();
    }

    private static async void
Connection_RequestReceived(AppServiceConnection sender,
AppServiceRequestReceivedEventArgs args)
    {
        if (args.Request.Message["GetEmployees"].ToString() == "true")
        {
            DatabaseService service = new DatabaseService();
            List<Employee> employees = service.GetEmployees();

            string serializedValue =
JsonConvert.SerializeObject(employees);

            ValueSet set = new ValueSet();
            set.Add("Employees", serializedValue);

            await args.Request.SendResponseAsync(set);
        }
```

```
    }
}
```

The **Main()** method takes care of initializing a thread using an **AutoResetEvent** object. This way, the process will be kept alive until the communication channel established by the app service is open.

The thread executes a method called **ThreadProc()**, which initializes the connection to the app service and starts listening for incoming messages from the main Universal Windows Platform application. The connection is established using the code we learned to use in the previous section. We use an **AppServiceConnection** object and set the **AppServiceName** property with the app service's name we previously declared in the manifest. Then we set the **PackageFamilyName** property with the PFN of the UWP application. Since the process is running inside the same package of the UWP application, they share the same identity. As such, we can retrieve it using the **Windows.ApplicationModel.Package.Current.Id.FamilyName** property.

In the end, we call the **OpenAsync()** method to establish the connection. If the operation fails, we call the **Set()** method on the **AutoResetEvent** object, so that the process can be terminated. Otherwise, using a **ValueSet** object and the **SendMessageAsync()** method, we send a message to the UWP app to tell it that we're ready to send and receive messages through the channel.

We have also subscribed to two events offered by the **AppServiceConnection** object. The first one is **ServiceClosed**, and it's triggered when the channel has been closed. As we did before in the case of an unsuccessful connection, we use this event to terminate the process by calling the **Set()** method on the **AutoResetEvent** object.

The **RequestReceived** event is where the real work happens. This event is triggered when the UWP application sends a request to the Win32 process. In our scenario, this will happen when the user presses the button to see the list of employees. When this event happens, the UWP application will send a message with the key **GetEmployees**. As such, inside this event handler, we retrieve from the database the list of employees. The details of the **DatabaseService** class implementation are out of the scope in this context, since we're just building a sample to showcase how to connect a UWP app to a Win32 process. In your scenario, you could use any .NET Framework API you may need to achieve your goal. The key information to know is that the **GetEmployees()** method is using classes like **OleDbConnection** and **OleDbCommand** to query the Access database and return a collection of **Employee** objects. These classes aren't supported by the Universal Windows Platform or .NET Standard, so they can't be leveraged directly from the main application.

In the end, we send the collection of employees back to the UWP application by using a **ValueSet**. However, being a key-value pairs collection, it supports only data that can be serialized. As such, we can't send it back to the collection as it is, we need to serialize it. In this case, as a JSON using the popular library Json.NET. We do it by calling the **SerializeObject()** method exposed by the **JsonConvert** class.

The Universal Windows Platform application

Inside the Universal Windows Platform application, we need to follow the same guidelines we have seen about implementing in-process app services. When the Win32 process opens a connection to our app service, the **OnBackgroundActivated** event exposed by the **App** class is triggered. Inside this code, we need to implement the execution of the app service. However, in the context of our scenario, the only thing that we need to do is store a reference to the connection, so that we can reuse it later in the other pages of the application, where the real communication will happen.

First, let's declare a couple of variables in the **App** class.

Code Listing 44

```
public static AppServiceConnection Connection = null;

private BackgroundTaskDeferral appServiceDeferral = null;
```

The first property will be used to store the connection. It's declared as **static** so that we can easily reuse it from other pages of the application. The second property is **BackgroundTaskDeferral**, which is needed to keep the connection alive. Using it is very important; otherwise the app service will drop the connection immediately after having initialized the channel.

Now we have everything we need to implement the **OnBackgroundActivated** event handler.

Code Listing 45

```
protected override void
OnBackgroundActivated(BackgroundActivatedEventArgs args)

{

    //this method is invoked when the Win32 process requests to open the
communication channel with the app service.

    base.OnBackgroundActivated(args);

    if (args.TaskInstance.TriggerDetails is AppServiceTriggerDetails)

    {

        appServiceDeferral = args.TaskInstance.GetDeferral();

        AppServiceTriggerDetails details =
args.TaskInstance.TriggerDetails as AppServiceTriggerDetails;

        Connection = details.AppServiceConnection;

    }
```

```
}
```

When the Win32 process makes a request to open a connection with the app service, we accept it and, by using the **AppServiceTriggerDetails** property of the activation arguments, we store it inside the **Connection** property we previously defined.

Now we can leverage this property in the main page of our application. When the user clicks the button to get the data, the Universal Windows Platform application will send a message through this connection to the Win32 process. Here is the implementation of the event handler raised by the button in the main page.

Code Listing 46

```
private async void OnGetEmployees(object sender, RoutedEventArgs e)
{
    //if the connection with the app service has been established, we
send the info about the flight to the Win32 process.
    if (App.Connection != null)
    {
        ValueSet set = new ValueSet();
        set.Add("GetEmployees", "true");

        AppServiceResponse response = await
App.Connection.SendMessageAsync(set);
        //if the Win32 process has received the data and has successfully
generated the boarding pass, we show a notification to the user.
        if (response.Status == AppServiceResponseStatus.Success)
        {
            string message = response.Message["Employees"].ToString();
            List<Employee> employees =
JsonConvert.DeserializeObject<List<Employee>>(message);
            EmployeesView.ItemsSource = employees;
        }
    }
}
```

If you understand how an app service works, the code should be easy to understand. We create a **`ValueSet`** with a message with the **`GetEmployees`** key. When the Win32 process receives such a message, it will query the Access database and send back the data using the JSON format. If the response is successful (we use the **`Status`** property of the **`AppServiceResponse`** object to determine it), we extract the message from the response and look for the data inside the item identified by the **`Employees`** key. Then we again use Json.NET to deserialize the data and to get back a collection of **`Employee`** objects. In the end, we set the collection as **`ItemsSource`** of a **`DataGrid`** control, which is part of the Windows Community Toolkit.

Launching the Win32 process

By default, when you start a Universal Windows Platform application, Windows will execute only the main entry point. If you have included a Win32 process inside the package, it won't be launched automatically. As such, all the code we have written so far won't work, because the Win32 process won't be running. In order to launch it, we need to make a couple of changes to our application.

The first change must be done in the manifest. We need to declare the full path of the process we want to launch. However, this option can't be configured using the visual editor, so you will have to right-click the **Package.appxmanifest** file and choose **View code**. Remember that, in our context, the main entry point is the Windows Application Packaging Project. As such, exactly like we did to declare the app service, we need to make this change in the **Package.appxmanifest** file included in the Packaging Project, and not in the UWP app.

You should already have a **`windows.AppService`** entry in the **`Extensions`** section for the app service. You will need to add a **`windows.FullTrustProcess`** extension right below it, as shown in the following snippet.

Code Listing 47

```
<Extensions>
  <uap:Extension Category="windows.appService">
    <uap:AppService Name="DatabaseService" />
  </uap:Extension>
  <desktop:Extension Category="windows.fullTrustProcess"
Executable="ExpenseItDemo.Export\ExpenseItDemo.Export.exe" />
</Extensions>
```

The only required attribute is **`Executable`**, with the full path of the Win32 process inside the package. In order to use this extension, you will have to declare the **`desktop`** namespace in the **`Package`** definition.

Code Listing 48

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


<Package

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

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

  IgnorableNamespaces="desktop">


  . . .

</Package>
```

The second step is to launch the process using a special API included in the Universal Windows Platform. However, this API isn't implemented on all the supported Windows 10 devices, but only on traditional computers. Before using it, you need to add the specific desktop extension for the Universal Windows Platform to your project. Right-click the Universal Windows Platform application and choose **Add reference**. Select **Universal Windows** > **Extensions** and look for a library called **Windows Desktop Extension for the UWP.** You may have multiple versions of it if you have multiple versions of the Windows 10 SDK installed on your machine. Select the most recent one and click **OK**.

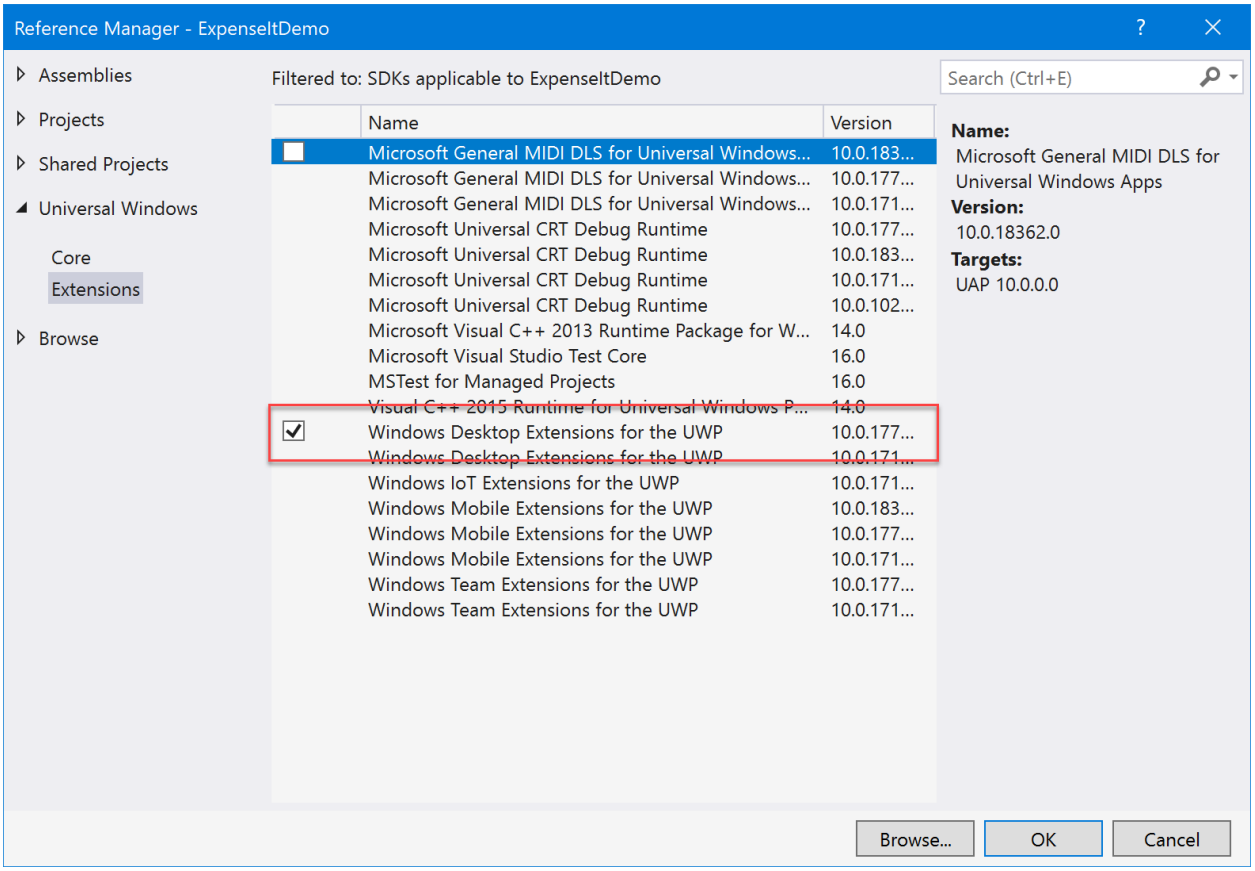


Figure 77: The UWP extension to support specific desktop features

Once you have added a reference to the library, you may add the required code to launch the Win32 process we declared in the manifest. It's up to you to choose where to add it, based on your requirements. A common scenario is to launch the process together with the main UWP application. In the following snippet, we launch it when the main page of the application is displayed.

Code Listing 49

```
protected async override void OnNavigatedTo(NavigationEventArgs e)

{

    if
(ApiInformation.IsTypePresent("Windows.ApplicationModel.FullTrustProcessLau
ncher"))

    {

        //we launch the Win32 process that generates the boarding pass on
the desktop.
```

```
        await
FullTrustProcessLauncher.LaunchFullTrustProcessForCurrentAppAsync();
    }
}
```

First we use the **`IsTypePresent()`** method exposed by the **`ApiInformation`** class. This technique is called **Adaptive Coding.** We use the **`Windows.ApplicationModel.FullTrustProcessLauncher`** class only if it's available on the current Windows 10 device. This way, if the application is launched on a device that doesn't support running Win32 processes (like Surface Hub or Xbox), the application won't crash, it simply won't execute the rest of the code.

To effectively launch the Win32 process, we must call the **`LaunchFullTrustProcessForCurrentAppAsync()`** method exposed by the **`FullTrustProcessLauncher`** class. We don't have to specify the process we want to launch, since the method will automatically leverage the one declared in the manifest.

That's it! Now the full architecture has been implemented. We have built a full UWP application that relies on a Win32 process to execute an operation that, otherwise, we wouldn't be able to achieve.

Chapter 7 Distribute Your MSIX Packages

One of the advantages of MSIX is its flexibility. It doesn't just open the doors to new distribution platforms like the Microsoft Store or the Microsoft Store for Business, it retains the ability to be deployed with a wide range of options that are commonly used in enterprises, like System Center Configuration Manager, Microsoft Intune, or simply a website or a file share.

In this chapter, we're going to explore two different distribution approaches—the Microsoft Store and the web—using a Windows 10 feature called App Installer. We're also going to see how you can use PowerShell to manage MSIX packages.

Distributing an application on the Microsoft Store

The Microsoft Store is a great distribution platform for applications that target consumers. The Microsoft Store, in fact, is built into Windows 10, and it provides users easy access to a wide catalog of applications. With the advent of the Desktop Bridge in the Windows 10 Anniversary Update, the Microsoft Store also opened the doors to classic desktop applications packaged as AppX or MSIX. Spotify, Slack, and iTunes are just some examples of Win32 applications that can be easily downloaded from the Store.

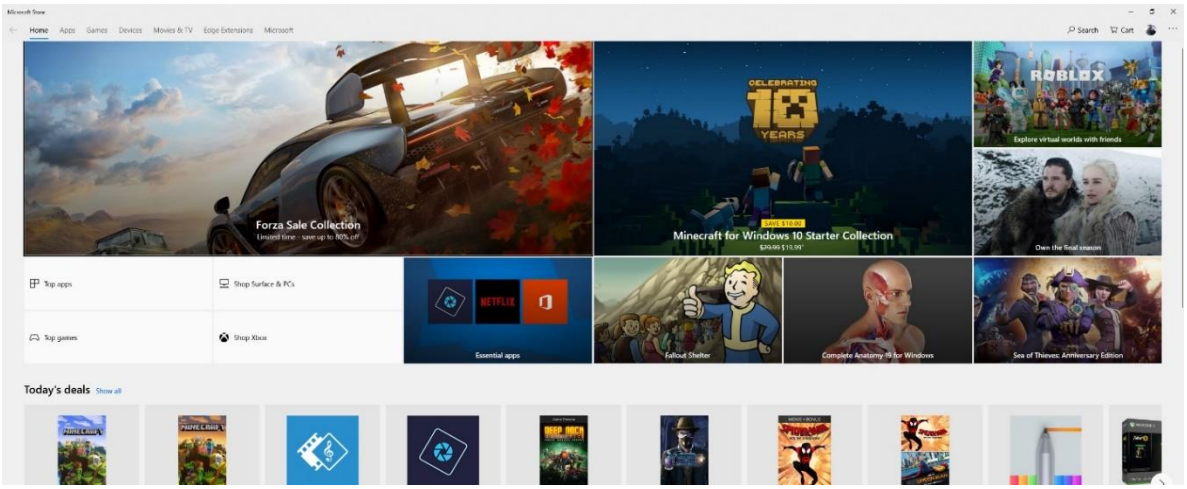


Figure 78: The Microsoft Store

The Store helps to solve many of the challenges that developers must face when they want to distribute applications:

- You don't have to handle the monetization. The purchase flow will be handled directly by Microsoft and, every month, you will receive your earnings, with the Store fee deducted.
- You don't have to handle licensing. You can enable features like trials or in-app purchases, and Windows will make sure that users won't be able to circumnavigate them and use the application without paying for the license.
- You don't have to handle updates. By default, automatic updates are turned on in the Microsoft Store, so users will always use the latest version of your application.

Additionally, the Microsoft Store includes many features that can be helpful for improving the development experience:

- You can create flights, subsets of users (identified by the Microsoft account) that will receive different versions of the application. With this feature, you can create multiple test rings (alpha, beta, production, etc.) and deliver preliminary versions of your application to your testers. Once an update has been validated, you can just move it from one ring to another to expand your audience.
- You can create private applications that only a specific set of users (identified by the Microsoft account) will be able to see and download.
- You can enable gradual rollout, so that only a random subset of users will get the update first. This way, in case of unexpected issues, you can quickly stop the distribution and fix the problem before resuming.

The submission process is managed through the Microsoft Partner Center dashboard, which is a web portal that allows developers to submit, validate, and update their applications.

Registering a developer account

To publish applications, you need to register yourself as an app developer, which is a single account that lets you submit apps and add-ins for the entire Microsoft ecosystem—not just Windows apps, but also Office add-ins on the Office Store or on the Azure Marketplace. The account costs $19 for individual developers and $99 for companies. It's a lifetime subscription, which you need to pay only once. However, if you have a Visual Studio subscription, among the available benefits, you will find a token to register for free.

You can start the procedure to create a developer account here. You will be asked for information about you or your company, based on which kind of account you want to open. The procedure is the same, and you'll be able to pay using a PayPal account or a credit card. The only difference is that:

- If it's a personal account, it will be immediately activated after the payment has been successfully completed.
- If it's a company account, it takes a bit longer to get the account activated. A Microsoft partner will contact you and request additional information to prove that the company really exists, and that you are legally allowed to operate on behalf of it.

Creating a package for the Microsoft Store

Thanks to Visual Studio, it's easy to create a package for the Microsoft Store. Whether it's a full Universal Windows Platform application or a classic application packaged using the Windows Application Packaging Project, by right-clicking the project, you will see the option **Store** > **Create app packages**. The first question you will be asked is if you want to create a package for the Microsoft Store or for sideloading. If you choose the first option, you will be asked to log in with the Microsoft account connected to your developer account. Visual Studio will show you a list of app names you have already registered, and it will give you the option to create a new one. This name must be unique, and it will be used to establish the identity of your package.

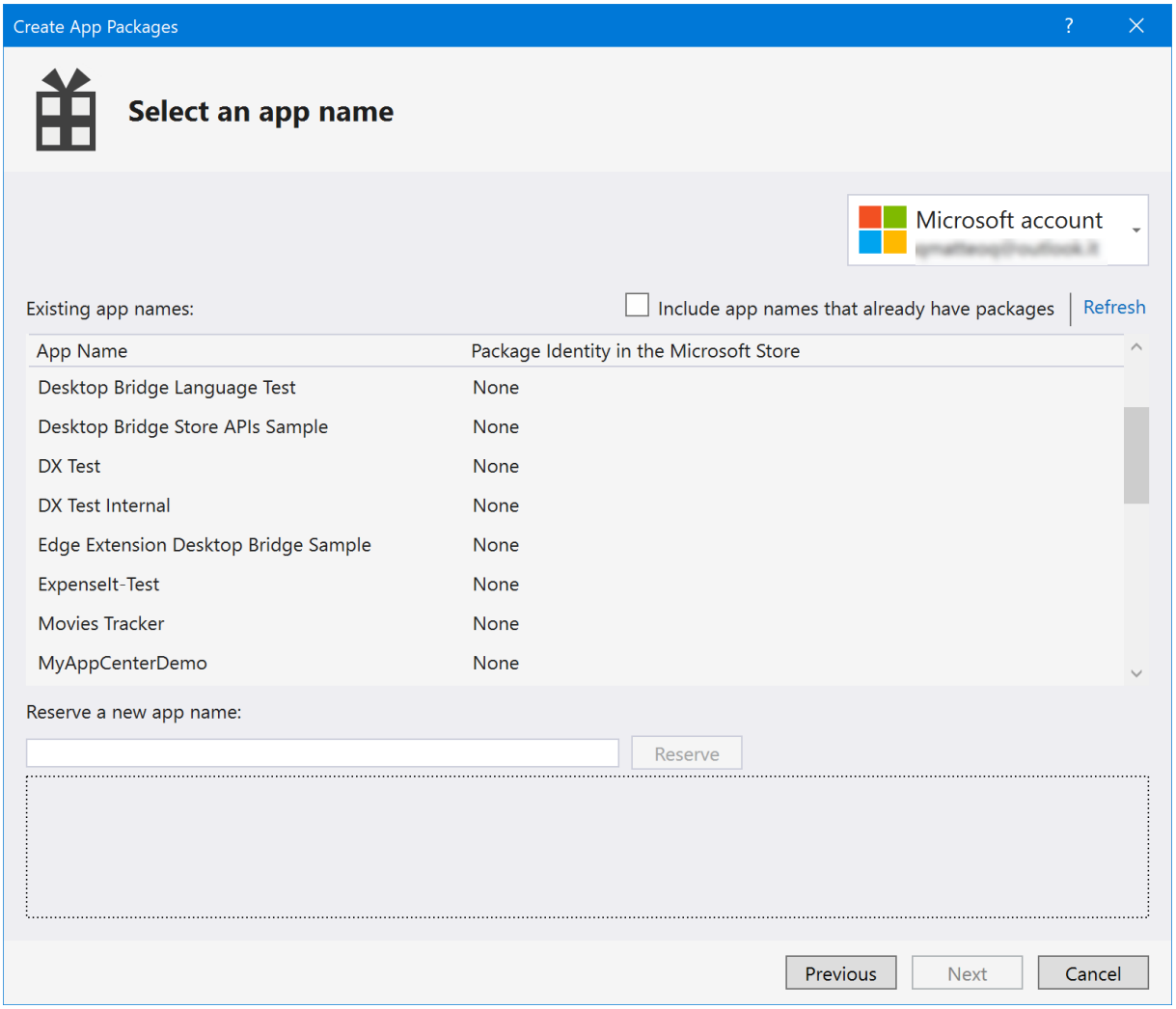


Figure 79: You must associate the application to an app name before generating a package for the Store

The rest of the wizard is the same as the one we saw in Chapter 2 when we talked about the Windows Application Packaging Project. You will be asked to set the version number and define which packages you want to create: target CPU architecture, configuration mode, etc.

At the end of the process, you will obtain a file with the extension **.**msixupload (or .appxupload, if you're targeting a Windows 10 version earlier than 1809). Once you have it, you can go to the Microsoft Partner Center website and log in with your developer account. In the applications list, you will find the name you have reserved through the Visual Studio wizard. Just click **Start your submission** to start the submission process.

You will be guided through a series of steps:

- **Pricing and availability:** In this section, you can choose the price at which you want to sell your app, and how to distribute it. You can choose to make it available only in a restricted number of countries, to distribute it to everyone, or to opt-in for private distribution. You can define a specific schedule for when to make the application available.
- **Properties**: In this section, you can choose the category of your application.

- **Age ratings**: In this section, you will have to complete a survey, which will be used to assign an age rating to your application. This is required to be compliant with the various laws and certifications around the world regarding age restrictions for software products.
- **App packages**: In this section, you will need to upload the **.msixupload** package that Visual Studio has generated for you.
- **Store listings**: In this section, you can provide the metadata of your application, which will be used to populate the Store page, such as a description, screenshots, a video trailer, and keywords.
- **Submission options**: In this section, you can provide notes for the tester in case the application has any special requirements, like credentials to log in, or dependency from a hardware device.

At the end of the process, you will be able to submit the application on the Microsoft Store. The submission will trigger a certification process that will validate your application to make sure it's compliant with the Store technical requirements and policies. The certification time may vary; in some cases, only automated tests are performed, which means that the application will be validated after a few hours. In some other cases, the application may be picked up for manual testing, which can take up to five business days.

Restricted capabilities

When you submit a classic Windows application packaged as MSIX, there's a caveat. If you recall what we learned in Chapter 1, these applications leverage a restricted capability called **runFullTrust**, which allows them to run outside the sandbox leveraged by full Universal Windows Platform applications. Being a restricted capability, it isn't granted by default to Store applications. As such, when you upload a **.msixupload** package containing a manifest that declares this capability, you will see a new step in the submission process. You will be asked to specify the reason why you're requesting this capability. Usually, it's enough to specify that your application is a Win32 app and, as such, it can't work properly without this capability.

The Microsoft Store for Business/ Education

The Microsoft Store for Business/Education is a private version of the Microsoft Store that can be leveraged by companies and schools. From a technical point of view, it uses the same infrastructure of the Microsoft Store. However, the environment is controlled by the administrator of the company or school, and it allows them to create a private and curated catalog of applications.

Being dedicated to companies and school, unlike for the public Store which requires a Microsoft Account, the access to the Store for Business/Education is granted using an Azure Active Directory account. Users will be able to log in using their work or school accounts.

After logging in to the Store with such an account, you will see a new tab with the name of your organization. When you click on it, you will see a curated list of applications selected by your company.

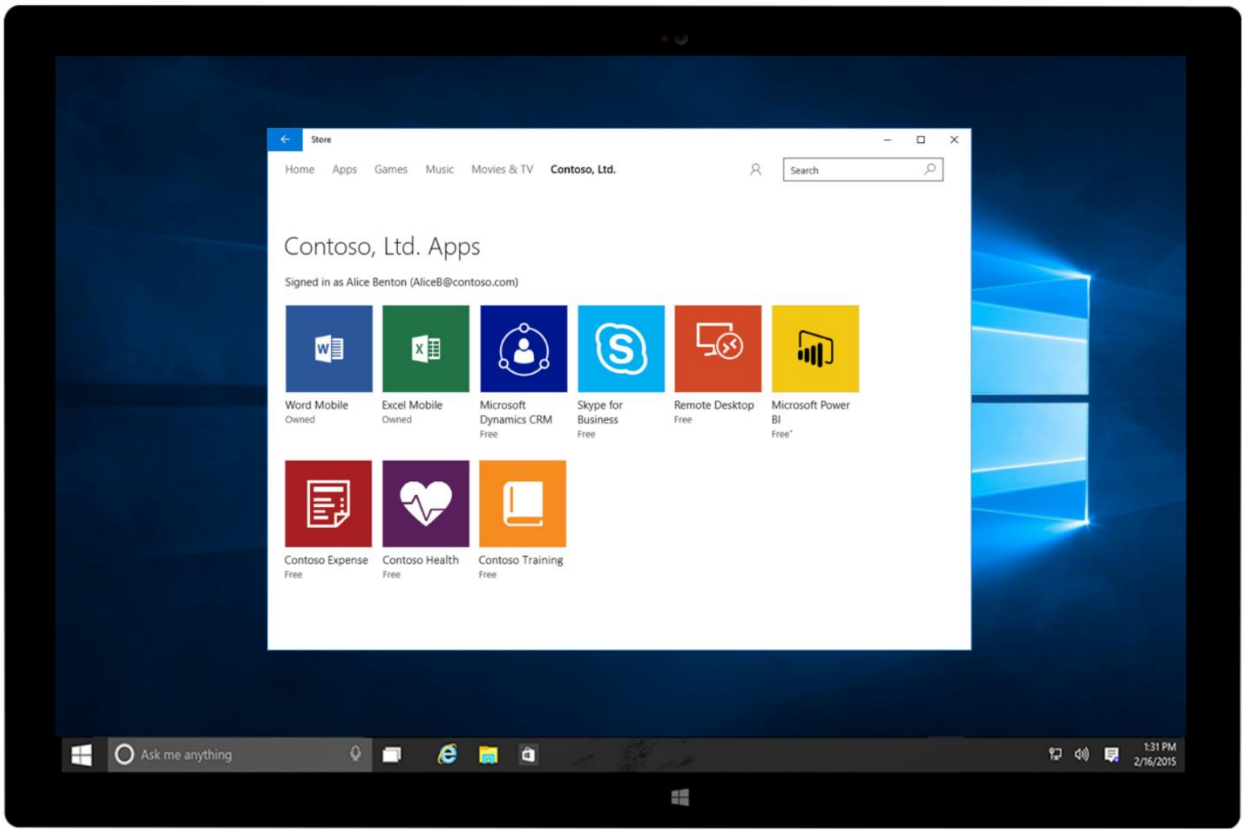


Figure 80: The Microsoft Store for Business

The administrator of the organization can use the web portal to acquire applications and add them to the internal catalog.

The acquisition models

The Store for Business supports the acquisition of free apps and paid apps. For both categories, there are two different acquisition models:

- **Online licensing:** Unlike the Microsoft Store, where the purchase is individual, the Store for Business supports bulk purchases of licenses. The administrator can purchase an arbitrary number of licenses and then assign them to a group of users within the organization. With the online model, licenses are tracked directly by the Microsoft Store, so you don't have to worry about their management. You just need to choose the group of users that will receive them. This licensing model is very flexible. If an employee or a student leaves the organization, the license isn't lost, and can be easily transferred to another person. This licensing model works only in combination with Azure Active Directory.
- **Offline licensing**: This approach can be useful for companies that are using tools like SSCM to deploy applications. They don't support Azure Active Directory and, as such, they can't access the Microsoft Store. In this scenario, the web interface of the Store for Business will simply allow the administrator to download the MSIX package, which then will need to be manually deployed on the users' machines. With this approach, you can't

leverage the automatic license tracking; it will be up to you to make sure you have purchased the right number of licenses for your users.

Publishing an application on the Microsoft Store for Business

As a developer, you have two approaches to make an application available on the Store for Business. If you have developed a general-purpose application, which can be appealing for many enterprise customers, you can publish it on the public Store and turn on the organizational licensing distribution options. With this approach, your application will be searchable not only on the public Store, but also on the Store for Business dashboard. Administrators will be able to find your app and acquire it, so that it can be added to the internal catalog.

To achieve this goal, you need to make sure that, during the submission process on the Partner Center, you enable two options in the first step, which is **App pricing and availability**:

- **Make my app available to organizations with Store-managed (online) licensing and distribution**: This option will enable your application to be available through the Store for Business using the online licensing model. This option is turned on by default.
- **Allow organization-managed (offline) licensing and distribution for organizations**: This option will enable your application to support the offline licensing model, which allows distribution outside the Store. This option is turned off by default.

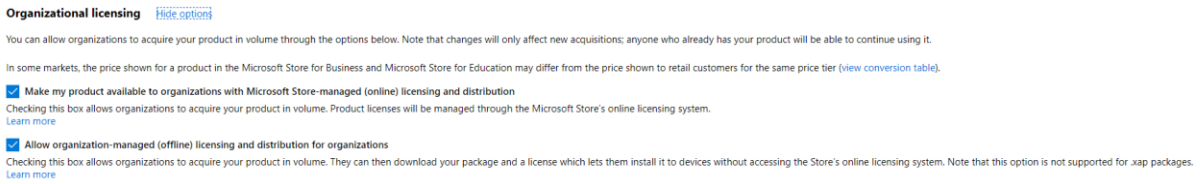


Figure 81: The options during the submission process to enable distribution through the Microsoft Stores for Business and Education

Note: You can publish an application on the public Store and opt out from publishing on the Store for Business and Education, but not the other way around. If you want to publish an application on the Store for Business, you must also publish it on the public Store, unless it's a private line-of-business app.

Another scenario is line-of-business applications that have been developed or customized for a specific customer. In this case, deploying this application on the public Store wouldn't make sense, because it can't be used by a generic customer. To satisfy this requirement, the Store for Business and Education enables administrators to invite a developer to publish applications only in their private catalog. This goal can be achieved from the **Permissions** section of the web dashboard, which offers a tab called **Line-of-business publishers**. When you click the **Invite** button, you will have to specify the email address associated with the Partner Center account of the developer that you would like to invite.

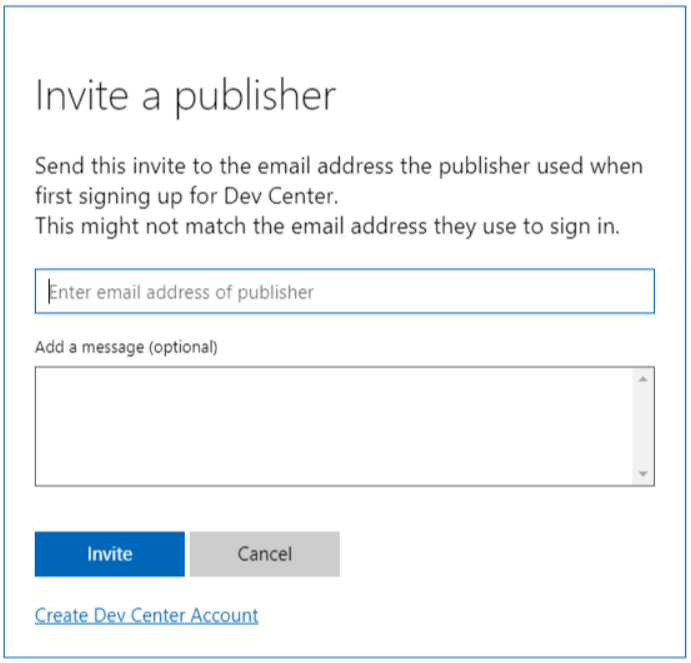


Figure 82: An administrator can invite a developer to publish applications only in a private catalog

The developer will receive an email, and upon accepting the invitation, will see a new option in the **Pricing and availability** section of the submission process.

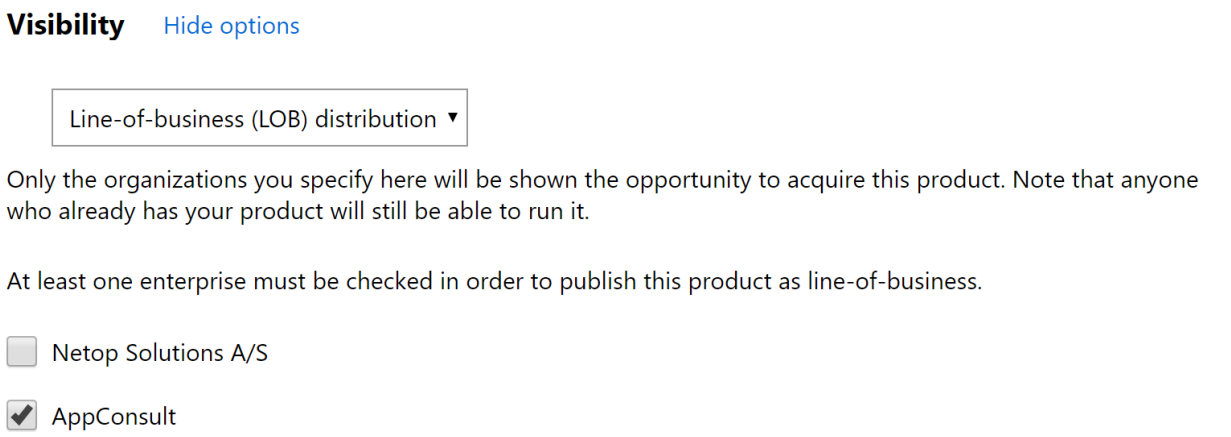


Figure 83: A developer has the option to submit an application only on the private store of an organization

In the **Visibility** section, there will be a new drop-down menu where the developer will be able to choose **Line-of-business (LOB) distribution.** The page will list all the organizations that have sent an invitation. The developer just needs to check the ones for which they're publishing the application, and then continue the submission as usual. At the end of the process, the application will be available only in the private catalog of the company, and not searchable by other users and companies.

This publishing approach gives you more flexibility over the certification process. Since the application won't be published on the public Store, all the restricted capabilities will be automatically allowed, and there won't be a real certification process, except for the basic technical checks. It will be up to the administrator of the organization to validate the application and to make sure it's compliant with the organization's requirements.

Controlling access to the Microsoft Store

As an enterprise, you can control the access to the Store for your users using group policies. You can find them by opening the **Group Policy Editor**, which can be launched by clicking the **Start** button and typing **`gpedit`**. The policies are available under **Computer Configuration** > **Administrative Templates** > **Windows Components** > **Store.** If you aren't using the Microsoft Store or the Microsoft Store for Business, you can disable it completely using the **Turn off Store application** policy.

If, instead, you want to leverage the Microsoft Store for Business, but you don't want to give your users access to the public Microsoft Store, you can turn on the **Only display the private store within the Microsoft Store** policy. In this case, users will be able to open the Store application, but they will see only the private catalog defined by the enterprise. They won't have access to the public catalog.

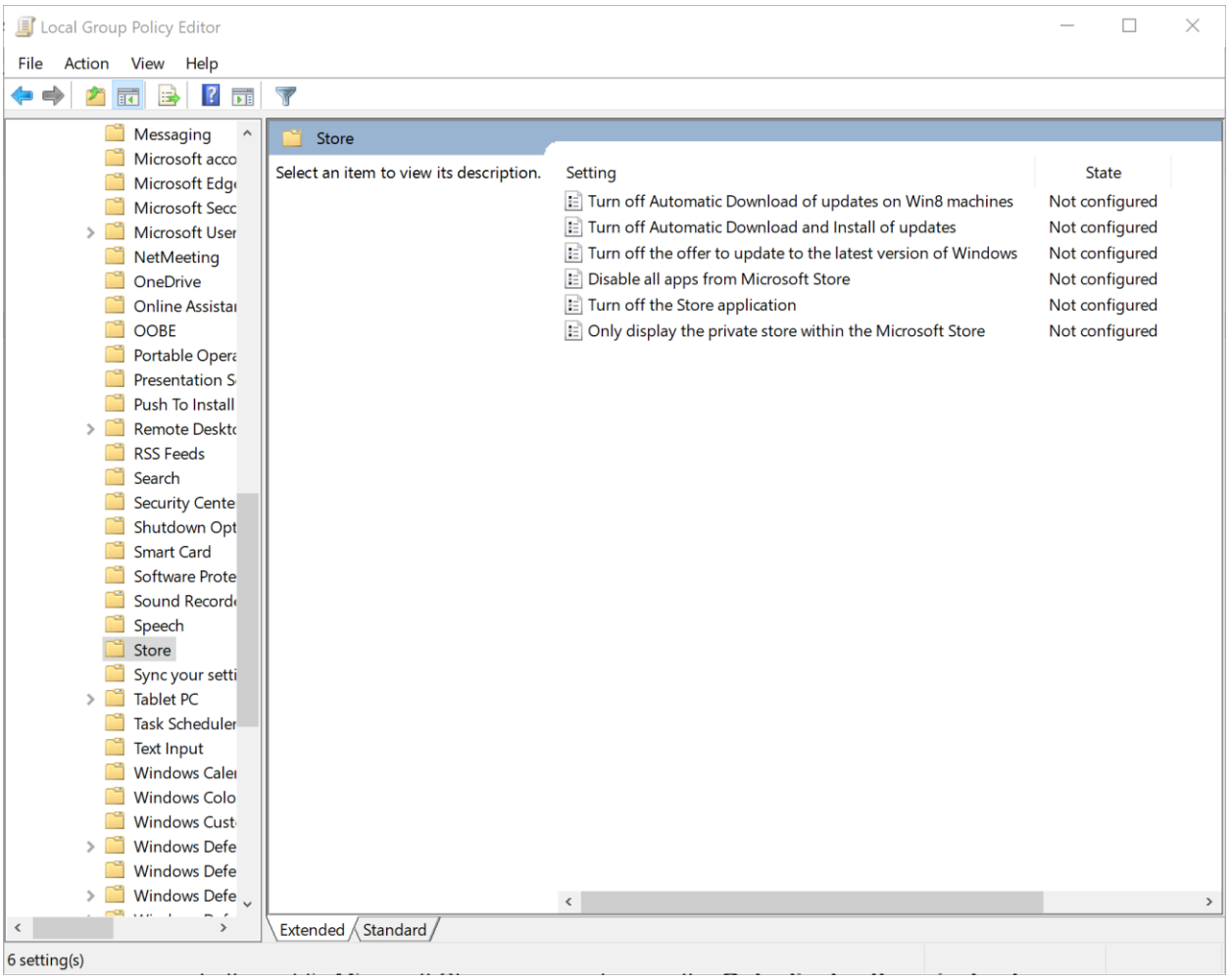


Figure 84: The policies to control access to the Microsoft Store

App Installer

The Store is a great distribution channel, but it may not always be the best fit for your scenario. You may need to deploy the application in an enterprise that hasn't adopted the Store for Business, or you may want to start an internal testing of a new app. In any case, you may be interested in retaining some of the advantages of Store distribution, like automatic updates.

Thanks to a Windows 10 feature called **App Installer**, you'll be able to retain some of the Store features by deploying the application on a website or a file share. Let's see how it works!

App Installer is the name of the technology in Windows 10 that makes it easy to deploy MSIX packages. In the past, PowerShell was the only option for deploying a package without using the Store. Thanks to App Installer, now you can just double-click an MSIX package and proceed with the deployment with the click of a button.

App Installer comes with support for the **ms-appinstaller** protocol, which you can use on a website or a file share to trigger the deployment of an MSIX package. When the user clicks on such a link in a browser, instead of having to manually download and install the MSIX package, Windows will immediately trigger the familiar user interface. In order to enable this feature, the **ms-appinstaller** protocol must point to a file with the .appinstaller extension. It's an XML file that describes the package and contains information like the URL where it has been deployed.

Here is a sample file.

Code Listing 50

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package.appinstaller"
Version="1.0.5.0"
xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
  <MainBundle Name="ContosoExpenses" Version="1.0.5.0"
Publisher="CN=AppConsult" Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package_1.0.5.0_x86.msixbundle"
/>
</AppInstaller>
```

The **`AppInstaller`** entry includes a property called **`Uri`**, which contains the full URL where the file itself has been deployed. The package is described with the **`MainBundle`** entry, which includes the name, version number, publisher, and the full URL where the package has been uploaded. All this information (except for the URL) must match the information included in the manifest of the application. In this case, we're using the **`MainBundle`** entry because we're deploying a bundle that has the **.**msixbundle extension. If, instead, our application is made by a single package with the **.**msix extension, we can use the **`MainPackage`** entry, as shown in this example.

Code Listing 51

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package.appinstaller"
Version="1.0.5.0"
xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
  <MainPackage Name="ContosoExpenses" Version="1.0.5.0"
Publisher="CN=AppConsult" Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package_1.0.5.0_x86.msix" />
</AppInstaller>
```

Once you have created such a file, you can link to it using the **ms-appinstaller** protocol and reference it from a webpage.

Supporting additional packages

The App Installer file allows you to declare not just the main package, but also additional packages that must be installed together with the main application. This way, the user will be able to install everything needed to run the application with just one click. App Installer supports three categories of additional packages: optional packages, modification packages, and dependencies.

Optional packages are declared inside the **`OptionalPackages`** section, which can contain **`Bundle`** items (in the case of **.msixbundle** files) or **`Package`** items (in the case of **.msix** files). The following sample shows the declaration of an App Installer file that includes a main package and an optional package.

Code Listing 52

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package.appinstaller"
Version="1.0.5.0"
xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
  <MainPackage
    Name="ContosoExpenses"
    Version="1.0.5.0"
    Publisher="CN=AppConsult"
    Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package_1.0.5.0_x86.msix" />
  <OptionalPackages>
    <Package
        Name="ContosoExpenses.Addon"
        Publisher="CN=AppConsult"
        Version="1.0.0.0"
        Uri="https://db-msixtest.azurewebsites.net/ContosoExpenses-
Addon.msix" />

  </OptionalPackages>
</AppInstaller>
```

Modification packages work in the same way. The only difference is that **`Package`** or **`Bundle`** items must be declared inside the **`ModificationPackages`** section, as in the following sample.

Code Listing 53

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package.appinstaller"
```

```
Version="1.0.5.0"
xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
  <MainPackage
    Name="ContosoExpenses"
    Version="1.0.5.0"
    Publisher="CN=AppConsult"
    Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package_1.0.5.0_x86.msix" />
  <ModificationPackages>
    <Package
        Name="ContosoExpenses.Customization"
        Publisher="CN=AppConsult"
        Version="1.0.0.0"
        Uri="https://db-msixtest.azurewebsites.net/ContosoExpenses-
Customization.msix" />

  </ModificationPackages>
</AppInstaller>
```

In the end, the App Installer file also supports dependencies, like the Visual C++ Runtime libraries. They are declared using the **Dependencies** section, and they work in a similar way as modification packages. You need to add one (or more) **Package** entry and describe all its features, including the path where you have deployed it, as in the following sample.

Code Listing 54

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package.appinstaller"
Version="1.0.5.0"
xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
  <MainPackage
    Name="ContosoExpenses"
    Version="1.0.5.0"
    Publisher="CN=AppConsult"
    Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package_1.0.5.0_x86.msix" />

  <Dependencies>
    <Package Name="Microsoft.VCLibs.140.00" Publisher="CN=Microsoft
Corporation, O=Microsoft Corporation, L=Redmond, S=Washington, C=US"
Version="14.0.24605.0" ProcessorArchitecture="x86" Uri="https://db-
msixtest.azurewebsitesfwkx86.appx" />
  </Dependencies>
```

```
</AppInstaller>
```

Typically, once the dependency is declared inside the app manifest, the Visual C++ runtime libraries are automatically downloaded from the Microsoft Store. However, since when you use App Installer you might be in a scenario where the Store is blocked, you need to also specify where you have deployed the package by using the **`Uri`** attribute.

Supporting automatic updates

Thanks to App Installer, you can also enable automatic updates. As a developer, all you need to do is copy in the same location (website or file share) an updated MSIX package and an updated App Installer that references the update. All the users who have downloaded the application from that location using the **ms-appinstaller** protocol will automatically get the update, based on the criteria you have chosen. They won't have to go back to the webpage to manually trigger the update.

The auto update feature is controlled by a setting in the App Installer file, as demonstrated in the following snippet.

Code Listing 55

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package.appinstaller"
Version="1.0.5.0"
xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
  <MainBundle Name="ContosoExpenses" Version="1.0.5.0"
Publisher="CN=AppConsult" Uri="https://db-
msixtest.azurewebsites.net/ContosoExpenses.Package_1.0.5.0_Test/ContosoExpe
nses.Package_1.0.5.0_x86.msixbundle" />
  <UpdateSettings>
    <OnLaunch HoursBetweenUpdateChecks="0" />
  </UpdateSettings>
</AppInstaller>
```

Compared to the previous snippet, we have added a new section called **`UpdateSettings`** that contains the **`OnLaunch`** entry. Thanks to this option, the application will check for updates every time it's launched. With the **`HoursBetweenUpdateChecks`** attribute, you can specify the frequency between each check. **`0`** means that updates will be checked every time. If you set, for example, **`3`**, it means that if you open the app multiple times over a period of time, Windows won't check for new updates before three hours have passed. If a new update is found, Windows will automatically download it. However, to avoid data loss or work disruption, it will apply it only once the application has been closed. The next time the user reopens the application, the new version will be used.

Adding an interactive prompt

Starting from Windows 10 version 1903, the **OnLaunch** feature also supports an interactive UI that will warn the user there's a pending update before launching the app. This option can be enabled by adding the **ShowPrompt** attribute, as in the following sample.

Code Listing 56

```
<UpdateSettings>
  <OnLaunch HoursBetweenUpdateChecks="0" ShowPrompt="True" />
</UpdateSettings>
```

When this feature is turned on, users will see the following UI when they open the application and there's a new version on the server:

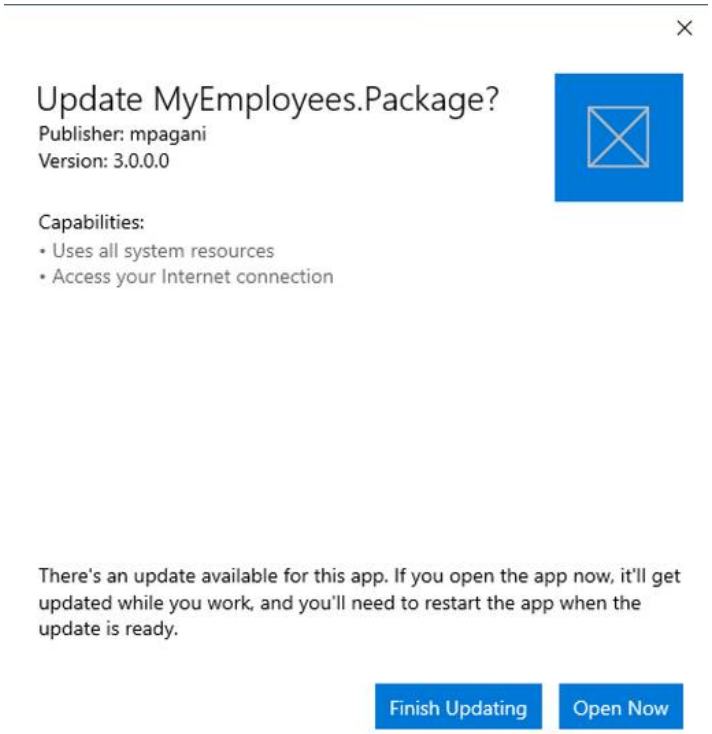


Figure 85: The message displayed to the user when there's a new app update

Users will have the option to continue with the current version and apply the update later, once they close the app, or to apply it immediately before starting to use it.

Handling critical updates

When the **ShowPrompt** attribute is set to **True**, you can also leverage another feature, which is the ability to declare an update as critical. This goal can be achieved with the **UpdateBlocksActivation** attribute, as in the following snippet.

Code Listing 57

```
<UpdateSettings>
  <OnLaunch HoursBetweenUpdateChecks="0" ShowPrompt="True"
UpdateBlocksActivation="True" />
</UpdateSettings>
```

When this feature is turned on, users will see the following UI if an update is available on the server.

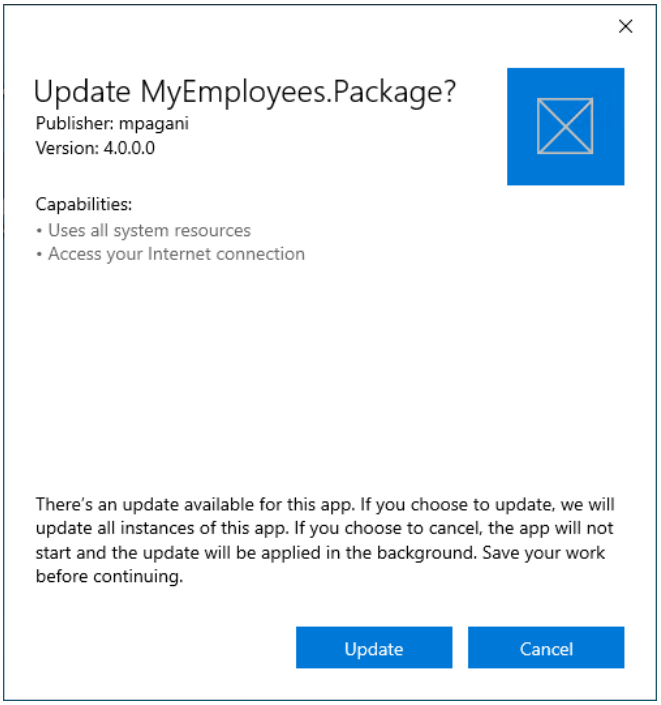


Figure 86: The message displayed to the user when there's a critical update for the application

As you can see from the messaging, the user doesn't have the option to postpone the update process anymore. The update will be applied regardless. Users can only choose if they want to automatically open the application or not after the update has been applied. This feature is helpful when you deliver an update that contains a critical security fix, or when you change something in the back end and the current version of the application won't work anymore.

Supporting downgrades

By default, MSIX packages can be only updated with a package with the same identity and a higher version number. Windows will block you if the new package has a lower version number. However, in some scenarios, you may need to roll back to a previous version, for example, if the new version introduced a critical bug that will take some time to be resolved and waiting for an updated package isn't an option.

In this case, you can enable the **`ForceUpdateFromAnyVersion`** option in the **`UpdateSettings`** section.

Code Listing 58

```
<UpdateSettings>
  <OnLaunch HoursBetweenUpdateChecks="0" ShowPrompt="True"
UpdateBlocksActivation="True" />
<ForceUpdateFromAnyVersion>true</ForceUpdateFromAnyVersion>
</UpdateSettings>
```

However, even with this feature turned on, you won't be able to automatically perform a downgrade. If you publish an MSIX package with a lower version number on the website or file share, the package won't be automatically downloaded and installed. It will be up to users to return to the webpage from which they downloaded the application the first time and click again on the button linked to the **ms-appinstaller** protocol. The difference is that, this time, Windows won't block the installation, it will just warn users that they're trying to install an update over a newer version.

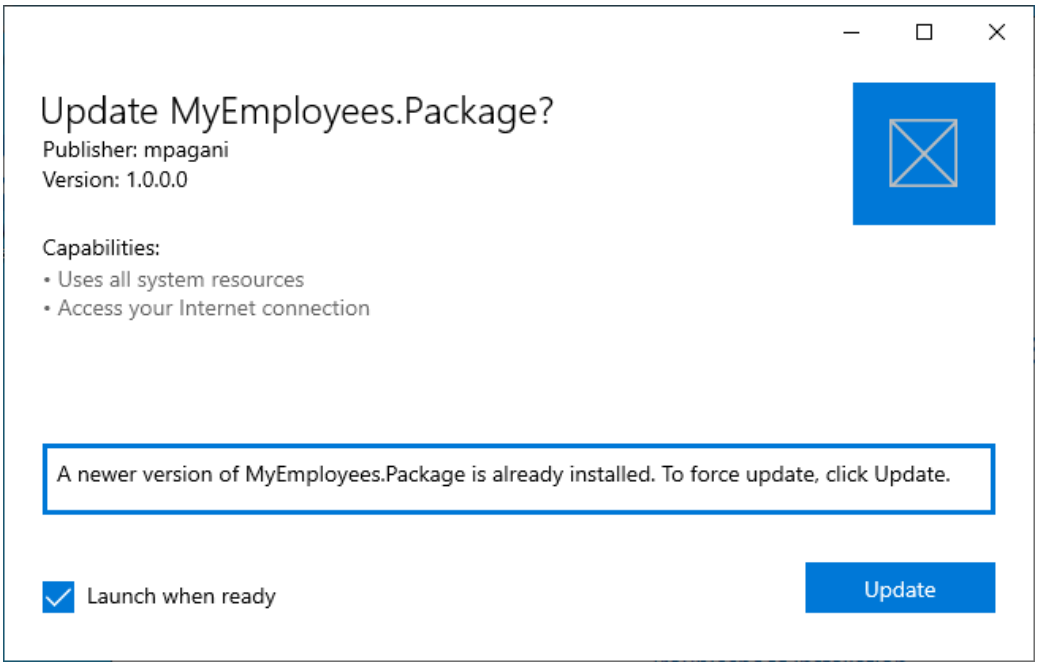


Figure 87: The message displayed to users who attempt to downgrade an existing application

Installing updates in background

All the options we have seen so far are enabled by the **`OnLaunch`** configuration, since it supports displaying a UI to the user before triggering an update. However, in some scenarios you might just want to keep the application always up to date, regardless if the user has launched it or not. In this case, you can use the **`AutomaticBackgroundTask`** option, as in the following snippet.

Code Listing 59

```
<UpdateSettings>
  <AutomaticBackground />
</UpdateSettings>
```

With this configuration, Windows will check for updates in the background every eight hours, regardless of when the user has last launched the application.

Generate an App Installer file

An App Installer file is nothing more than an XML file, so you can manually create it following the guidelines included in the official documentation. You will be able to reference not only main MSIX packages, but also dependencies and modification packages. This way, the user will automatically install everything required by the application to run as expected.

Visual Studio can help you by generating an App Installer file for you during the package creation wizard, which you can trigger by right-clicking the UWP or Windows Application Packaging Project and choosing **Store** > **Create app packages**. As the first question, you're asked if you want to create a package for the Store or for sideloading. In the second scenario, you can select the check box called **Enable automatic updates**.

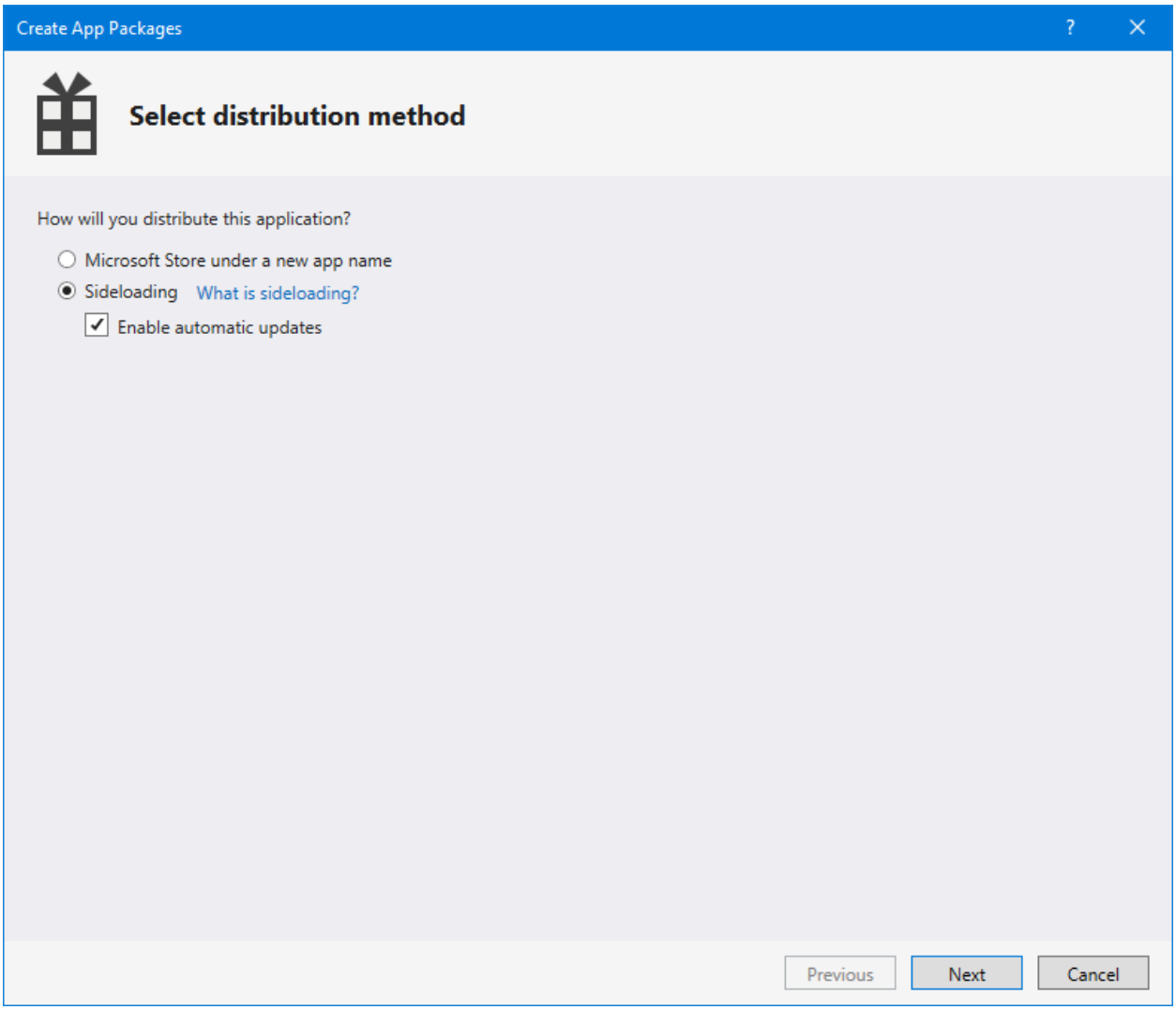


Figure 88: The option in Visual Studio to create a package for sideloading with automatic updates

Once you enable this option, you will have the opportunity to specify the URL where the package will be deployed (it can be a web URL or a file share path) update frequency.

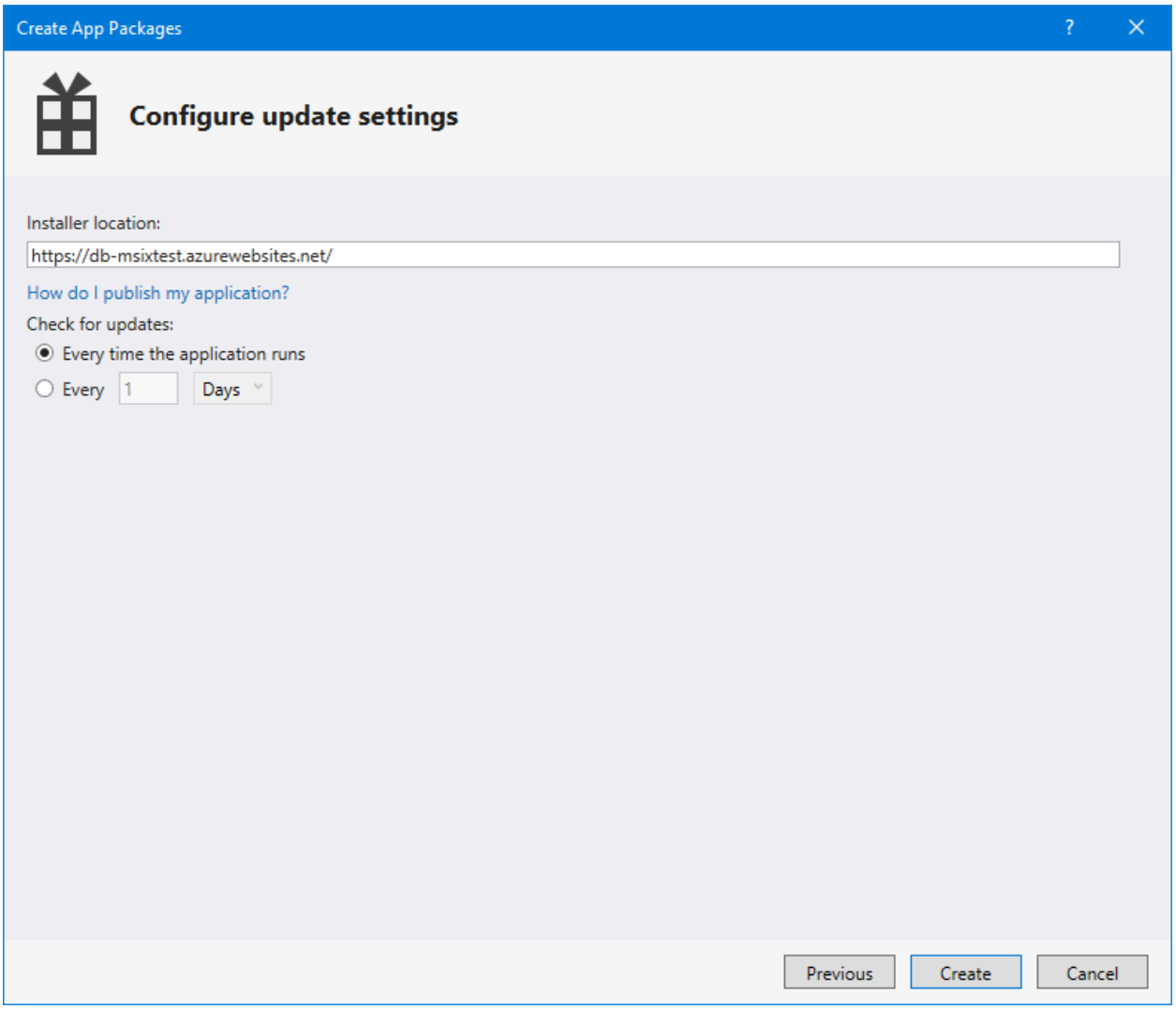


Figure 89: The option to configure automatic updates during the package creation wizard

At the end of the process, Visual Studio will generate, in addition to the folder with the package, a file with an .**appinstaller** extension and a default HTML page with a link to install it using the **ms-appinstaller** protocol.

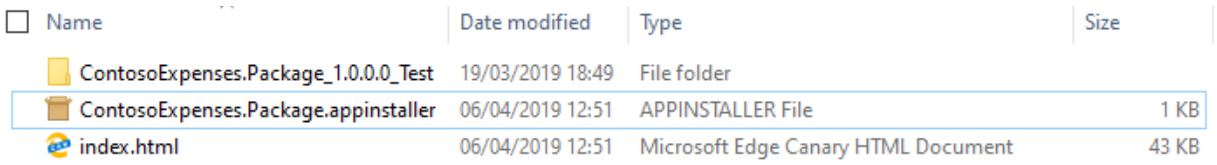


Figure 90: Visual Studio when you create an MSIX package for sideloading with the automatic check for update feature enabled

Your task will be to copy the contents of this folder to your website or file share path. Your users will just have to open the URL to see the webpage and trigger the installation of the application. The generated HTML page contains some useful information, like the app version, the publisher, and the logo based on the application's assets.

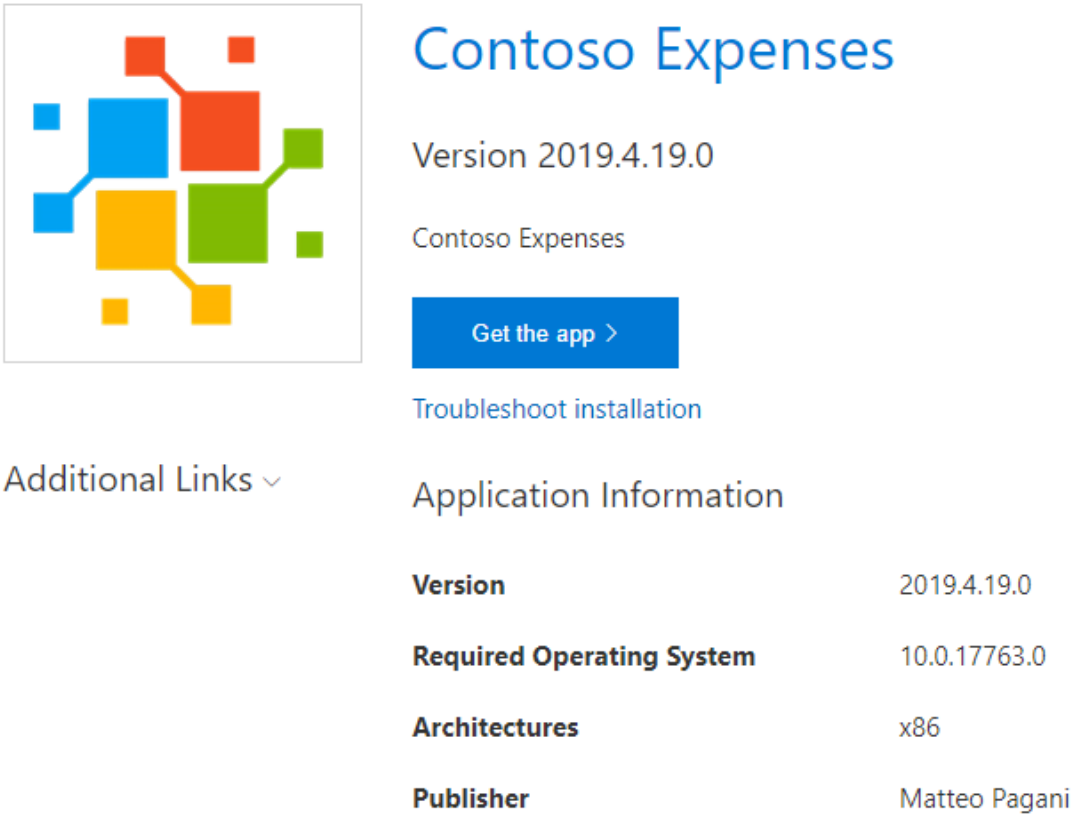


Figure 91: The page generated by Visual Studio to install the MSIX package using the ms-appinstaller protocol

The wizard has a downside, however. You can specify only a few options, like the update frequency and the installer location. What if you want to enable one of the new features we have just explored, like the ability to show a prompt or to mark an update as critical? You would need to manually edit the generated App Installer file after the package creation has been completed. However, this approach has many downsides, especially if you're planning to include an automation system (like a CI/CD pipeline, as we're going to see in Chapter 8) in order to automatically deploy the generated package.

Starting from Visual Studio 2019 version 16.2, you have the opportunity to add an App Installer template to your project, which you can customize. The relevant information (like the installer URL and the package identity) will be replaced at runtime during the package creation. To add a template, right-click your Windows Application Packaging Project (or directly in the UWP one, if it's a full UWP application) and choose **Add** > **New item.** You will find a template called **App Installer**.

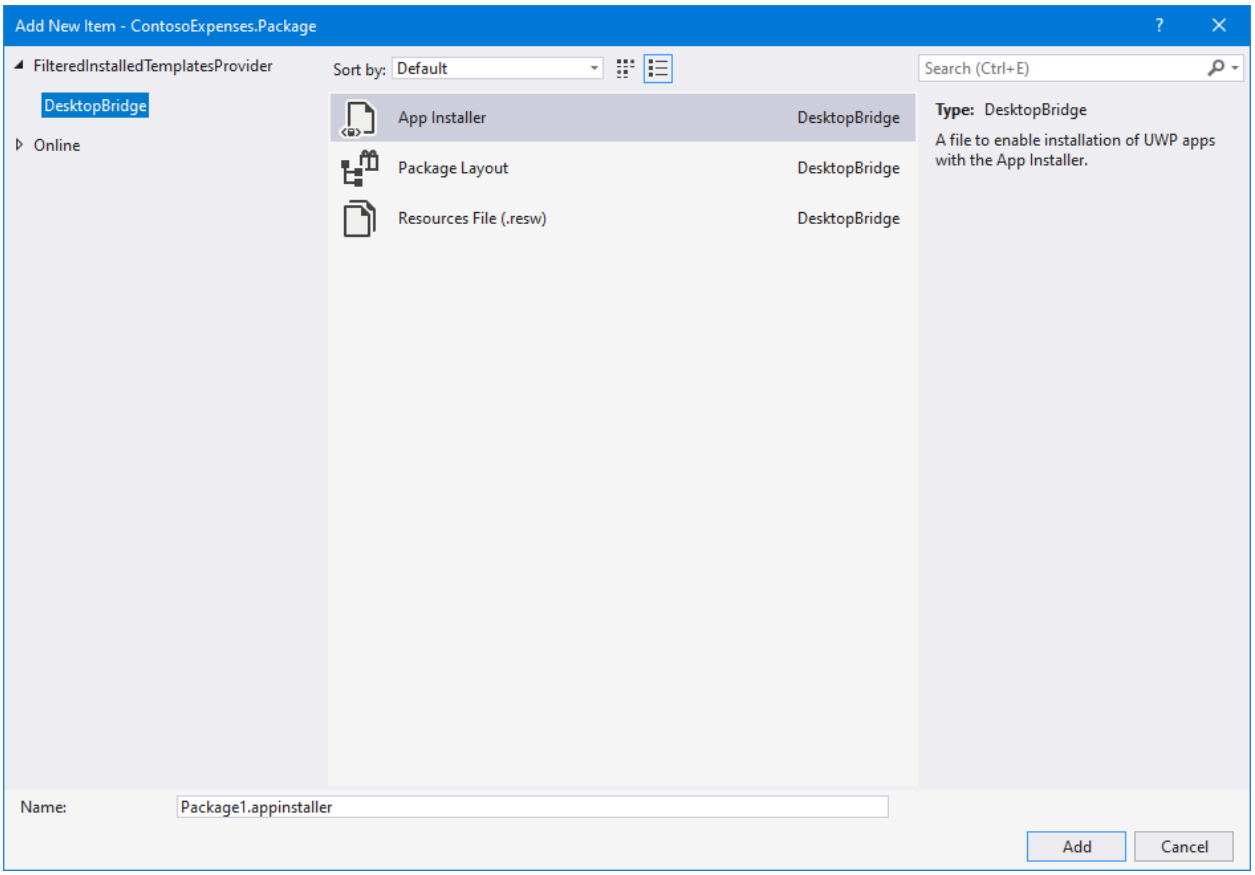


Figure 92: The template to generate an App Installer file

The file will look like the following.

Code Listing 60

```
<?xml version="1.0" encoding="utf-8"?>
<AppInstaller Uri="{AppInstallerUri}"
              Version="{Version}"

xmlns="http://schemas.microsoft.com/appx/appinstaller/2018">

  <MainBundle Name="{Name}"
              Version="{Version}"
              Publisher="{Publisher}"
              Uri="{MainPackageUri}"/>

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

</AppInstaller>
```

As you can see, all the custom information is included with a placeholder, like **{AppInstallerUri)** or **{Version}**. These values will be replaced during the package generation, with the ones that are set in your manifest and that you have specified during the wizard. This way, you can customize the configuration (for example, by adding the **ShowPrompt** attribute to the **OnLaunch** item) and let Visual Studio automatically generate the correct App Installer file when you create a new MSIX package.

You will notice that the template is correctly recognized because, when you try to generate a new package, you will see the following notification in the wizard.

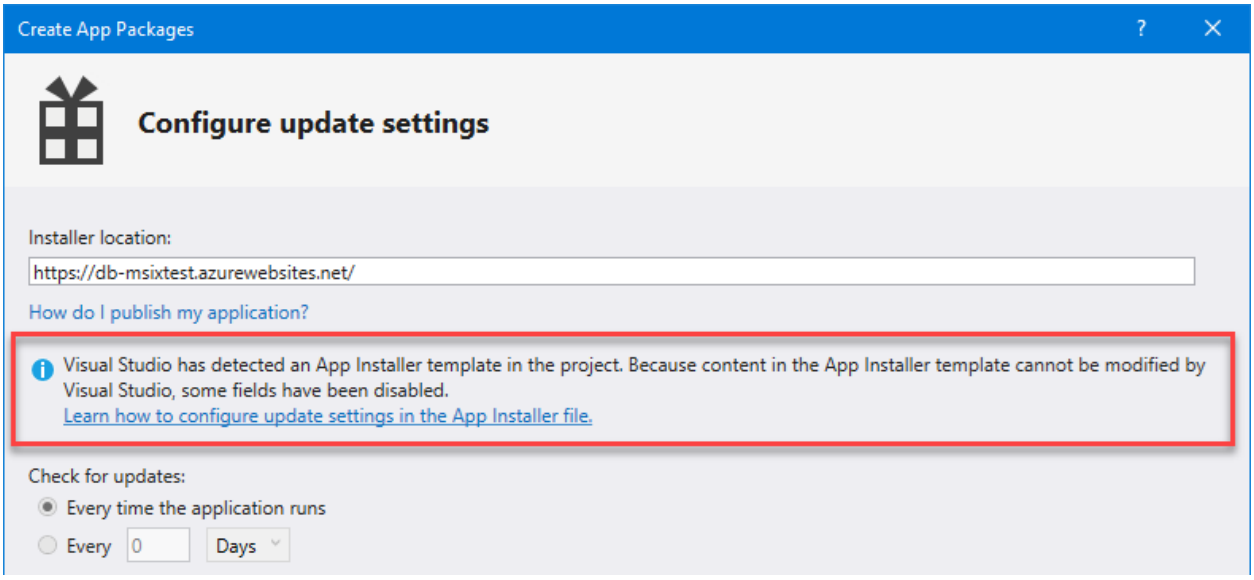


Figure 93: Visual Studio informs the user that an App Installer template is included inside the project

Another option is to leverage an open-source project created by Microsoft, called MSIX Toolkit, which includes many helpers to facilitate the work of IT pros and developers who are adopting MSIX. Among the tools, you will find one called **AppInstaller File Builder**, which provides a convenient user interface to generate an App Installer file.

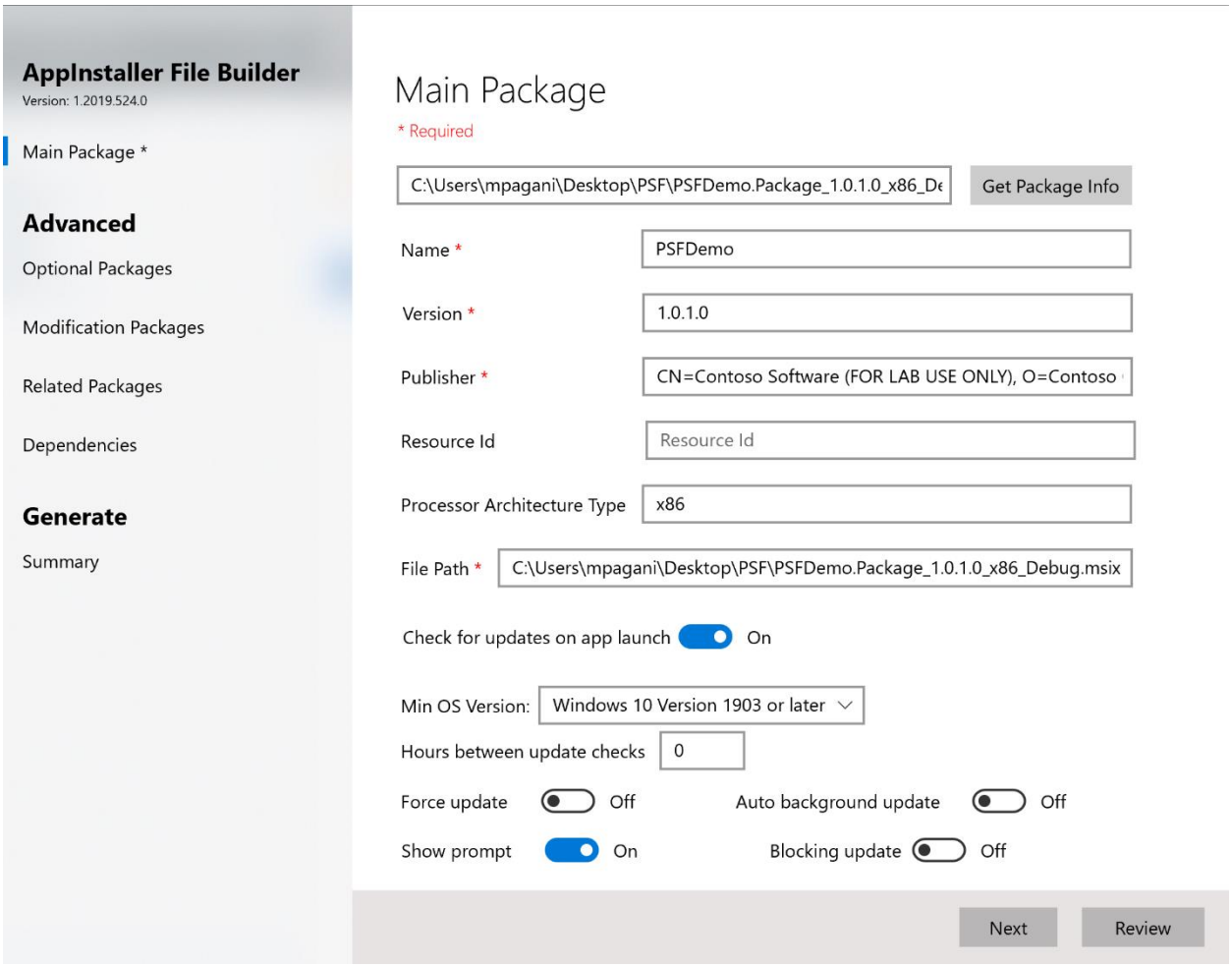


Figure 94: The AppInstaller File Builder tool

In the **Main Package** section, you can manually fill in all the information about the main application, or just click **Get Package Info** and choose the MSIX package you want to deploy from your hard disk. The tool will automatically retrieve all the relevant information from the manifest and fill the various fields.

In the second area, you can turn on the automatic check for updates feature. Based on the **Min OS Version** value you select from the drop-down list, you will see different options, since not all the features are supported by all the Windows 10 versions.

The **Advanced** section of the tool can be used to configure the various additional packages that are supported to be deployed together with the main one: optional packages, modification packages, related packages, and dependencies. All the various sections work in the same way: you can manually fill the various fields, or you can select the MSIX package and the tool will extract all the relevant information from the manifest.

Once you have completed the configuration, you can click **Review** to reach the final stage, where the tool will summarize all the options you have selected. Just click **Generate** to choose a folder on your computer where the App Installer file will be generated.

Publishing the package on a website

In some cases, you might experience some issues when you publish an MSIX package with the related App Installer file on a website. The reason is that not all the web servers are configured to recognize file extensions like .msix or .appinstaller. If that's the case, you will have to configure the server to support the following MIME types.

Code Listing 61

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <!--This is to allow the web server to serve resources with the
appropriate file extension.-->
    <staticContent>
      <mimeMap fileExtension=".appx" mimeType="application/appx" />
      <mimeMap fileExtension=".msix" mimeType="application/msix" />
      <mimeMap fileExtension=".appxbundle"
mimeType="application/appxbundle" />
      <mimeMap fileExtension=".msixbundle"
mimeType="application/msixbundle" />
      <mimeMap fileExtension=".appinstaller"
mimeType="application/appinstaller" />
    </staticContent>
  </system.webServer>
</configuration>
```

The previous snippet shows how to configure an ASP.NET web server to support the required MIME types.

Adding a custom update experience

In some scenarios, you may want to add a custom update experience. For example, you may want to display a notification to users if an update becomes available while they are using the application, so that they are aware that they must restart it to get it.

The Universal Windows Platform comes with a set of APIs that you can use to query the server, based on the App Installer configuration, and check if any update is available. Since the APIs are part of the UWP ecosystem, you first need to enhance your application by following the guidance we saw in Chapter 6.

The API is part of the **`Windows.ApplicationModel`** namespace, and it's simple to use.

Code Listing 62

```
private async Task CheckForUpdates()
{
```

```
    var result = await Package.Current.CheckUpdateAvailabilityAsync();
    if (result.Availability == PackageUpdateAvailability.Available)
    {
        MessageBox.Show("There's a new update! Restart your app to
install it");
    }
}
```

You can retrieve a reference to the current package using the **`Package.Current`** singleton, and then call the asynchronous method **`CheckUpdateAvailabilityAsync()`**. You will get back, inside the **`Availability`** property, a value from the **`PackageUpdateAvailability`** enumerator, which you can use to detect if an update is available. It's up to you to implement the logic that works best for your application. For example, you can display a message or send a toast notification. In the previous code snippet, we use the standard **`MessageBox`** from the .NET Framework to display a pop-up to the user to notify him that there's an update available.

Choosing the right certificate

When it comes to distributing MSIX packages, there's one important decision that you must make: the certificate you're going to use to sign the package. As mentioned in Chapter 1, MSIX enforces a strong security model. All the packages must be signed with a digital certificate, which must be trusted by the machine.

As such, it's very important to choose the right certificate to ensure a smooth deployment experience.

Publishing on the Microsoft Store

The simplest approach is to publish your application on the Microsoft Store or the Store for Business/Education. You won't have to sign your package; the Microsoft Partner Center will do it for you during the certification process. The package will be signed with a Microsoft certificate, which is trusted by all Windows 10 devices.

Sideloading

When you sideload your application, you need to handle the package signing yourself. There are a few different options:

- **Use a certificate provided by an internal certification authority (CA)**. This is the most common scenario when it comes to internal line-of-business applications. Many major enterprises have an internal CA, which can generate certificates that are automatically trusted by every machine in the company. When a package is signed with this certificate, all the company assets will be able to install the application without having to manually trust the certificate.

- **Use a certificate provided by a public certification authority**. This scenario is typical when you want to distribute an application to a public audience (either consumers or enterprises) without using the Microsoft Store. Since the certificate is coming from a public CA, the chain will be already trusted by Windows, and every user will be able to install it without having to manually trust the certificate. These certificates are purchased from an organization, and during the process, the CA will validate your identity.
- **Use a self-signed certificate.** This option is good when you're in testing phase or it's a very limited distribution, but it isn't considered a best practice when it comes to enterprise or public distribution. A self-signing certificate validates the integrity of the package, but not the identity. Since there isn't any identity verification process (like when you purchase a certificate from a public CA), you can easily create a self-signing certificate using a name for which you don't have any rights. Additionally, before using it, the self-signing certificate must be manually added to the **Trusted People** certificate store in Windows, which requires administrator rights.

Creating a self-signed certificate

To create a certificate, you can use a set of PowerShell commands. The first step is to open a PowerShell prompt with administrative rights, which are required to add new items to the Windows certificate store. The easiest way to do it is to right-click the Start menu in Windows 10 and choose **Windows PowerShell (Admin)**.

First, you will have to type the following command.

Code Listing 63

```
New-SelfSignedCertificate -Type Custom -Subject "CN=MyCompany" -KeyUsage
DigitalSignature -FriendlyName MyApplication -CertStoreLocation
"Cert:\LocalMachine\My"
```

The two parameters that you must customize are highlighted in bold:

- **`Subject`** must match the publisher name declared in the manifest of the application.
- **`FriendlyName`** is a free value that you must use as a unique identifier for your certificate.

This command will create the certificate inside your local certificate store. As output, you will see a recap of the operation including the thumbprint of the certificate, as highlighted in Figure 95. Take note of it, because we're going to need it in the next step.

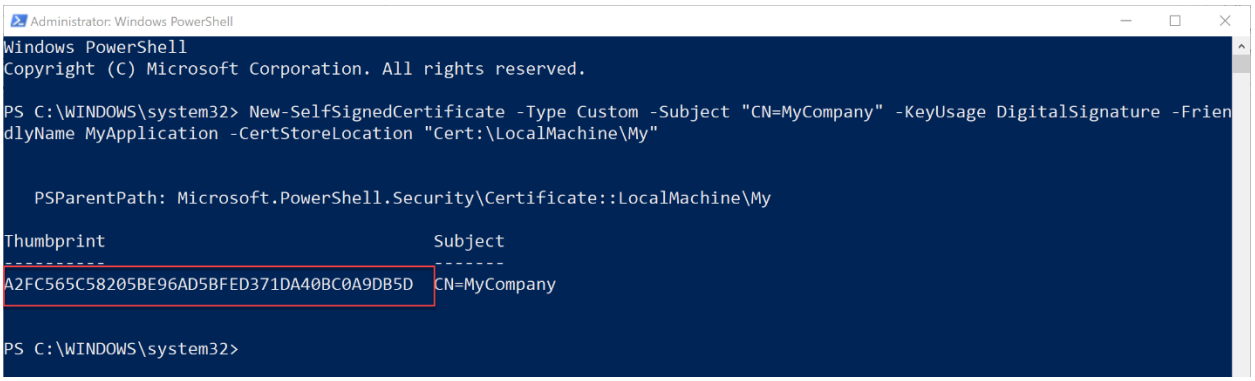


Figure 95: The output of the PowerShell command to create a certificate

Now that we have a certificate, we need to export it as a file so that we can choose it during the setup of the packaging process in the MSIX Packaging Tool.

Here is the PowerShell command to execute.

Code Listing 64

```
$pwd = ConvertTo-SecureString -String <Your Password> -Force -AsPlainText
Export-PfxCertificate -cert "Cert:\LocalMachine\My\<Certificate
Thumbprint>" -FilePath <FilePath>.pfx -Password $pwd
```

The first line of the command will define a variable called **$pwd**, which signifies where to store the password of the certificate that must be passed as the value of the **-String** parameter.

The second line will perform the export operation and requires customizing two parameters:

- In the **cert** parameter, you must replace the **<Certificate Thumbprint>** placeholder with the value you noted in the previous step, right after creating the certificate.
- In the **FilePath** parameter, you must specify the full path where you want to save the PFX file.

At the end of the process, you'll obtain a password-protected PFX file, which can be used in combination with the MSIX Packaging Tool to sign the package.

Signing the package

A package can be signed by using the **signtool** utility included in the Windows 10 SDK. You can get to it by opening a command prompt and pointing it to the folder **C:\Program Files (x86)\Windows Kits\10\bin\10.0.xyz.0\x86**, where **xyz** is the most recent version of the Windows 10 SDK you have installed. We learned to use this utility in Chapter 3, but let's review how to use it.

This is the command to execute.

Code Listing 65

```
signtool sign /tr http://timestamp.digicert.com /td sha256 /fd sha256
/f "c:\path\to\mycert.pfx" /p pfxpassword "c:\path\to\package.msix"
```

The relevant parameters to customize are **/f**, which must include the path of the PFX file with the certificate, and **/p**, which must include the certificate's password. The last parameter is the full path of the MSIX package you want to sign. Remember that the **Publisher** declared in the manifest of the package must match the subject of the certificate; otherwise, the signing operation will fail.

If you're using Visual Studio, you also have the ability to sign the package with a UI included in the manifest editor. Double-click the **Package.appxmanifest** of your UWP or Windows Application Packaging Project and move to the **Packaging** section. Near the **Publisher** field, you will find the **Choose Certificate** option.

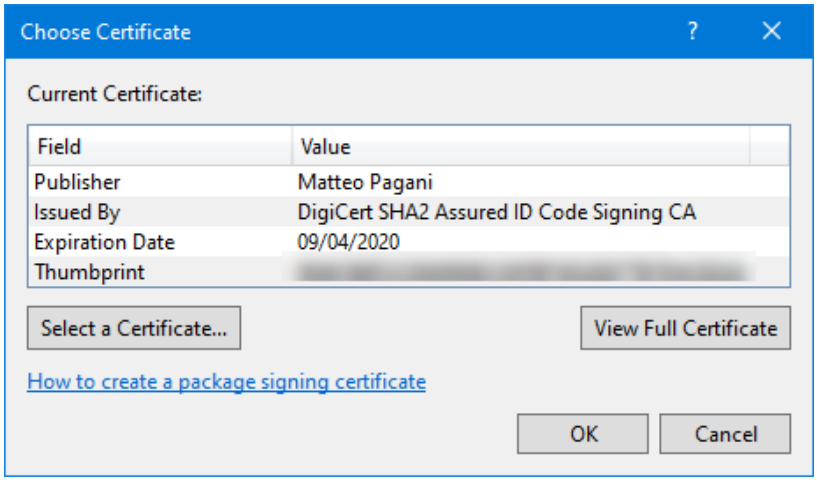


Figure 96: The Visual Studio option to choose a certificate

When you click **Choose Certificate**, you will have the opportunity to select one of the certificates you have in your personal storage.

Supporting Windows 10 in S Mode

Among the various Windows versions, Microsoft offers a special edition called **S Mode**, which is tailored for education and business scenarios. Windows 10 in S Mode offers enhanced security and performance, since you can install only applications coming from the Microsoft Store. This severely reduces the surface attacks from malicious developers, since all the Store applications are vetted from a certification process. Additionally, many of the vectors commonly used to perform malicious operations (like leveraging PowerShell or the Command Prompt) are blocked by default.

When it comes to Win32 applications packaged with MSIX, however, there are a couple of additional requirements to keep in mind.

The first one is that you aren't allowed to launch any executable that lives outside the package or that is blocked by default in S Mode, like cmd.exe, powershell.exe, or reg.exe. You can find the full list of blocked executables in the official documentation. If the executable is included inside the package, you can safely launch it using `Process.Start()` or any other equivalent to invoke another process.

The second requirement is that the application can't load code from a location outside the folder where the package is deployed. An application is allowed to generate code at runtime, but only if it's kept in memory. If, for example, it's generated and stored in a temporary folder, the operation will be blocked.

When it comes time to test if your packaged application is fully compliant with Windows 10 in S Mode, you can hit a blocker. Even if you own a device running Windows 10 in S Mode, you won't be able to sideload an application, since you can install only packages coming from the Store. Microsoft has made available a set of policies for Device Guard, a technology included in Windows 10 to lock down the operating system, which can be used to simulate S Mode.

When you enable these policies, the two extra requirements we just mentioned will be enforced. The first step is to download the policies, which are available in the documentation. Inside the compressed file, you will find three different files with the .p7b extension, which correspond to three different policies:

- **SiPolicy_Audit.p7b**: When this policy is enabled, the application will continue to behave like it's on a regular Windows 10 machine, but all the code integrity violations will be logged in Event Viewer.
- **SiPolicy_Enforced.p7b**: When this policy is enabled, Windows will behave like the real S Mode. You will be able to get applications only from the Store, and all the code integrity violations will be enforced.
- **SiPolicy_DevModeEx_Enforced.p7b**: When this policy is enabled, Windows will behave like the real S Mode, but you will be able to sideload applications that are signed with a special certificate, which is included inside the file you have downloaded.

Once you have selected the policy you want to enable, you must copy the **.p7b** file in the **C:\Windows\System32\CodeIntegrity** folder and rename it **SiPolicy.p7b**. Then reboot your machine. At restart, the policies will be applied.

Tip: It's best to use a virtual machine to simulate Windows 10 in S Mode, since all the applications and services you have installed on a regular production machine may stop working, causing random issues.

All the errors will be logged in Event Viewer. To view them, navigate to **Application and Services Logs** > **Microsoft** > **Windows** > **CodeIntegrity** and look for the **Operational** section.

The following snippets show some examples of real errors you might see when you launch an application on Windows 10 in S Mode.

Code Listing 66

```
Code Integrity determined that a process (\Device\HarddiskVolume2\Program
Files\WindowsApps\2bd04f34-968e-47a1-b7bc-
```

```
3f667a2f1498_1.0.1.0_x86__8x64rcfmpem5g\LaunchCmd\LaunchCmd.exe) attempted
to load \Device\HarddiskVolume2\Windows\SysWOW64\cmd.exe that did not meet
the Enterprise signing level requirements or violated code integrity policy
(Policy ID:{a244370e-44c9-4c06-b551-f6016e563076}).
```

This error is logged when the first extra requirement isn't satisfied. The application is trying to launch one of the executables that is blocked on Windows 10 in S Mode (**cmd.exe**), and a code integrity violation is logged in Event Viewer.

Code Listing 67

```
Code Integrity determined that a process (\Device\HarddiskVolume2\Program
Files\WindowsApps\Contoso.MyEmployees_4.0.0.0_x64__b4ranh25ygct6\app\MyEmpl
oyees.exe) attempted to load
\Device\HarddiskVolume2\Users\qmatteoq\AppData\Local\Temp\29684c81-1b32-
4447-8ab6-fed88bd6202d.tmp.node that did not meet the Enterprise signing
level requirements or violated code integrity policy (Policy ID:{a244370e-
44c9-4c06-b551-f6016e563076}).
```

This is an example of the second requirement enforced by Windows 10 in S Mode. The application has generated a temporary file, which contains some code that it's trying to load. However, since this file is stored in a temporary folder (**AppData\Local\Temp**) and not in the package folder, the operation is blocked.

Updating the package for Windows 10 in S Mode testing

The policy file called **SiPolicy_DevModeEx_Enforced.p7b** is one of the most effective when it comes to testing your application in S Mode. Thanks to this policy, you can get helpful logging in Event Viewer; at the same time, you can see if your application is indeed failing as a consequence of one of the additional requirements.

However, sideloading an application on a machine configured with such a policy can be challenging. As we have learned, the subject of the certificate you use to sign a package must match with the publisher name declared in the app manifest. Since, in order to do the sideloading, you need to sign the package with the special certificate provided by Microsoft, you also need to temporarily change the publisher in the manifest to match its subject, which is **CN=Appx Test Root Agency Ex**.

As such, you would need to:

1. Unpack the package with the **makeappx** tool included in the Windows 10 SDK.
2. Edit the **AppxManifest.xml** file to change the **`Publisher`** attribute of the **`Identity`** tag.
3. Use the **makeappx** tool to recreate the package.
4. Use the **signtool** utility in the Windows 10 SDK to sign the package with the testing certificate provided by Microsoft.

To make your life easier, you can use a PowerShell script that is part of the MSIX Toolkit called ModifyPackagePublisher. Once you have downloaded the latest release from GitHub, you need to extract it in a folder of your choice. Then, inside this folder, you will have to copy:

- Two tools from the Windows 10 SDK: **signtool.exe** and **PackageEditor.exe**. You can get them from the **C:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86** folder.
- The certificates that are supported by the special policies to simulate Windows 10 in S Mode. The files are called **AppxTestRootAgency.cer** and **AppxTestRootAgency.pfx**, and they are included in the **AppxTestRootAgency** folder inside the compressed policies file you have downloaded from the official documentation.

In the end, you can run the following command.

Code Listing 68

```
.\modify-package-publisher.ps1 -directory "C:\path\folderWithThePackage" -
certPath ".\AppxTestRootAgency.cer" -pfxPath ".\AppxTestRootAgency.pfx"
```

The three relevant parameters are:

- **`-directory`**, which must point to the folder that includes the MSIX package you want to test. The tool supports multiple packages, so you're free to include in the folder all the packages you want to test.
- **`-certPath`**, which must point to the **AppxTestRootAgency.cer** file included in the policies.
- **`-pfxPath`**, which must point to the **AppxTestRootAgency.pfx** file included in the policies.

If the certificate has a password, you can use the **`-password`** parameter to set it.

At the end of the process, you will have an MSIX package that you can install on your Windows 10 machine with the **SiPolicy_DevModeEx_Enforced.p7b** policy applied.

Using PowerShell to handle MSIX packages

MSIX packages are very flexible and can be deployed in multiple ways: through the Microsoft Store, using a website or a file share, and using Intune or SSCM. One of the most powerful options is to use PowerShell to control the lifecycle of MSIX packages. Let's see the most common commands.

Deploy a package

Deploying a package is as easy as passing to the **`Add-AppPackage`** command the full path of the MSIX file you want to deploy using the **`Path`** parameter.

Code Listing 69

```
Add-AppPackage -Path SamplePackage.msix
```

Deploy a package to all the users on a machine

MSIX packages are deployed per user. However, in some scenarios you may want to deploy an application to the current and future users who will log in on the same machine. This goal can be performed using the **Add-AppProvisionedPackage** command.

Note: This command must be executed from a PowerShell prompt with administrative rights.

There are two different options to use this command. The first scenario is deploying the package to all the users of the current Windows 10 machine.

Code Listing 70

```
Add-AppxProvisionedPackage -Online -PackagePath "SamplePackage.msix"
```

The second scenario is when you want to apply the package to a Windows 10 image so that you can provision machines with the application already installed for all the users. In this case, you need to replace the **Online** parameter with the **Offline** one and specify the path that contains the image.

Code Listing 71

```
Add-AppxProvisionedPackage -Path "C:\Windows10Image" -PackagePath
"SamplePackage.msix"
```

Get all the information about a package

In order to get all the information about a deployed package, you can use the **Get-AppPackage** command. If you issue it without passing any parameter, you will get a list of all the packages installed on your machine, ordered from the oldest to the most recent one.

If, instead, you're looking for information about a specific package, you can pass as parameter a search query, which also supports wildcards. For example, if the application is called **MyEmployees**, you can find it using the following command.

Code Listing 72

```
Get-AppPackage *MyEmployees*
```

Regardless of the method you choose, the output will be similar to the following one.

```
Name              : MyEmployees
Publisher         : CN=Contoso Software (FOR LAB USE ONLY), O=Contoso Corporation, C=US
Architecture      : X64
ResourceId        :
Version           : 1.0.0.0
PackageFullName   : MyEmployees_1.0.0.0_x64__8h66172c634n0
InstallLocation   : C:\Program Files\WindowsApps\MyEmployees_1.0.0.0_x64__8h66172c634n0
IsFramework       : False
PackageFamilyName : MyEmployees_8h66172c634n0
PublisherId       : 8h66172c634n0
IsResourcePackage : False
IsBundle          : False
IsDevelopmentMode : False
NonRemovable      : False
IsPartiallyStaged : False
SignatureKind     : Developer
Status            : Ok
```

Figure 97: The information displayed by the Get-AppPackage command about a deployed application

As you can see, the output lists all the main information about the package, like the name, the publisher, the **`PackageFamilyName`**, the version number, etc.

If you are in a multi-user scenario and you execute the **`Get-AppPackage`** command in a PowerShell prompt with administrative rights, you will also see the information about which users have already installed the package in the **`PackageUserInformation`** field.

```
Name                   : MyEmployees
Publisher              : CN=Contoso Software (FOR LAB USE ONLY), O=Contoso Corporation, C=US
Architecture           : X64
ResourceId             :
Version                : 1.0.0.0
PackageFullName        : MyEmployees_1.0.0.0_x64__8h66172c634n0
InstallLocation        : C:\Program Files\WindowsApps\MyEmployees_1.0.0.0_x64__8h66172c634n0
IsFramework            : False
PackageFamilyName      : MyEmployees_8h66172c634n0
PublisherId            : 8h66172c634n0
PackageUserInformation : {S-1-5-21-2586070391-1003120513-3189238791-1000 [msix]: Installed,
                         S-1-5-21-2586070391-1003120513-3189238791-1002 [MsixLab]: Installed}
IsResourcePackage      : False
IsBundle               : False
IsDevelopmentMode      : False
NonRemovable           : False
IsPartiallyStaged      : False
SignatureKind          : Developer
Status                 : Ok
```

Figure 98: A package that has been deployed to multiple users on the same machine

Modification packages won't appear as regular packages, so you won't find them using the **`Get-AppPackage`** command. However, if an application includes one or more modification packages, they will be displayed in the **Dependencies** section, as shown in the following image.

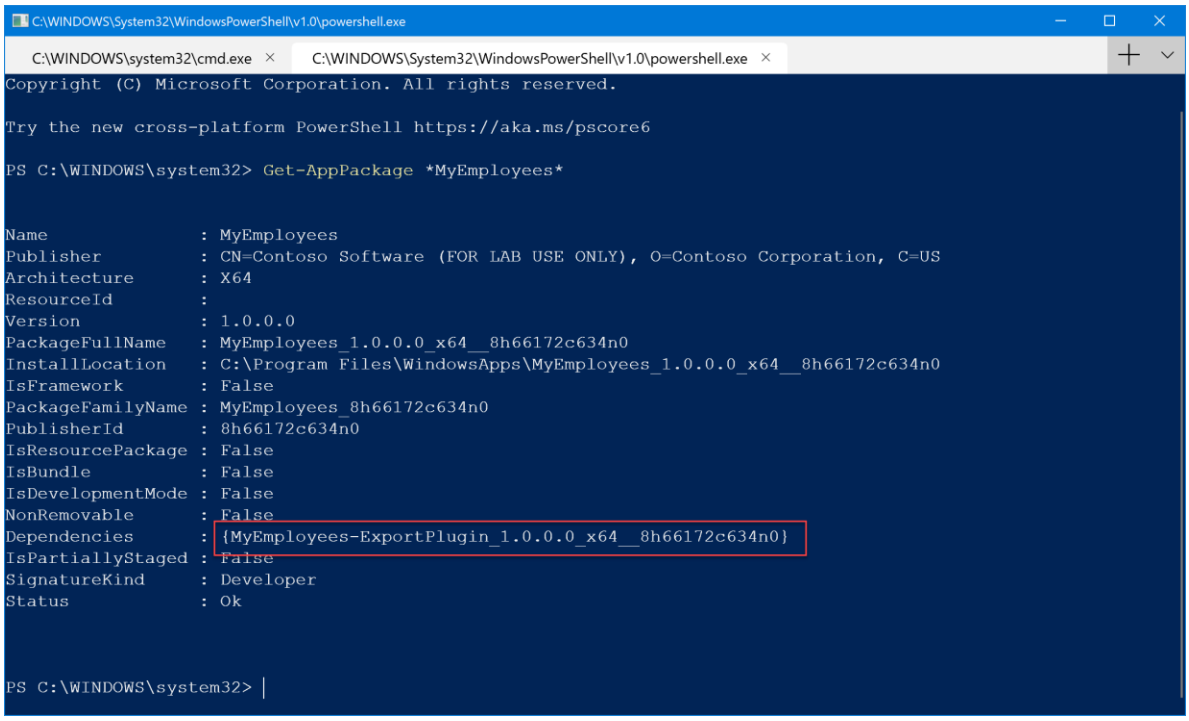


Figure 99: An application deployed together with a modification package

Remove a package

To remove a package, you can use the **`Remove-AppPackage`** command, which requires the **`Package`** parameter with the Package Full Name of the application you want to remove. You can obtain this information using the **`Get-AppPackage`** command we have just learned to use. For example, to remove the **MyEmployees** application, you can use the following command.

Code Listing 73

```
Remove-AppPackage -Package MyEmployees_1.0.0.0_x64__8h66172c634n0
```

Remove a package for all the users

If a package has been deployed to all the current and future users on the machine, you can use the **`Remove-AppProvisionedPackage`** command to remove all the installed instances. You must specify the **`Package`** parameter, followed by the Package Full Name of the application you want to remove. If you want to remove the application from the current machine, you must also add the **`AllUsers`** parameter.

Code Listing 74

```
Remove-AppPackage -Package MyEmployees_1.0.0.0_x64__8h66172c634n0 -
AllUsers
```

If you want to remove the package from a Windows 10 image you have built, you can use the **Path** parameter to specify the path that contains it.

Code Listing 75

```
Remove-AppPackage -Package MyEmployees_1.0.0.0_x64__8h66172c634n0 -Path
C:\Windows10Image
```

Note: This command must be executed from a PowerShell prompt with administrative rights.

Execute a process inside the container

When you're troubleshooting a packaged application, some operations might be complicated by the container, which virtualizes the file system and the registry. Let's say, for example, that you want to check whether the application has correctly created a key in the registry. If you open the Registry Editor, you won't be able to see it, because the key is created inside the virtual registry. Another example is the deployment of a dependency or of a modification package. If you want to check whether a file is really there, you can't use a normal command prompt or File Explorer, because the system folders are handled with the Virtual File System.

To overcome this limitation, PowerShell offers a command that you can use to invoke a process inside the same container of an application. This means the process will have access to the virtual registry and the virtual file system, exactly like the main application. Thanks to this approach, you can launch a command prompt or the Registry Editor inside the container and explore the full file system or the registry.

The command is called **Invoke-CommandInDesktopPackage**, and it requires three parameters:

- **AppId**, the application identifier declared in the manifest of the application.
- **PackageFamilyName,** the Package Family Name of the application.
- **Command**, the process you want to launch inside the container.

Note: To use this command, the Developer Mode must be enabled in your Windows 10 PC. To enable it, you must open your Settings, go to Update & Security > For developers, and choose Developer mode.

For example, this is the command you must launch if you want to run the Registry Editor inside the same container of the **MyEmployees** application.

Code Listing 76

```
Invoke-CommandInDesktopPackage -AppId MyEmployees -PackageFamilyName
MyEmployees_8h66172c634n0 -Command regedit
```

The Registry Editor will be opened with a merged view of the system registry with the virtual registry. The screenshot in Figure 100 shows a set of keys that belong to the MyEmployees application and are deployed by the MSIX package. As such, they wouldn't be visible if you just launch the Registry Editor outside the container.

The following example shows how to run a command prompt inside the application's container.

Code Listing 77

```
Invoke-CommandInDesktopPackage -AppId MyEmployees -PackageFamilyName
MyEmployees_8h66172c634n0 -Command cmd
```

Thanks to this command prompt, you'll be able to navigate the virtual file system as if they're actual files and folders on your hard drive. In Figure 16, you can see the list of files inside the **C:\Program Files (x86)\Contoso\MyEmployees** folder. This folder doesn't really exist in the system, but it's included in the **VFS\ProgramFilesX86** folder of the package.

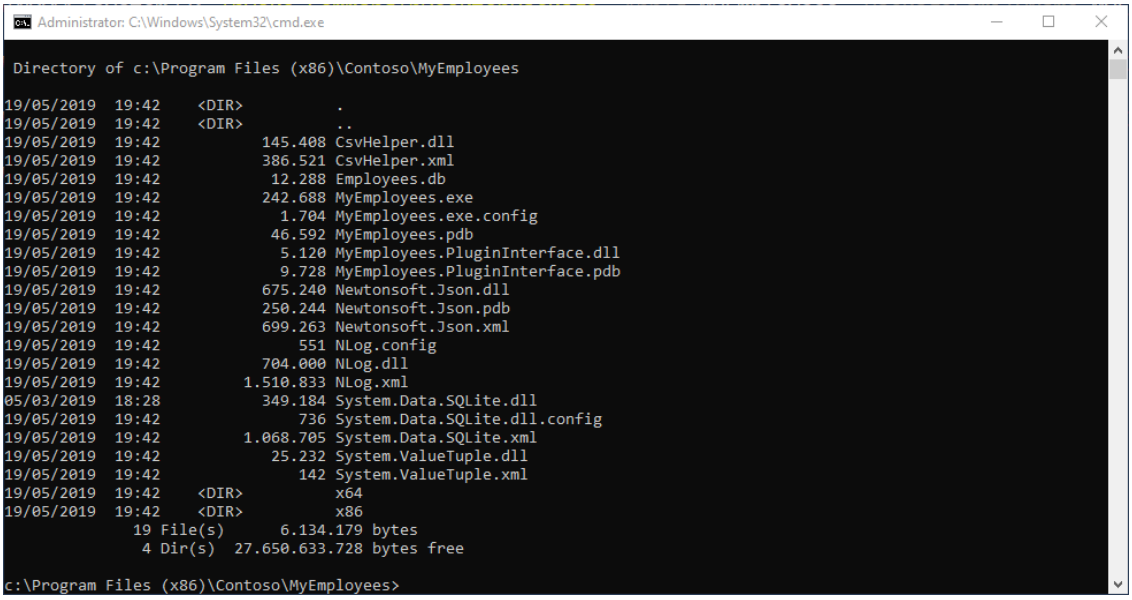


Figure 100: A command prompt opened inside the container shows files that belong to the virtual file system as if they're part of the physical hard drive

An easier way for troubleshooting

The **`Invoke-CommandInDesktopPackage`** command is very useful, but its use isn't very straightforward. You need to retrieve the application identifier and the Package Family Name of the application before using it, which might be challenging.

Caphyon, the company behind the popular installer authoring tool Advanced Installer, has released a free tool called Hover, which makes it easier to launch other applications inside a container.

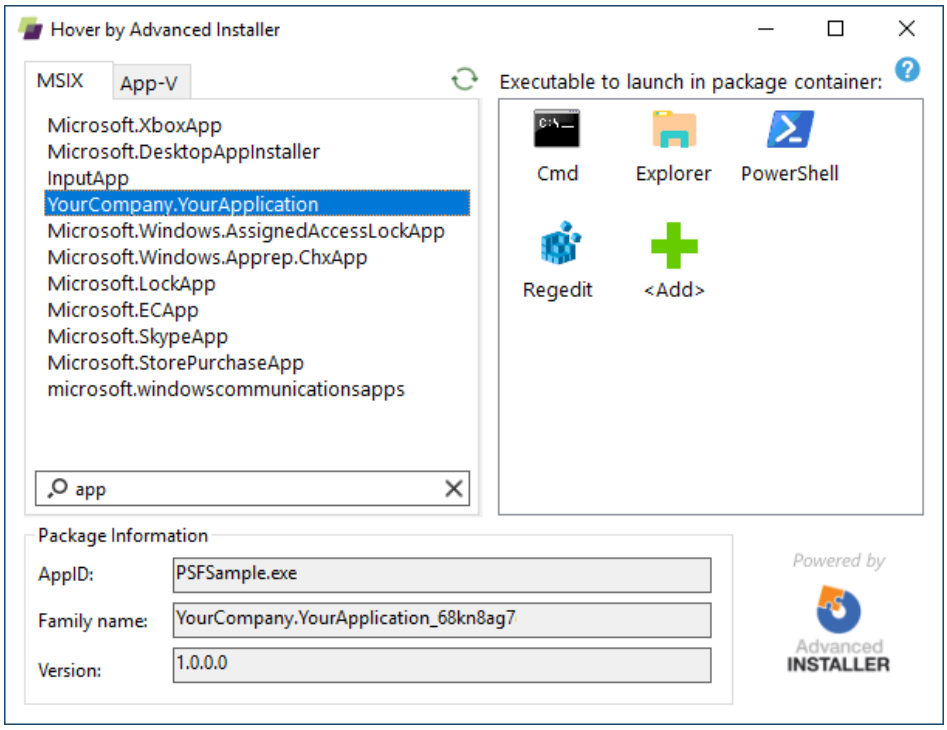


Figure 101: Hover, a tool that makes it easier to launch applications inside an existing container

The user interface is simple to decode. Under the **MSIX** tab, you will find a list of all the MSIX-packaged applications installed on your machine. On the right, you can choose the executable you want to launch. The tool already includes the most popular ones, like command prompt, File Explorer, and Regedit, but you can also add your own executable.

You just have to double-click the executable in the right panel to launch it inside the container, so that it will be able to leverage the registry keys and files that are included in the virtualized ecosystem.

Chapter 8 DevOps for Windows Desktop Applications with MSIX

Adopting a DevOps culture is becoming more and more critical to delivering successful software projects. Thanks to DevOps, you can address feedback from your customers quickly, you can identify problems as soon as they happen, and you can react to changes in a more efficient way.

As Donovan Brown, one of Microsoft's DevOps leads, says, DevOps is all about "the union of people, process, and products to enable continuous delivery of value to your end users." It doesn't matter which tools and approaches you decide to adopt; the starting point should always be identifying the value that you can bring to your users.

For a Windows desktop application, the added value of adopting DevOps is making sure that your users are always up to date, which is still a challenge in the desktop world. From this point of view, a web application is easier to deploy. Your users will always have access to it from a browser and from a single entry point, which is the URL of the web application. As such, they will always use the most recent version of your product, since you just need to deploy it on a single server.

Note: In the real world, web applications can have a complex architecture, made by microservices, multiple servers, containers, etc. However, this doesn't change the fact that, in the end, your customers will always access it using a unique entry point.

Windows applications, instead, can run on hundreds, thousands, millions of different computers all around the world, so deploying updates isn't as straightforward as with a web application. MSIX helps us solve this challenge. As we saw in Chapter 7, deployment solutions like the Microsoft Store and the App Installer make it easier for users to always stay up to date. We just need to make the process agile, so that releasing updates can become an easy and automatic procedure.

Welcome Azure DevOps

Azure DevOps is the platform built by Microsoft to provide all the services you need to create a DevOps experience for your project. The platform is made up of five different services:

- **Azure Boards**: Used for project management, it allows you to track the status of the project, define work items, track bugs, etc.
- **Azure Pipeline**: Used for building a continuous integration and continuous delivery experience in your project. It's the platform we're going to talk about more in this section.
- **Azure Repos**: This provides a space to host the source code of your project, using the Git platform.
- **Azure Test Plans**: Used to run advanced automatic tests on your projects.
- **Azure Artifacts**: Makes it easier for development teams to build and share packages and libraries.

One of the biggest advantages of Azure DevOps is its flexibility. You can choose only the services you need, and you can easily integrate them with existing third-party services in the DevOps space. For example, you can use Azure Pipeline to build your projects, but host your code on GitHub. Or, you can host your code on Azure Repos, but use Jenkins to enable CI/CD.

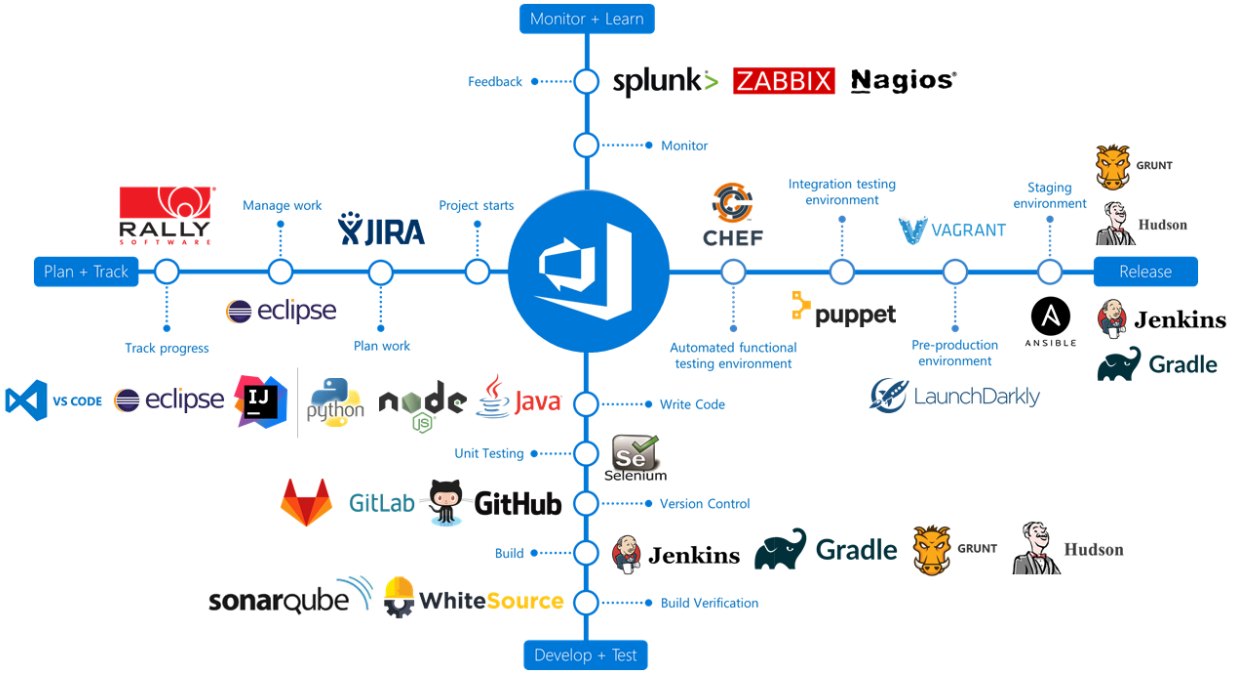


Figure 102: Azure DevOps supports integration with many third-party services

If you don't already have an Azure DevOps account, you can create one on the official website. There are multiple plans, including a free one, which include as main features:

- Up to five users (if a user has a Visual Studio subscription, the user doesn't count against the total).
- Unlimited repositories.
- One hosted job with 1,800 minutes of CI/CD per month.

If you're working on an open-source project, you can get an unlimited number of users and unlimited CI/CD minutes for free, with up to 10 parallel jobs.

Azure DevOps is a great fit for deploying Windows desktop applications, since:

- You can build your projects with a Windows-based hosted agent, which already comes with Visual Studio, the Windows 10 SDK, and the .NET Framework.
- You have tasks that will help you sign your package with the right certificate, in case of sideloading.
- You have tasks that can deploy a MSIX package directly on the Microsoft Store or to a website.

Let's see it in more detail.

Using Azure Pipeline

Azure Pipeline is a service you can use to build a CI/CD experience in your projects. What does that mean, exactly?

- **Continuous Integration** is the process of triggering a new build of a software project every time new code is committed to the repository. As part of this project, you might also trigger the execution of unit tests, to make sure that the code you have added doesn't break any existing feature of the application. Every successful build will generate a new artifact, which can be deployed to your users. In the case of a Windows desktop application, this artifact will be an MSIX package. This task is implemented using **build pipelines**.
- **Continuous Deployment** is the process of automatically deploying to your users the artifacts that are produced by a build pipeline. The deployment can happen in multiple stages (a dev environment, a testing environment, a production environment, etc.), and there may be gates between one stage and another. For example, you may require approval from the administrator of the project before triggering a deployment to the production environment. In the case of a Windows desktop application, we're going to enable CD to deploy the application on the Store or on a website. This task is implemented using **release pipelines**.

The biggest advantage of implementing a CI/CD pipeline is that you only need to configure it once. After it's up and running, new versions of the application will be automatically created and deployed without any manual intervention from your side.

A pipeline is nothing more than a series of tasks that are executed one after the other. These tasks can perform various operations, like compiling the source code, running unit tests, and installing a dependency. Azure Pipeline supports a wide range of tasks, either built into the platform, or distributed by third-party developers using the Marketplace.

Tasks are executed on an agent, which is a sort of virtual machine that contains the build environment. Azure DevOps offers a wide range of hosted agents, which are prebuilt machines with all the tools required to perform a specific set of tasks. These agents offer different operating systems (Windows, Linux, MacOS) and different build environments (Visual Studio 2017, Visual Studio 2019, etc.). If you need to leverage special tools that can't be easily installed on a hosted agent, you have the opportunity to create your own custom agent and upload it on Azure DevOps.

Hosting the project

Before building a pipeline, you will have to host your code on a repository. Azure DevOps offers its own platform, Azure Repos, but you're free to use any platform of your choice, like GitHub or GitLab. If you're packaging a Win32 application, make sure the repository also includes the Windows Application Packaging Project, which is required to create a MSIX package.

Creating a build pipeline

Pipelines in Azure DevOps are defined using YAML, which is a markup language. This solution enables the **infrastructure as code** philosophy, which allows you to manage and provision resources using a code file rather than physical hardware configuration or interactive configuration tools. This way, the pipeline becomes part of your source code and can evolve with it.

Build pipelines can be created under the **Pipelines > Builds** section of your project on Azure DevOps. As a first step, you will be asked which platform is hosting the source code of your project. You can choose from among many options, like Azure Repos, GitHub, and BitBucket. In the second step, Azure Pipeline will propose a set of templates for different project types. Each of them will create a basic YAML file with some tasks already configured. In our scenario, the template to choose is **Universal Windows Platform**, which will compile our UWP or Windows Application Packaging Project and create an MSIX.

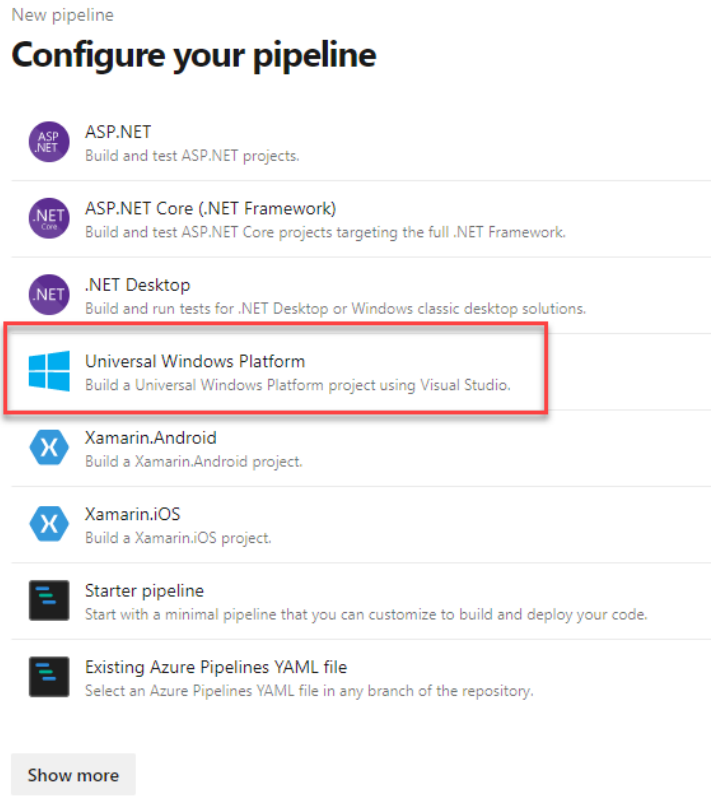


Figure 103: The template to choose to build an MSIX package in Azure Pipeline

The template will create the following YAML file.

Code Listing 78

```
# Universal Windows Platform

# Build a Universal Windows Platform project using Visual Studio.
```

```yaml
# Add steps that test and distribute an app, save build artifacts, and
more:
# https://aka.ms/yaml

trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'x86|x64|ARM'
  buildConfiguration: 'Release'
  appxPackageDir: '$(build.artifactStagingDirectory)\AppxPackages\\'

steps:
- task: NuGetToolInstaller@0

- task: NuGetCommand@2
  inputs:
    restoreSolution: '$(solution)'

- task: VSBuild@1
  inputs:
    platform: 'x86'
    solution: '$(solution)'
    configuration: '$(buildConfiguration)'
```

```
    msbuildArgs: '/p:AppxBundlePlatforms="$(buildPlatform)"
/p:AppxPackageDir="$(appxPackageDir)" /p:AppxBundle=Always
/p:UapAppxPackageBuildMode=StoreUpload'
```

Let's analyze the meaning of the various elements.

The configuration

The first three items, **`trigger`**, **`pool`**, and **`variables`**, are used to configure the build environment.

The **`trigger`** section is used to enable continuous integration and specifies the criteria used to trigger a new build. By default, it contains the name of the branch we're building (**`master`**), which means that every commit to this branch will trigger a new build.

The **`pool`** section contains the configuration of the agent that will execute the build. With **`windows-latest`** we specify that we want to use the latest Windows image.

The **`variables`** section defines a set of parameters that are leveraged during the build process. Specifically:

- **`solution`** defines which solution we want to build. By default, the pipeline will build all the solutions included in the project.
- **`buildPlatform`** defines which architecture we want to support inside the package.
- **`buildConfiguration`** is the Visual Studio configuration used for the build.
- **`appxPackageDir`** is the folder where the package will be created if the build is successful.

The steps

The **`steps`** section contains the tasks that will be performed one after the other. Each task has a unique identifier and a set of properties to customize it. The first two tasks, **`NuGetToolInstaller`** and **`NuGetCommand`**, will download the most recent version of NuGet and restore all the dependencies in the project. The last task, called **`VSBuild`**, will build the project and create an MSIX package.

Customizing the build

Before saving the YAML file, we need to make a few changes to fit our MSIX packaging scenario.

The first one is to make sure we're building for the right CPU architecture. By default, the **buildPlatform** entry under **variables** will include x86, x64, and ARM, since these are the architectures supported by Universal Windows Platform applications. However, if you're building a Win32 application packaged with the Windows Application Packaging Project, ARM won't be supported, so make sure to remove it and leave only x86 or x64 (or both, based on your application's configuration).

The second one is to disable the package signing. By default, MSIX packages are signed with a self-signing certificate generated by Visual Studio during the build process. However, signing the package during the build process isn't a good practice because we would need to upload the certificate to the repository. This means every developer working on the project will have access to the certificate, increasing the risk of identity theft. As such, the recommended approach is to sign the package in the release pipeline and store the certificate in a safe way. We're going to see how to do this later in the chapter. For the moment, just disable the signing during the compilation by adding the **/p:AppxPackageSigningEnabled=false** parameter to the **msBuildArgs** property of the **VSBuild** task.

The last change is to upload the artifacts. You can think of the hosted agent as a sort of virtual machine. Every time a new build is triggered, a new instance is created, which takes care of executing all the tasks, one after the other, and then it's disposed of at the end. The consequence is that, if we don't store the output of the build somewhere, it will be lost as soon as the hosted agent is disposed of. Azure DevOps offers its own cloud storage for storing the artifacts. Artifacts are available to the developer for manual download and are important for building a release pipeline. In a CD pipeline, in fact, the deployment is typically kicked off when a new artifact is available as a consequence of a CI pipeline that has been successfully completed. To achieve this goal, you will need to add the following task as the last step.

Code Listing 79

```
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)\AppxPackages'
    ArtifactName: 'drop'
```

This is how the final YAML file should look.

Code Listing 80

```
# Universal Windows Platform
# Build a Universal Windows Platform project using Visual Studio.
# Add steps that test and distribute an app, save build artifacts, and
more:
# https://aka.ms/yaml
```

```
trigger:
- master

pool:
  vmImage: 'windows-latest'

variables:
  solution: '**/*.sln'
  buildPlatform: 'x86|x64'
  buildConfiguration: 'Release'
  appxPackageDir: '$(build.artifactStagingDirectory)\AppxPackages\\'

steps:
- task: NuGetToolInstaller@0

- task: NuGetCommand@2
  inputs:
    restoreSolution: '$(solution)'

- task: VSBuild@1
  inputs:
    platform: 'x86'
    solution: '$(solution)'
    configuration: '$(buildConfiguration)'
    msbuildArgs: '/p:AppxBundlePlatforms="$(buildPlatform)"
/p:AppxPackageDir="$(appxPackageDir)" /p:AppxBundle=Always
/p:UapAppxPackageBuildMode=StoreUpload
/p:AppxPackageSigningEnabled=false'
```

```
- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)\AppxPackages'
    ArtifactName: 'drop'
```

Once you have finished editing the YAML file, you can click **Save and run**. The YAML file will be saved in the root of your repository, and the build will be triggered. You will be able to follow the build step by step, thanks to real-time logging. If the build is successful, you'll be able to access the artifacts using the **Artifacts** button that will appear at the top of the build details page. From there, you will be able to explore and download the files that have been created.

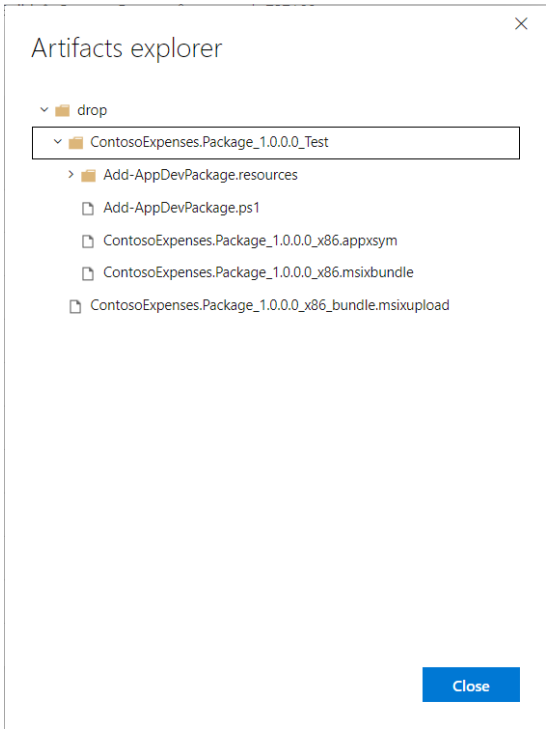


Figure 104: The build output of the pipeline

Setting the package version

If you check the artifact, you will notice that the MSIX package has been generated using the version number that is declared in the manifest of your project. By default, however, the version number will not change for future builds, as the build environment is not persisted between them. It's our duty to manually update the manifest every time we push some code to the repository. However, this approach can lead to many problems. If we forget to update the number, and we generate an update with the same version number as the prevision one, we will break the update chain.

The solution is to leverage the build number generated by Azure DevOps to update the package version also, so that it will be automatically increased at every execution. However, there's a catch. By default, Azure DevOps uses the following expression to generate a build number.

Code Listing 81

```
$(date:yyyyMMdd)$(rev:.r)
```

The dollar sign is used by Azure DevOps to reference variables, which can be configured on the portal. However, some of them are already built into Azure DevOps, like the one used for the date in the expression above. You can find the full list here. The expression in Code Listing 81 will generate a build number like the following.

Code Listing 82

```
20190504.1
```

This build number, however, isn't compatible with the version number required by MSIX packages, which must follow the convention **x.y.z.0**. We can change the build number by editing the YAML file and modifying the build configuration. To achieve this goal, go back to **Pipelines > Builds** in your Azure DevOps project, locate the build pipeline you previously created, and click **Edit.** You will get access to the advanced YAML editor.

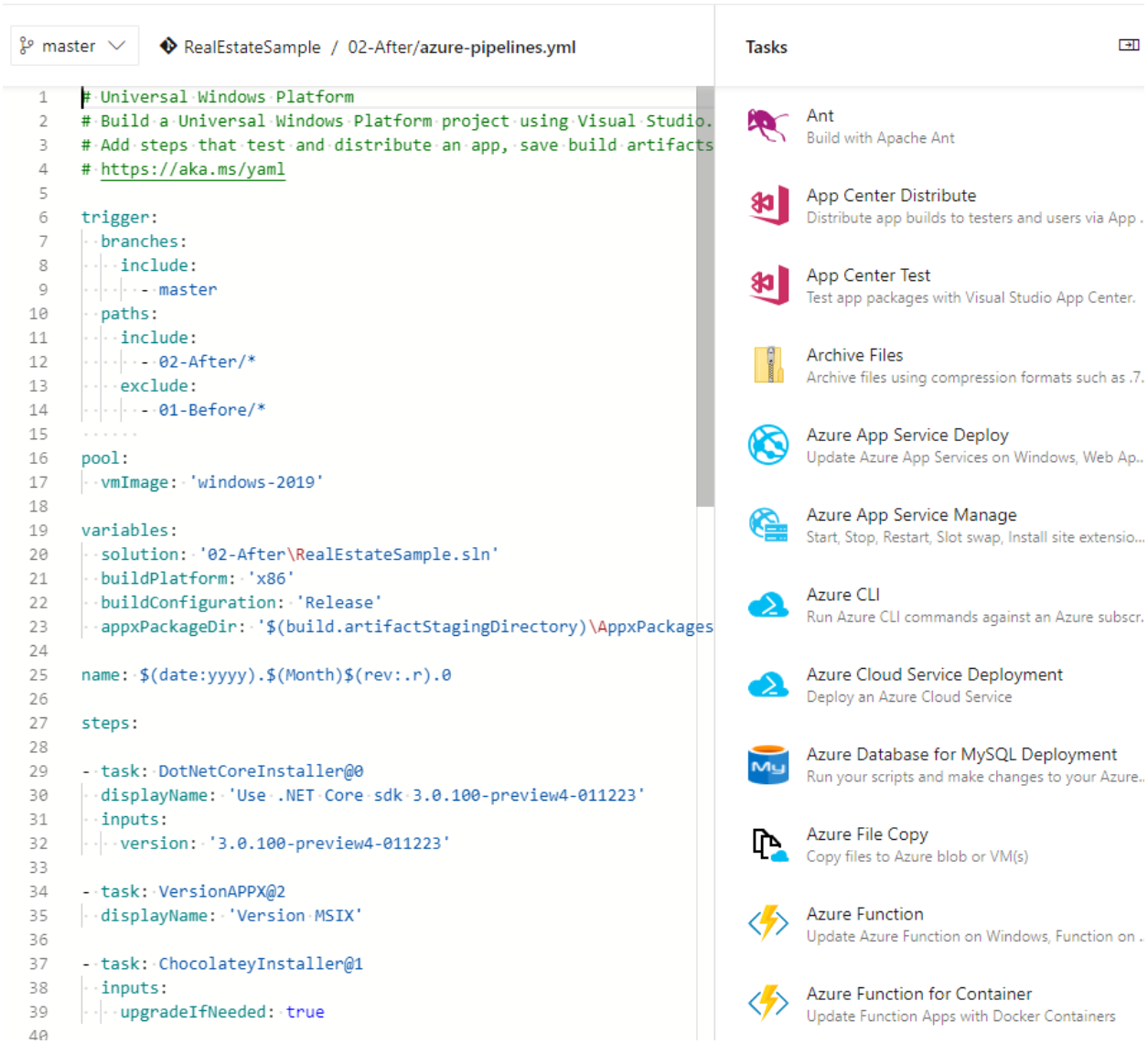


Figure 105: The advanced YAML editor

Compared to the editor you used the first time to create the YAML file, this version includes a more friendly user interface. From the panel on the right, in fact, you'll be able to choose from among many tasks and configure them using a visual UI. The option you have selected will be automatically translated into YAML markup.

To define the new build number, you have to add a new entry before the **`steps`** section using the following snippet.

Code Listing 83

```
name: $(date:yyyy).$(Month)$(rev:.r).0
```

With the **`name`** entry, we're defining a new versioning for the build number. This time, we're generating a version that is compatible with the MSIX requirements.

The last step is to apply this build number to the manifest of our MSIX package. We can use a task created by a third-party developer. Save the YAML file you have updated, and then go back to the Azure DevOps dashboard. Locate the marketplace icon at the top and choose **Browse marketplace**.

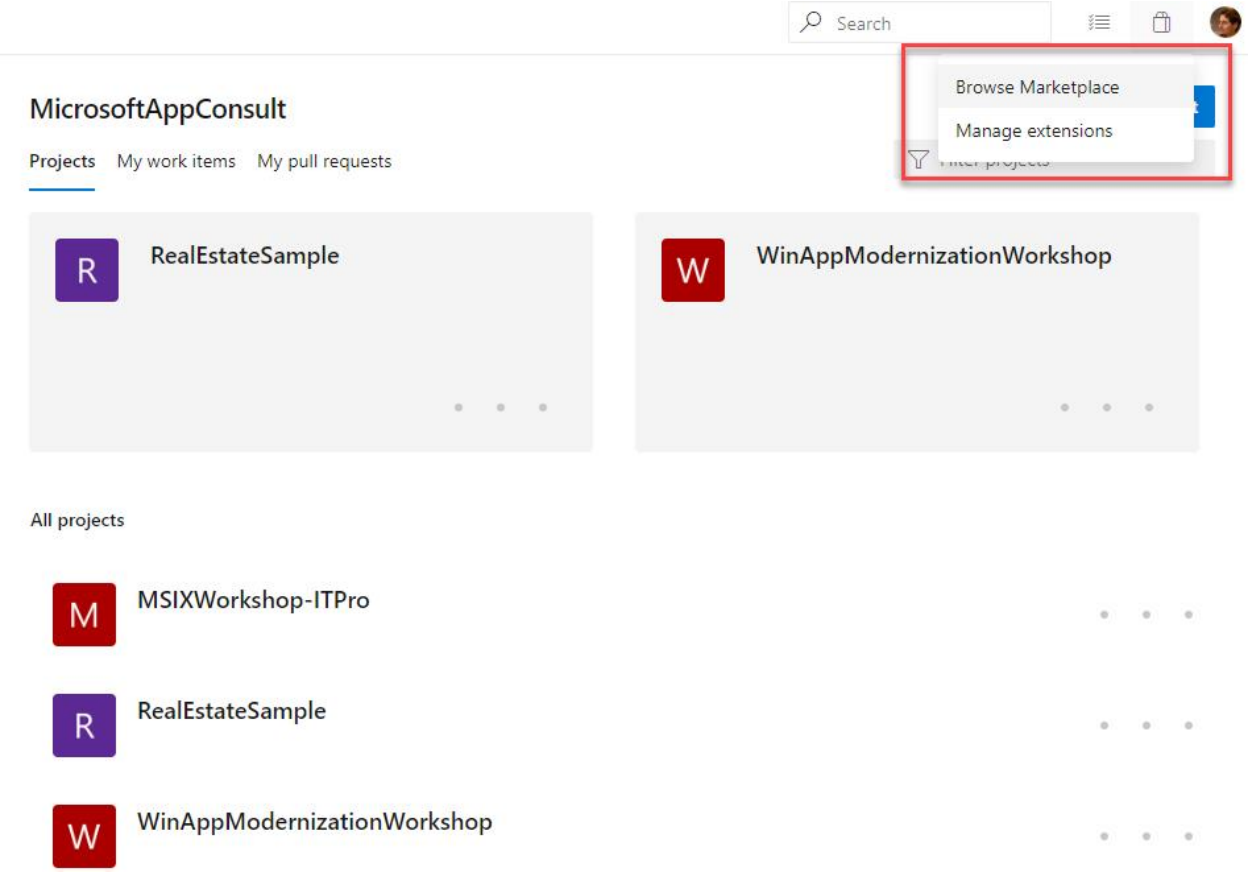


Figure 106: The icon to access to the Azure DevOps marketplace

Search for an extension called **Manifest Versioning Build Task** by Richard Fennell, click it, and then click **Get it free**. You will initialize the process to add the extension to your Azure DevOps account. Once the extension has been installed, you can go back to the pipeline, click **Edit**, and add the following step before the **`VSBuild`** step.

Code Listing 84

```
- task: VersionAPPX@2
  displayName: 'Version MSIX'
```

This task doesn't require any special parameter. It will simply edit the manifest of your project and apply the build number as the version.

Note: Since the YAML file is stored in the repository of your project, you can edit it on your local machine using an editor like Visual Studio or Visual Studio Code. The latter also offers an extension that adds IntelliSense support for the various tasks offered by Azure Pipeline.

Supporting App Installer

If you want to support App Installer, the easiest way is to let it be generated by Visual Studio. It's enough to generate a package just once in your local environment using the **Store > Create app package** wizard. Just make sure to choose **Sideloading** and keep the automatic updates feature turned on. Otherwise, you can manually edit the **.csproj** file of the Windows Application Packaging Project and include the highlighted changes in the main **`PropertyGroup`**.

Code Listing 85

```
<PropertyGroup>
  <ProjectGuid>c807bb90-d408-451a-b267-4972342402a2</ProjectGuid>
  <TargetPlatformVersion>10.0.17763.0</TargetPlatformVersion>
  <TargetPlatformMinVersion>10.0.17763.0</TargetPlatformMinVersion>
  <DefaultLanguage>en-US</DefaultLanguage>

<EntryPointProjectUniqueName>..\ContosoExpenses\ContosoExpenses.csproj</Ent
ryPointProjectUniqueName>

<PackageCertificateThumbprint>99461B8F2CD60990EC93FBF5652E877B7D4CE02A</Pac
kageCertificateThumbprint>
  <AppInstallerUri>https://db-msixtest.azurewebsites.net/</AppInstallerUri>
  <AppInstallerUpdateFrequency>1</AppInstallerUpdateFrequency>

<AppInstallerCheckForUpdateFrequency>OnApplicationRun</AppInstallerCheckFor
UpdateFrequency>
  <GenerateAppInstallerFile>True</GenerateAppInstallerFile>
  <AppxAutoIncrementPackageRevision>True</AppxAutoIncrementPackageRevision>
  <AppxBundlePlatforms>x86</AppxBundlePlatforms>
</PropertyGroup>
```

These properties match the information that you can set during the visual wizard:

- **`AppInstallerUri`** is the URL of the website of the network share where the package will be deployed.
- **`AppInstallerUpdateFrequency`** defines how many hours should pass between each update check.
- **`AppInstallerCheckForUpdateFrequency`** defines when you want to check for updates (at application's launch or in the background).

`GenerateAppInstaller` is a generic property that must be set to **`true`** in order to generate the .appinstaller file as part of the build process. You can go back to Chapter 7 if you want to know more about the parameters supported by App Installer.

You can also leverage an App Installer template as described in Chapter 7. It will be picked up by the build process and used to customize the App Installer file that will be generated.

Create a release pipeline

Now that you have a build pipeline that produces an MSIX package, you can define a release pipeline that will deploy it to your users.

Release pipelines are created in the **Pipelines > Release** section of Azure DevOps. Building a release pipeline is a bit different than the steps we saw in the previous section, as you can see from the starting template in Figure 107.

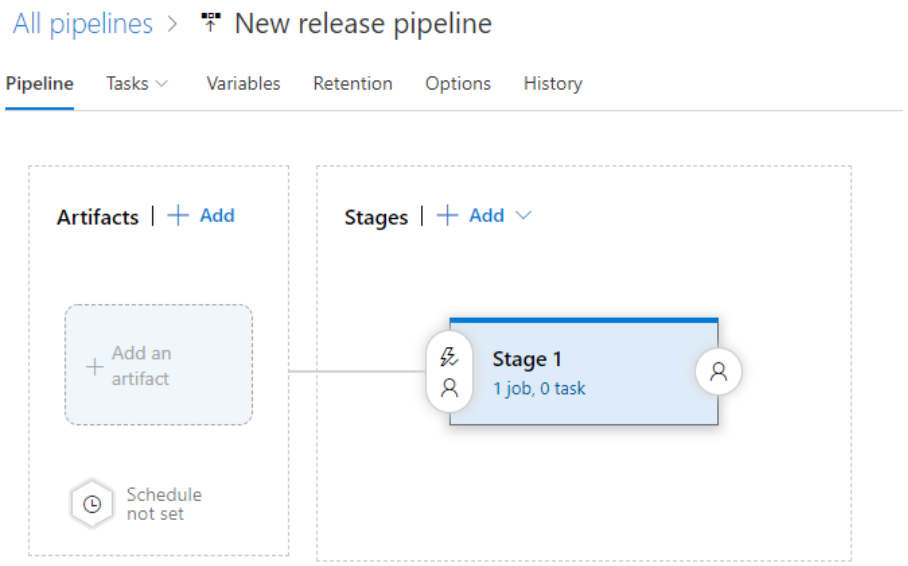


Figure 107: The template to create a release pipeline

In the first section, called **Artifacts**, you will have to specify which build output will be used for deployment. The default choice is to use the build artifacts, so you will have to select from the list the build pipeline you previously created.

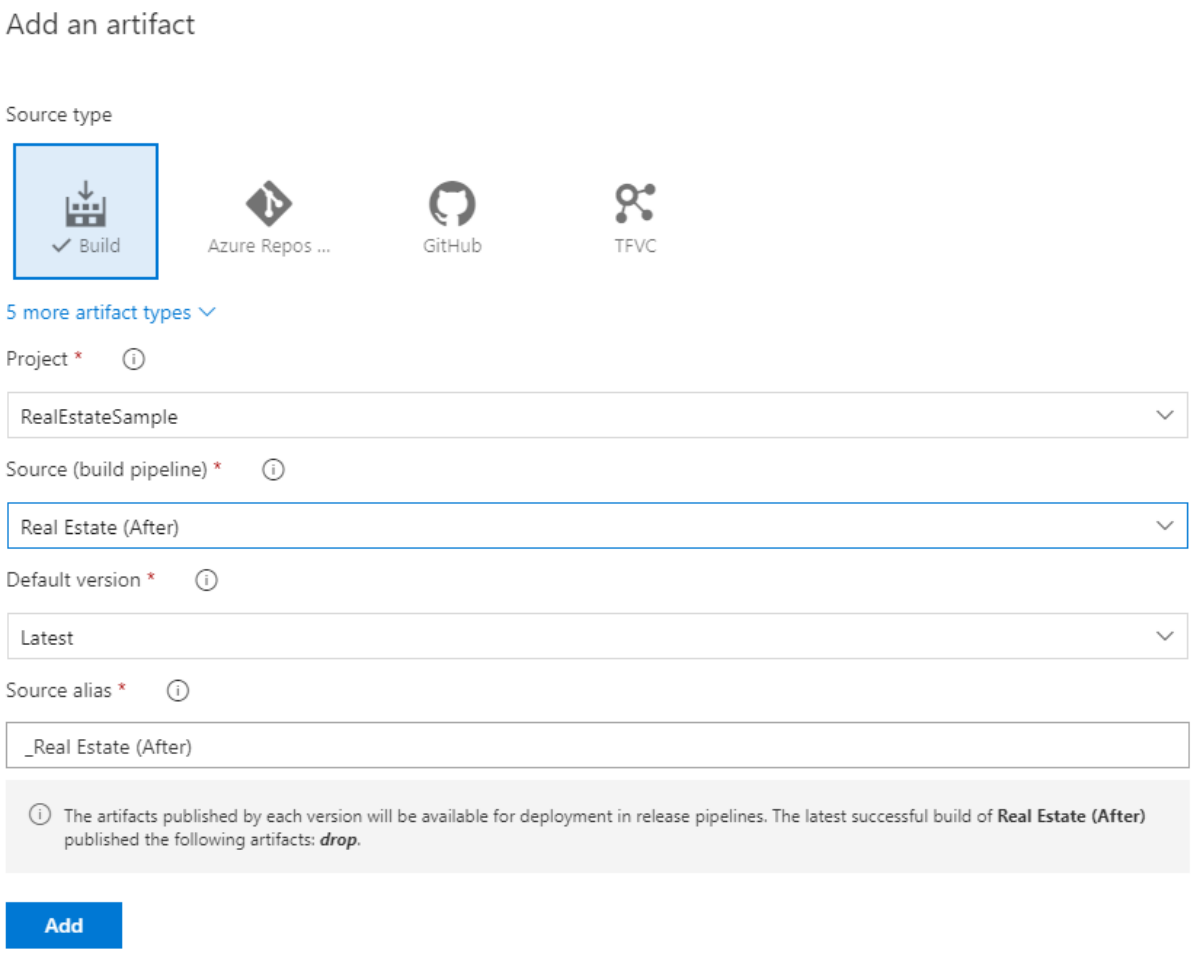


Figure 108: The option to choose the build artifact generated by a build pipeline

Once you have added it, you will notice a lightning symbol near the artifact's name. Click it and enable the **Continuous deployment trigger**. It will turn the pipeline into a CD pipeline, which will trigger a new deployment every time a new build pipeline is successfully completed.

In the second part of the template, you can create one or more stages, which are the various phases of the deployment. Each stage is typically mapped with a different environment: development, testing, production, etc. Each stage can run one or more tasks, which will take care of performing the actual deployment.

In order to configure a stage, you just need to click the link below the stage name. You will get access to the visual task editor.

Signing the package

As previously mentioned, it isn't a good practice to sign the package in the build pipeline. The best place to perform this task is the release pipeline, since it allows us to store the certificate in a safe way, so that we don't have to share it with other developers.

To achieve this goal, we need to leverage another extension from a third-party developer. Go back to the Marketplace, look for an extension called **Code Signing** by Stefan Kert, and install it to your Azure DevOps account.

After that, go back to the release pipeline you're building, and press the **+** sign near the agent job to add a new task. Look for the task called **Code Signing** and add it.

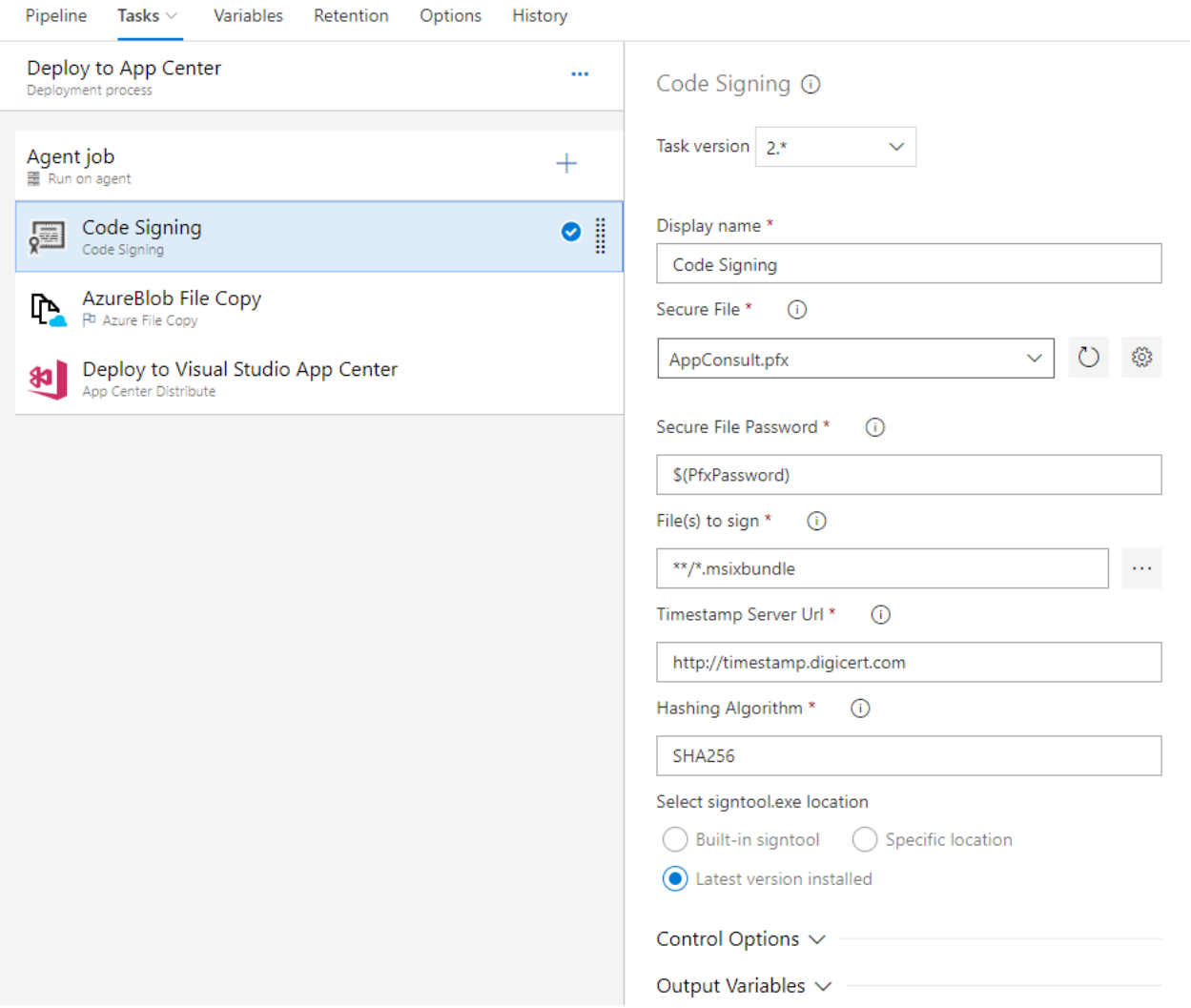


Figure 109: The code signing task

This task will launch, under the hood, the **signtool** utility to sign your package, using the information provided in the configuration:

- **Provide the certificate.** Just click the **Settings** icon near the **Secure File** field and upload a PFX certificate from your machine. The file will be stored using the Secure File feature provided by Azure DevOps. Thanks to this feature, the file will be safely stored in the cloud without giving anyone the opportunity to download it. The only options are to leverage it in a pipeline or delete it.

- **Define the password of the certificate.** It isn't a good idea to provide the password in clear, so for the moment we just define a variable called **PfxPassword**, which we're going to define later.
- **Specify the file to sign.** Since our artifact will contain only an **.msixbundle** file, we can just use a wild card to specify this extension.

Once you have saved the task, you can move to the **Variables** section of the pipeline to define the password. Just click **Add** and set as the name **PfxPassword** and, as the value, the real password. Then click the lock icon displayed near the field to hide its value.

Deploying the application

The next step is to add a task to deploy the MSIX package, together with the App Installer file and the HTML page, in a location your users will be able to reach. Azure DevOps provide multiple tasks that can be used to achieve this goal:

- You can use **AzureBlob File Copy** if you want to host your package on an Azure Blob Storage.
- You can use **Azure App Service Deploy** if you want to host your package on an Azure web application.
- You can use **FTP Upload** if you want to host your package on a website hosted by any web provider.

Going into the details in this book would be off topic, since there isn't a unique solution, but it all depends on the requirements of your project. Additionally, all the tasks are easy to configure. For example, if you want to deploy your package using Azure Blob Storage, you will just have to link your Azure DevOps account with your Azure account and choose which one of your storage accounts will be the destination. Or if you want to deploy over FTP, you will have to provide the FTP URL, port, username, and password.

Another option is to deploy the MSIX package directly on the Microsoft Store, thanks to an extension created by Microsoft. Once you have installed this extension on your Azure DevOps account, you'll be able to add to your release pipeline one of the two available tasks:

- **Windows Store – Publish** to publish the application as public.
- **Windows Store – Flight** to publish the application in a private flight ring.

The first step is to configure the service endpoint, which will allow Azure Pipeline to authenticate to the Store using your Dev Center account. Then you must provide the Application ID, the new metadata (if you want to update them as part of the process), and a reference to the MSIX package created by the build. The selection of the package is made easy by the artifact explorer, which you can invoke by clicking the three dots near the **Package file** field.

However, there's a catch. After you have selected the MSIX package, the output will look like the following.

Code Listing 86

```
$(System.DefaultWorkingDirectory)/My build
pipeline/drop/ContosoExpenses.Package_2019.5.23.0_Test/
ContosoExpenses.Package_2019.5.23.0_x86.msixupload
```

As you can see, the path contains the version number of the package, which will change at every build. As such, the release pipeline will complete successfully for the current build, but it will fail for the next ones. The solution is to use one of the global variables, **`Build.BuildNumber`**, which will be automatically replaced with the correct build number at every iteration.

Code Listing 87

```
$(System.DefaultWorkingDirectory)/My build
pipeline/drop/ContosoExpenses.Package_$(Build.BuildNumber)_Test/
ContosoExpenses.Package_$(Build.BuildNumber)_x86.msixupload
```

Thanks to this task, the updated MSIX package will be automatically submitted to certification at the end of the CI/CD process.

Wrapping Up

We have reached the end of our book! During this journey, we have learned how MSIX, the new packaging format introduced in Windows 10, is the perfect starting point to move your applications forward, no matter if you are an IT pro or a developer. IT pros will be able to deploy applications inside an enterprise in a more agile way by decoupling them from the operating system and the customizations required by the company. Developers won't have to build and maintain an installer technology anymore, and they can instead focus on the application's code. Additionally, they will be able to add new Windows 10 features to the application without rewriting them from scratch, instead starting from the existing code base.

The first part of this book was dedicated to IT pros:

- Thanks to tools like the MSIX Packaging Tool (which was discussed in Chapter 2), IT pros can repackage applications even without owning the source code, by just using the standard installer as the starting point.
- By using modification packages, which were the main topic of Chapter 3, IT pros can include the customization required by the company in a dedicated package, independent from the main application. This means that, whenever the developer releases a new update of the application, it can be deployed right away. The IT pro engineer no longer has to repackage the application every time.
- MSIX provides a lightweight container, which helps Windows to be more effective in the application's deployment. However, it may introduce some friction for some applications and introduce some unexpected behaviors. Thanks to the Package Support Framework, which was described in Chapter 4, IT pros can fix some of these problems without having to change the source code.

The second part of the book was dedicated to developers:

- In Chapter 5, we learned how Windows developers can repackage their applications directly from the development environment, Visual Studio. The Windows Application Packaging Project gives developers the opportunity to repackage WPF, Windows Forms, and Visual C++ applications straight from the source code. Additionally, thanks to the integrated manifest editor, you can easily integrate your apps with the operating system by registering for file-type associations, defining execution aliases, setting a startup task, etc.
- MSIX is not just a better way to deploy and distribute applications, it's also the starting point to move your applications forward and integrate them with all the new features that Windows 10 has added in recent years. Before MSIX, only full Universal Windows Platform applications were able to take advantage of the new features. Thanks to MSIX, you can use APIs from the Universal Windows Platform in your Win32 applications; you can use XAML Islands to integrate new UI controls, which provide better support for accessibility and new input technologies, like touch and inking; and you can leverage new components like background tasks and app services. We explored all of these opportunities in Chapter 6.

MSIX also opens new opportunities to deploy your applications. You can leverage the existing technologies, like SSCM or Intune, but also new and exciting ones like the Microsoft Store. You can even use traditional approaches (like deploying the application on a website or a network share) and gain new capabilities like auto-updates thanks to App Installer. All these deployment approaches were described in Chapter 7.

In the end, as a developer, you can combine the power of MSIX and its deployment flexibility to be more agile and to introduce a DevOps approach in your projects. In Chapter 8 we learned how, thanks to MSIX and Azure DevOps, you can go from the source code to the deployment automatically, without any action from your side, other than committing new code to the repository where your project is hosted.

These are exciting times to be a Windows developer!

MSIX Labs

Microsoft recently published an open-source project on GitHub called MSIX Labs. It offers many exercises and a step-by-step manual that covers the MSIX Packaging Tool, the Package Support Framework, modification packages, and many other concepts we have covered in this book. The exercises also come with a series of overview videos created by the MSIX team and the Windows AppConsult team that will guide you through the tools and exercises.

Both the exercises and the videos are highly recommended if you want to put into practice what you have learned in this book!