Powered by Advanced INSTALLER

MSIX Packaging Fundamentals

Tim Mangan
Bogdan Mitrache
Kevin Kaminski

Contents

About the Authors

Tim Mangan

Tim is one of the most known and respected professionals in the packaging industry, awarded as Most Valuable Professional (MVP) by Microsoft in Application Virtualization for the last 12 years, as well as a Citrix CTP Fellow for lifetime achievements, is an author of multiple books, trainer, and consultant. He is currently contributing to the industry through his own company TMurgent Technologies, building products and providing support about application virtualization.

You can have a piece of Tim's knowledge on TMBlog[1].

Bogdan Mitrache

A skillful product management professional with proven experience in the computer software industry, Bogdan Mitrache is the VP of Product of Advanced Installer. He is involved in all aspects of the organization and uses his deep understanding of the packaging and deployment technology to empower the Advanced Installer team to build critical educational resources and tools for IT professionals and developers.

Bogdan is obsessed with MSIX and has authored multiple articles on the Advanced Installer Blog[2].

[1] https://www.tmurgent.com/TMBlog/
[2] https://www.advancedinstaller.com/blog/page-1.html

Kevin Kaminski

With over 20 years of experience with Microsoft technologies ranging from device management to the data center, today Kevin Kaminski is a knowledgeable Windows expert that shares his hands-on experience through training and consulting.

Also recognized by Microsoft as MVP for Windows and Devices for IT, his work now focuses mainly on modern management using Microsoft Endpoint Manager.

Check out the latest news from Kevin at Check Your Logs - Kevin Kaminski[3].

Special thanks to other expert contributors:

Alex Marin[4]
Horatiu Vladasel[5]
Radu Popescu[6]

[3] https://www.checkyourlogs.net/author/kkaminski/
[4] https://www.advancedinstaller.com/authors/alex-marin.html
[5] https://www.advancedinstaller.com/authors/horatiu-vladasel.html
[6] https://www.advancedinstaller.com/authors/radu-popescu.html

Introduction to Modern Applications

As an ever-evolving art, we constantly struggle to apply discipline and semi-automated tools to produce successful and timely deployments for end-users. Hence why we gathered some experienced colleagues and industry connoisseurs to address and write a book with a specific purpose: to help enterprises properly prepare and deliver Windows applications.

In the sea of changes that have occurred in application deployment over the last 20 years, perhaps the most important is **the evolution of separation**. Rather than having a single entity, a hard disk filled with an unorganized mess, today we prefer to split up the OS from the hardware, the applications from the OS, and the data from all of the above. We use the term “Modern Applications” to encompass several forms of packaging, delivery, and run-time systems, which enable application separation. Even when delivered in bundled images, organizations today treat and manage applications as individual components (at least for part of the process), and those processes can be improved by applying these disciplines.

We will primarily focus on Windows Application packaging and how companies prepare applications for their distribution and use, either within their company or their clients' organizations. While the content of this book is fundamentally intended as an introduction to the practices of application packaging, we believe that it is also valuable even for the most experienced IT Pros. We hope to help them evaluate a different perspective that could open their way to different packaging techniques from the ones that they are used to and understand how these techniques fit into the larger picture.

Modern, and other, Application Forms

Software has traditionally been provided to you from internal and external software developers in one of the traditional forms of EXE and MSI installers. Organizations prepare the application, customizing both the form of delivery and the contents in ways that improve efficiencies and end-user satisfaction.

Organizations that have moved to modern forms have most likely done so on their own. We see that this is starting to change with MSIX, as vendors appear to take interest in providing future applications in this modern format. But, even when delivered in the modern format, organizations will still need to process and configure the application for delivery.

The modern form includes technologies such as Application VIrtualization, Application Layering, and Application Containers (where MSIX falls).

The Purpose of Packaging

The reason we package an application is to simplify the life of the end-user, shielding them from things that they don't need to be involved with. Sometimes, it is a convenient way to hide aspects from the end-user, such as licensing keys or back-end infrastructure details that they do not need to know. Some of the benefits of application packaging for companies include:

- The customization of general-purpose applications to fit the needs of the organization.
- Increased productivity of end-users - The majority of which are not IT Pros.
- A wider applicant pool for a given position (they don't need to be computer experts but know how to use the app),
- Increased security of its systems by reducing the need or use for admin credentials.

Ultimately, the purpose of packaging comes down to reducing costs. While certainly there are costs associated with the packaging process, these are outweighed by the cost savings that come from simplifying a process that allows end-users to focus on the tasks they are being paid for - that don't necessarily include installing software and its complications.

Handling Dependent Reusable Components

More often than not, Application packages include dlls components that are reusable from other parties. The most common are the Microsoft Visual C++ runtime components (although there are many other sources as well).

In a native installation, these components may be installed either into system locations or in the same package folder with the application. In the application repackaging industry, there are two different schools of thought on how to handle these components.

These two approaches are as follows:

- One is to identify dependencies, separating them for independent delivery, and removing them from the repackaged app.
- The second is to keep them within the repackaged app.

Although some organizations believe their approach is the right one to use, the truth is that: *there's not a one-size-fits-all method that works for all organizations,* **and** *using only one approach may not be suitable for all situations within any organization*.

Traditionally, we worried about version conflicts when these components were written into shared areas, a concept referred to as “dll hell”, where installing one new application might break any of the others on a system and you wouldn’t know unless you test all of them.

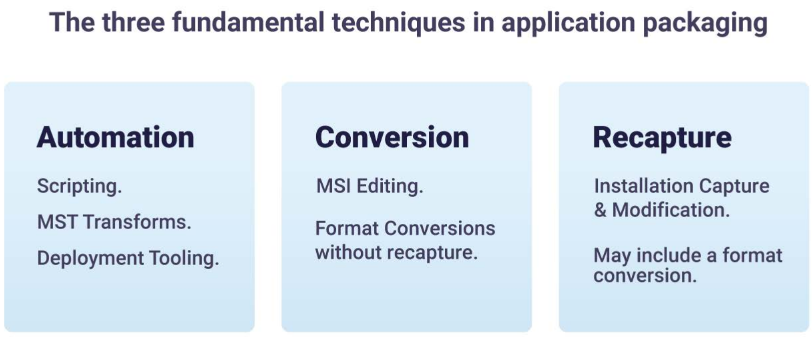
Today, different versions for most of these components are installed into different file locations, even in a system installation, so it is less of an issue. The problem now tends to be with older application installers that add known flawed shared components that lack fixes for known security vulnerabilities, and we want to ensure these are always patched.

Organizations using Configuration Manager to deploy apps repackaged as MSIs usually prefer the separation approach. While organizations repackaging for an isolation environment, such as App-V or MSIX tend to take the other approach; which takes much less work and it is less unlikely for vulnerabilities to be exploited, especially in an isolated environment.

We're not here to guide you through what the best approach is for you. But, to make you aware of these choices we recommend that you follow the approach(es) used by your organization. We can also suggest that as you gain more experience and changes occur in the technologies we use, the so-called “correct” approach today may not be so “correct” in the future.

The Techniques of Application Packaging

There are three general techniques used in application packaging.


The three fundamental techniques in application packaging
Automation
Scripting.
MST Transforms.
Deployment Tooling.
Conversion
MSI Editing.
Format Conversions without recapture.
Recapture
Installation Capture & Modification.
May include a format conversion.

- **Automation**. This encompasses techniques that do not alter the vendor installer directly but are used to apply it. This includes the use of scripts wrapping the application installer, as well as the use of tools that provide additional files such as MST Transforms.
- **Conversion**. Converting an installer file from one form into a new form. This includes the use of MSI editors that produce modified MSI files and format conversions (such as from MSI to either App-V or MSIX) that do not require an installation capture of the installer.

- **Capture/Re-packaging**. Performing the installation and customization of an application within a monitored environment to capture the changes and produce a customized package.

As someone responsible for application packaging, you will probably need to use a variety of these techniques over time. Your organization will likely have selected a small subset of all of the available first and third-party tools to help make packaging activities consistent and reliable. The MSIX Packaging Fundamentals book will provide you the knowledge that you need to succeed no matter what techniques or tools are used.

The Traditional Software Installer

Software is generally provided in the form of an installer. The installation is primarily a combination of writing files and entries into the Windows file system, often utilizing system utilities to perform registration activities.

Most of the software you work with comes in the form of an EXE or MSI based installer. The MSI installer is usually preferred for packaging as it uses a set of tables that can be easily examined and manipulated, while the EXE-based installer tends to be less transparent. Sometimes the EXE-based installer embeds an MSI that is used as part of the process. But ultimately, the EXE-based installers and any custom actions included in an MSI are binary “black boxes” that we can only understand by watching what they do.

The purpose of the vendor installer is not only to place the appropriate components in place on the system but also to configure the application for the user. Often, this involves some decision-making from the installer's side based on the environment it is being installed on.

This adaptation to the environment could potentially include examining the machine, the OS, the end-user, as well as other software on the machine, and even the surrounding network including other servers or services.

The role of the packager and packaging software is to end up with a generic package that may then be deployed to any system with organization-wide customized defaults. Special care is required when the techniques used include a repackaging process. Issues can arise in repackaging due to the unintentional capture of unwanted components or configurations by the installer that are specific to the capture machine.

> The goal of any repackaging process that you are involved with should be to produce a clean and customized package that will work on any intended system, along with appropriate documentation that would allow that same package to be reproduced in the future.

On Spoofs, Redirections, and Overlays

Virtualization, Layering, and Containerization all employ similar techniques to achieve their goals. The details of how and when these techniques are used vary, but first, you need to have an understanding of the basic techniques that they employ.

We will refer to them as Spoofs, Redirections, and Overlays. In essence, Redirections are a specific form of Spoof, and Overlays are an application of redirections, so technically, we could just use a single term - but as we will see, there is value in separating them out into three terms.

Spoofs

In reality, everything done in modern application runtimes falls under the “spoof” category. We can define a spoof as something that intercepts an action (function call, for example) and alters the requested action or result. In MSIX terms, a Package Support Framework shim, which intercepts a Windows API call and modifies it, implements various kinds of spoofs.

Modern application runtimes routinely implement spoofs for file redirection, changing a COM GUID, or renaming a semaphore for isolation purposes.

Redirection

A particularly popular form of spoof is the redirection spoof. Used against calls to the file system or registry, it takes a filepath or registry key path as requested by the application and in essence, it says: “I know you asked for this, but I’m going to give you this other thing instead”.

Fundamentally, this allows for traditional code to not know that it is not running in a modern environment. It might think that it is installed to the default “C:\Program Files\VendorName” folder, while all of the files are located wherever the modern app runtime puts them.

A common use of Redirection used in several modern systems involves a folder called VFS (for *Virtual File System)* along with subfolders with specific names that refer to an equivalent “un-redirected” location. Thus “VFS\ProgramFIlesX64\VendorName” equates to the “C:\Program Files\VendorName” folder that the app might request.

Layers

Layers are a visual analogy used to describe the concept of how the modern runtime systems act.

The visualization we use to describe the layering concept is called *panes of glass*, whereby a piece of software, depicted in the image as an Application, sees the system through a series of glass panes. When items are placed on a glass pane, they can add to, or replace parts of what the application would see without them.

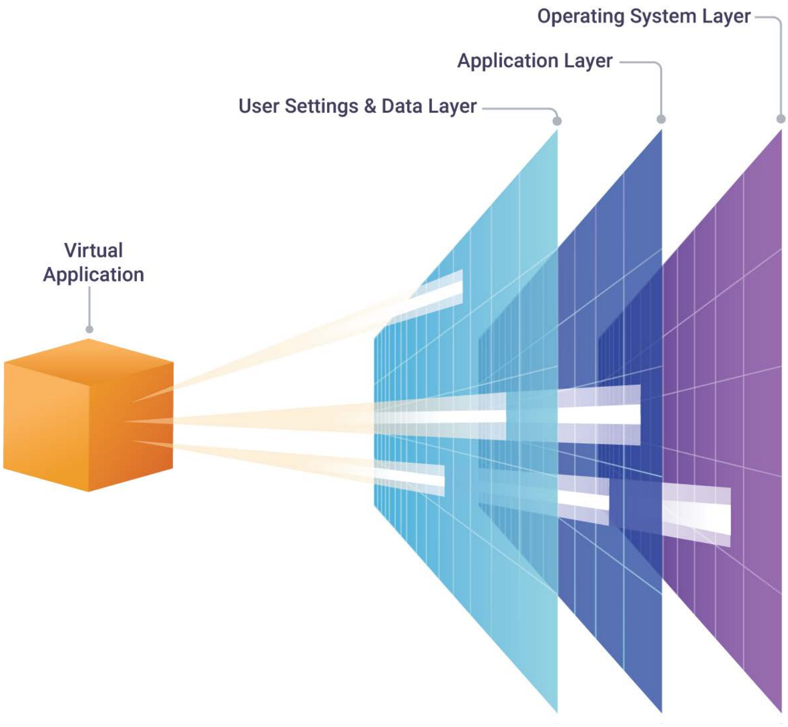


Illustration of a virtual application

In application runtime systems, we usually think of a model such as the one shown above, where the application is working with three layers. Drawn horizontally, the rightmost layer is the operating system with its file system and registry. Just left of that is the application layer, containing additional and/or replacement files and registry entries. The application layer is immutable, meaning that the runtime system does not allow changes to occur in that layer when the user runs the application. Instead, changes to the application are made to the third layer, shown left of it. We can call this third layer: **the user settings and data layer** for that application.

This makes a nice visual, but of course, things are always more complicated! Items written to the upper layer can be used only by the application, and often we want settings and data to be usable outside of this application. So while some information needs to be written to that layer, other pieces need to be allowed to be written to system locations, such as the Documents folder, cloud sharing folders, home drives, and network shares. As a general rule, the runtime systems generally use the presence of items in the application layer to make this determination.

The layering concept is implemented by using redirections. Redirections may be implemented in the runtime as user mode intercepts (added to the application process by dll injection) or kernel filter drivers. Some runtime systems employ both techniques in different areas. In a redirection, the request by an application to perform an operation, such as opening a file with a certain file path, is examined and the request is redirected to a different location, which we think of like one of those panes of glass. In reality, this is just a different location on the system, but it is easier to just think of it as that glass pane.

Combining these techniques to solve application challenges

While the effect of redirecting a file or registry call is simply giving the application the best location choice, internally it is much more complicated. Consider the following case:

- The OS has a folder with some files in it.
- The application layer has additional files in its overlay folder.
- When running the application, the end-user added a new file to that folder or modified an existing one, from the third layer.
- If the application then makes a request to find files in that folder, the redirection software must query folders on all three layers to form the query reply.

Such a situation requires a combination of spoofing, redirection, and layering and is just one example of how modernizing applications alters what the application sees and does.

Moving Towards MSIX

Microsoft is making a major commitment with MSIX. In the future we expect MSIX to become the preferred format for software vendors to deliver applications to their customers, overtaking the ubiquitous MSI during this decade. For the software vendors, it is about staying current in their products so that they may take advantage of the new OS features that will only become possible when writing code that runs inside the MSIX container. For Microsoft customers, MSIX offers the promise of simpler application customizations, secure and safe installs and more stability for the end user.

MSIX is evolving as it is being delivered. In the remaining chapters of this book we will focus on the fundamental technology as it stands today, look at currently available tooling. After that, we then dive into both packaging and deployment fundamentals which should provide you with a complete picture to work from for the future.

MSIX Technology Fundamentals

What is MSIX

Microsoft presented MSIX[7] in 2018 as an improved version of the AppX package (initially used only for UWP apps) to better support traditional desktop applications on Windows 10. MSIX is the result of their experience with MSI and App-V packages and the Desktop Bridge[8] program.

Structure-wise, an MSIX package is very similar to an AppX or App-V package. It is basically a zip package that contains your application files and some configuration XML files.

While MSIX is designed to support the latest development trends, it also comes with significant support for older Win32 (which includes traditional x86 and x64 unmanaged-code apps and DotNet Framework-based applications), e.g., the standard desktop applications that we've been using all of these years. Now, you can package your standard desktop application using the new MSIX format and deploy it using the tools you already have (SCCM, Intune, etc...).

[7] https://docs.microsoft.com/en-us/windows/msix/
[8] https://developer.microsoft.com/en-us/windows/bridges/desktop

You can also take this new package and publish it in the Microsoft Store or for direct download from a website while leveraging all the new advantages from the modern Windows APIs (if the app is still in development).

Not all Win32 applications can be packaged and launched inside an MSIX container. Microsoft has prepared a list of requirements[9] -- you can check this article to make sure your application is suitable for migrating to MSIX.

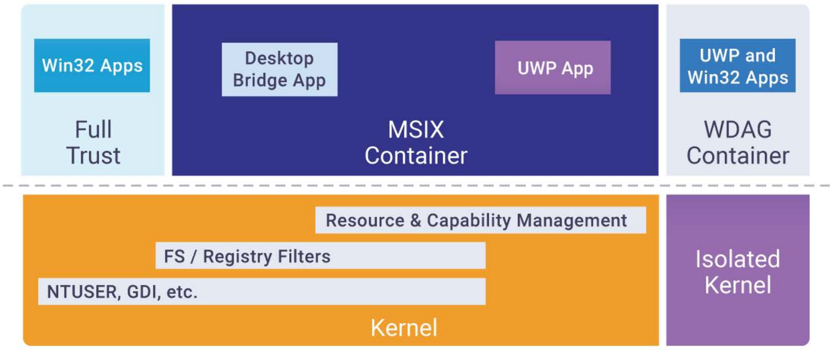
The MSIX Container

The application delivered with an MSIX package runs on systems inside a container. This container, sometimes called "Helium" by Microsoft, is a lightweight version of the containers on Windows Servers that are meant to be used by services such as Docker. As you will see, we will sometimes refer to this runtime environment as the MSIX Runtime.

Depicted below, you will find that the MSIX container can bring together:

- MSIX packaged applications (full trust model),
- Applications packaged as Desktop Bridge apps (files and registry virtualization),
- Applications packaged using UWP apps (the capability model).

[9] https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-prepare



MSIX Container

These different programming models share the container, but do not necessarily support the same features.

Classic Win32 apps converted to MSIX have runtime functionalities that UWP apps do not have. This is because they declare a special capability called “runFullTrust” in the manifest. The name refers to the trust level granted within the container and is not related to UAC elevation on the system, so it may be a bit misleading.

While the apps we deal with these days are either traditional Win32 based code or UWP based code, we're noticing a shift of direction. Now, developers with source code access may build a hybrid app that leverages both platforms, as long as they build it as an MSIX package. As it is more of a bolt-on approach for the developer, today we think of this as a hybrid, but eventually, we will likely begin to think of these as the “native MSIX apps”.

Classic Win32 MSIX packaged applications will still run only on desktop devices, and they don't “become“ universal apps simply because we are packaging them in a new format. These converted apps are still compiled for x86 and x64 processors and need all of the traditional Windows dlls, so they cannot run on non-Windows tablets/phones or other non-supported devices. In general, MSIX packaged Win32 apps currently only work on Windows 10, running x86 or x64 Intel/AMD processors.

There is one exception, the Always-Connected-PC[10] devices, running Windows 10 in S mode, where Win32 apps run on ARM devices with emulation support (included in the OS) and specific ARM CPUs.

Remember. You can deploy an MSIX package of a Win32 app-only on desktops or Windows 10 devices.

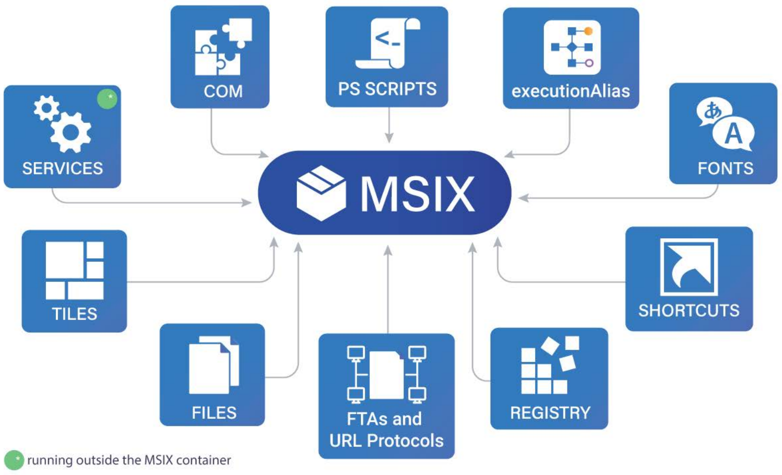
System Interfaces

The MSI or EXE installation of a traditional application does much more than lay down files and registry items. During the integration of an application in the system, the installation touches various system locations - which could negatively impact other existing applications and the system itself.

Adding an MSIX or a UWP package to the system is different. For the most part, everything exists within the container and very few resources are exposed outside of it. The resources that are exposed are defined in the package manifest. Because of this, you have more control over possible impacts on the system through the AppInstaller software when adding or removing the package.

[10] https://www.microsoft.com/en-us/windows/s-mode

The following diagram indicates the OS resources supported at this time (for a Win32 application) that an MSIX package can install. On top of these, from your source code you can enable your application to also use the new native UWP components (app services, background task, etc) to better integrate with the OS or other applications.



MSIX - Supported OS resources

Here is a summary of what an MSIX package can natively install today:

- **Files**
- **Registry** (available as a package Application Hive only seen within the container)
- **Fonts**
- **Tiles** (a replacement of the common shortcuts you've created so far)
- **Services** (which get installed and run outside the container)
- **COM**
- **File Associations**

- **URL Protocols**

One important thing to understand is that all of the above resources are defined in a completely new manner for MSIX packages. Gone are the days when you could write to the registry to define a file type association or integrate a COM component. You can read more about it in our "AppxManifest.xml aka the Package Manifest" chapter.

If you're looking for drivers, these are not supported directly in MSIX packages but are referenced as an external component. Microsoft recommends that all drivers are uploaded to the Microsoft Store by the hardware providers, so the OS manages their installation automatically for the end-user.

As MSIX is still in development, we can still expect changes, but until then, you can use the hybrid-solution adopted by the App-V folks, where drivers still get deployed using our old friend, the MSI package.

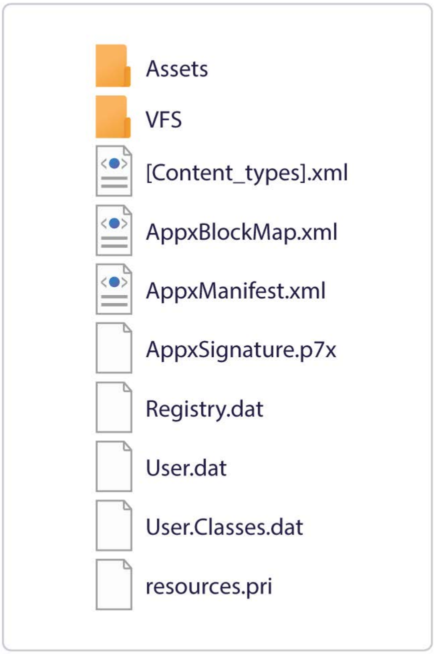
Important. The list of supported resources is constantly updated by Microsoft, check Advanced Installer Blog - MSIX Support Windows[11] article for the latest version.

The MSIX Package Layout

The image below shows the most common and basic MSIX package layout. You can see a similar list of resources if you extract an MSIX package using makeappx.exe or using a tool like 7-Zip.

[11] https://www.advancedinstaller.com/msix-support-windows.html

Extracting the package this way makes it quick to inspect its contents without using a dedicated editor/viewer. The structure you see below is exactly the same one that will be expanded by the OS in the installation folder, under %ProgramFiles%\WindowsApps.



Contents of an extracted MSIX package

If you have experience using App-V, you can notice the similarities between the two package formats. Unlike App-V, the MSIX package does not come with additional external configuration files.

Let’s have a quick look at each of the main areas from the package.

Assets - As the name implies, this folder is where all the app's graphics assets should be found. These are usually generated/extracted automatically by the packaging tool.

MSIX packages are much more aware of the graphical capabilities of a device and can carry graphical assets for any screen size and DPI, while optimizing their delivery.

VFS - This folder normally contains the application binaries. DLLs, EXEs, config files, and so on. The VFS and Assets folders are also known as the *app payload*.

At installation time, the files from the VFS folder are overlaid on top of the system's known folders. Your application will perceive these files to be in the well-known system locations (*like Program Files or System folders*), when in fact they are all found in the redirected locations inside: `%ProgramFiles%\WindowsApps\package_name\VFS.`

Remember. Nothing gets written in the installation folder at runtime.

Any configuration files that will need to be updated by your application will actually be redirected by the third layer (application files and settings). You can read more about this in our VFS chapter.

In some cases, the application might fail to run from the VFS folder. In this case, the packaging tool should give you the option to place the application resources directly into the package root.

Registry.dat - This file stores the HKLM and HKCU registry entries of the app. Some packages also have additional .dat files that contain a copy of the user registry entries (User.dat), and the equivalent of the HKCR view (User.Classes.dat),

This means that your registry settings from the package are no longer installed directly on the machine, polluting the system registry hives. Instead, you have an Application Hive, only available to the application running inside the container with your package. Just as with the VFS, the OS maps the registry from your package on top of the registry from the system, making the application see a unified version of all the registry hives.

Remember. Your app can no longer use the registry to share data stored in application-specific folders with other applications (since these registry entries are only visible from inside the container of your app).
Also, you can no longer define FTAs or other similar resources by manipulation of the registry alone. That is now all done through the AppxManifest.xml.
A good repackaging tool, however, will use the older style registrations to automatically trigger the addition of the appropriate entries into the manifest.

Tip! If a ".dat" registry hive file is present in an MSIX package it indicates the packaged application is a Win32 one, not a full UWP application. Only MSIX packages for Win32 apps can contain registry ".dat" files.

Resources.pri - This file contains app resources, like localized strings and paths for additional resource files. Packages contain one file per language.

AppxManifest.xml - This XML file is the core resource of an MSIX package, as detailed in the following chapter. It contains all the information required by the system to identify, install, and run the application.

This is where you will find the:

- Applications installed by your package
- Package declarations & capabilities (FTAs, Services, etc.)
- Dependencies for other MSIX packages (redistributables, or the package that a modification package layers onto, etc.)

The package manifest is automatically generated by the packaging tool you use. You can also create/edit it manually and use **makeappx.exe** to manually package the extracted resources into an MSIX.

Note! While general compression tools, such as 7-Zip are practical for viewing the MSIX package contents, if you just rename a file from .7x to .msix, it will not create a valid package. Only makeappx, or a tool that uses makeappx under the covers, can generate a valid MSIX package.

AppxBlockMap.xml - This is a file generated at build time, by the packaging tool, it contains a list of all the app's binaries and their hashes. It is used by the system for integrity checks and for performing differential updates (lowering bandwidth usage by downloading only the changed files), as well as single-instance storage (across all packages installed on the system).

AppxSignature.p7x - This file stores the digital signature information for the package contents. Just like the block map file, it is automatically generated by the packaging tool.

Remember. All MSIX packages must be digitally signed with an SHA-256 code signing certificate.

If you build the package and deploy it internally in your company, you can also use a self-signed certificate (that was previously installed on all the machines where the package will be deployed, otherwise the package deployment will fail).

All the packages from the Microsoft Store are digitally signed with a Microsoft certificate (not the vendor's certificate) which is trusted by the OS.

Digital Signing is a completely new requirement for application packaging (which was optional for MSIs). We wrote an entire section on this topic to help you employ a streamlined process across your entire packaging team.

[Content_Types].xml - Contains information about the types of content in an MSIX package, used by the system at installation. This resource is also generated automatically by the packaging tool.

AppxManifest.xml aka the Package Manifest

The AppxManifest.xml file is the MSIX package manifest and an MSIX package contains a single manifest file.

MSIX bundles, which can carry multiple MSIX Packages such as one for x86 and one for x64 deployment, contain a manifest package for each MSIX package found inside that bundle.

The manifest holds the information defining the application and its features. This information is used by the system to install or uninstall, as well as to update and control the app's behavior during its lifetime.

Here is where you will find the definitions used for package installation, including:

- Applications (programs that have a user-accessible entry point) installed by your package;
- Services;
- Package declarations & capabilities (FTAs, Services, executionAlias, COMs, etc.);
- Dependencies for other MSIX packages (redistributables, or the package that a modification package layers onto, etc.)

This file is usually generated automatically by the tool building the MSIX package, be it Visual Studio, Advanced Installer, or any other packaging tool. Of course, you can create it manually, although it is not recommended. This file ends up in the installation folder of your app and it is read-only.

The contents of the file must follow the schemas imposed by the OS. These schemas may differ between different major Windows 10 updates, therefore it is very important to know what OS versions you're targeting with your app, otherwise, you might end up using features that are not available for your application.

Important. The list of supported resources is constantly updated by Microsoft, check Advanced Installer Blog - MSIX Support Windows[12] article for the latest version.

Manifest Sample Template

Below, you can find a basic template for a package manifest. Most apps that you will work with will have a larger manifest with more information, but this is the core information you should find. The manifest often contains the:

- **Schema References** - references to the various schema parts and updates that will be used.
- **App Identity** - this is a unique identifier of your package
- **Properties** - information about the app, such as the name to be included in the *Settings -> Apps* and other resources.
- **Prerequisites** - the criteria the app needs to meet to install properly
- **Resources** - the package/app location and others
- **Application(s)** - including:
  - The entry point(s) - This is what you see in the "Start Menu" after installation.
  - FTAs - Your file type associations.
  - URLs - Your protocol handler associations (not shown in the example).

[12] https://www.advancedinstaller.com/msix-support-windows.html

- **COM** - Not shown in the example.
- **Services** - Not shown in the example.

Remember. FTAs and other resources that you used to define in the registry when building an MSI must now be defined in the package manifest.

Other entries are optional. Optional entries added by updated schemas, such as the UAP10 extensions, may need a more up to date version of the Operating System to be used. You can find a full list of schema elements and requirements in this Microsoft Documentation[13].

```
<?xml version="1.0" encoding="utf-8"?>
  <Package xmlns="http://schemas.microsoft.com/appx/
                  2010/manifest"
      xmlns:uap="http://schemas.microsoft.com/appx/
                 manifest/uap/windows10"
      xmlns:rescap="http://schemas.microsoft.com/appx/
                    manifest/foundation/windows10/
                    restrictedcapabilities" >
      <Identity Name="MyCompany.MySuite.MyApp"
                Version="1.0.0.0"
                Publisher="CN=MyCompany, O=MyCompany,
                           L=MyCity, S=MyState,
                           C=MyCountry"/>
        <Properties>
          <DisplayName>MyApp</DisplayName>
          <PublisherDisplayName>MyCompany
```

[13] https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/schema-root

```
                        </PublisherDisplayName>
    <Logo>images\icon.png</Logo>
  </Properties>
  <Dependencies>
    <TargetDeviceFamily
         Name="Windows.Desktop"
         MinVersion="10.0.17763.0"
         MaxVersionTested="10.0.18335.0" />
  </Dependencies>
  <Capabilities>
    <rescap:Capability Name="runFullTrust" />
  </Capabilities>
  <Resources>
    <Resource Language="en-US" />
  </Resources>
  <Applications>
    <Application
         EntryPoint="Windows.FullTrustApplication"
         Executable="myapp.exe" Id="myapp.exe">
      <uap:VisualElements
         BackgroundColor="transparent"
         DisplayName="My App"
         Description="A useful description."
         Logo="images\icon.png"
         SmallLogo="images\small_icon.png"
         ForegroundText="dark"
BackgroundColor="#FFFFFF" >
      <SplashScreen Image="images\splash.png" />
      </VisualElements>




      <Extensions>
           <uap:Extension
```

```
                  Category=
                    "windows.fileTypeAssociation"
                  EntryPoint=
                    "Windows.FullTrustApplication"
                  Executable="myapp.exe">
                    <uap:FileTypeAssociation
                         Name=".foo">
                       <uap:EditFlags
                         AlwaysUnsafe="false"
                         OpenIsSafe="false"/>
                         <uap:SupportedFileTypes>

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

Package Files and VFS

Package files may exist either under the root of the package (next to the AppXManifest.xml file referenced earlier) as well as in numerous subfolders that may be created, or under a special subfolder named VFS or Virtual File System.

We will be covering both of these notions within the current section.

Package “Root” folder

Although you may be tempted to put files under the root folder as it gives you the impression that it mimics the Win32 style of installing into the Program Files folder, in practice all files will be placed under the “Program Files\WindowsApps'” folder. VFS files are just written to VFS subfolders, which allow for layer merging opportunities with native files and other packages. .

For example:, when the package contains configuration files that an Enterprise IT Pro might want to override by using a Modification Package with updated versions of those files. This scenario will require the developer to use VFS pathing, at least for those files.

Files under the package root folder (and subfolders other than VFS) are normally not writable without using the Package Support Framework to include additional redirection capabilities. However, on OS 2004 and later, there is a new manifest entry in the UAP10 schema called “**InstalledLocationVirtualization”**. When specified, this manifest instructs the runtime to perform the same action that the PSF FileRedirectionFixup would have taken to allow writes.

Virtual File System (VFS)

For compatibility purposes, when repackaging traditional apps, we often prefer to place the files under the VFS tree.

Inside this VFS folder, there may be additional specially named folders that represent original system locations. It might be convenient to think of them as variables similar to the %Home% environment variable, or KnownFolderIds.

The VFS allows the traditional Win32 application to either make requests to the actual installed location, or (when inside the VFS folder) allow the application to access its own resources without requiring any code changes. When it comes to application files access, the redirection kicks in automatically when the application is trying to access files from the equivalent location on the end-user's system. The support provided may vary depending on the nature of the file operation, and the location of the file.

When the application running inside the container attempts to find or read a file under the known path, the system will automatically check the redirected package VFS location first, and then (if necessary) check the location requested by the application.

While Read operations should work without any problems (except for AppData and LocalAppData, as noted below), Write and other operations will lead to many application failures. This is expected unless Package Support Framework fixups are applied to the program to solve the issue.

Please note that Read, Write, and other operations to files and folders that are not part of the package, or VFS mapping, are not controlled by the container and are allowed as long as the user has permission to perform the operation requested. For example, as long as the package does not have a VFS\Documents folder in it, any attempt to Read or Write files to the user's Documents folder will be handled externally and those files will be automatically visible to other applications.

These details of VFS support may vary depending on the folder, and possibly the OS version. Many (but not all) of the VFS folders supported are documented by Microsoft here[14].
Be aware that this document hasn't been updated since the previous Desktop Bridge technology, so some details are no longer applicable.

Developers with access to source code should consider that it is highly recommended that you use API calls for retrieving the location of standard folders rather than “hard-coding” paths in your software. Both Environment variables and KnownFolderID APIs are available for this purpose, and will help make the application more portable. For example, for AppData folder paths, you might use something like this:

```
//local app data
string localPath = Environment.GetFolderPath
      (Environment.SpecialFolder.LocalApplicationData);

//roaming app data
string roamingPath = Environment.GetFolderPath
       (Environment.SpecialFolder.ApplicationData);
```

The most important VFS folders are listed here:

ProgramFilesX86 and ProgramFilesX64

On a 64-bit Windows 10 system, the known paths:

[14] https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes

```
C:\Program Files\
C:\Program Files (x86)\
```

Will map to paths looking like these:

```
C:\Program Files\WindowsApps\<AppName>\
   VFS\ProgramFilesX64\
C:\ProgramFiles\WindowsApps\<AppName>\
   VFS\ProgramFilesX86\
```

Documents

Maps to the users Documents folder.

Windows

Maps from `C:\Windows`

Although some sub-folders under this system folder have separate mappings, this VFS reference is used for all other references. For example, DotNet components added to the global assembly cache appear under the VFS\Windows\Assembly folder.

Fonts

Maps from `C:\Windows\Fonts`

Systemx86

Maps from `C:\Windows\System32` on 32 bit systems.
Maps from `C:\Windows\SystemWow64` on 64-bit systems.

Systemx64

Maps from `C:\Windows\System32` on 64-bit systems.

AppData, LocalAppData, and CommonAppData

AppData maps from C:\Users\[username]\AppData\Roaming
LocalAppData maps from C:\Users\[username]\AppData\Local
CommonAppData maps from C:\ProgramData

When constructing the package, one has to consider these common locations where settings and data are commonly placed. If the package is constructed without one of these VFS folders referenced, then (similar to the Documents folder example), all Reads and Writes will work just as the original Win32 app.

Typically, we want to pre-configure repackaged applications with settings, or include required data files, as part of the package. When we use these three VFS folders, the runtime support generally has special limitations, possibly depending on the folder and on the OS runtime version:

- **AppData** - Prior to the 1903 Runtime, files in the VFS\AppData folder of the package are not visible to the application by VFS redirection. In these versions, attempts to write to this location would be treated the same as for a non-containerized application.

  When the VFS\AppData folder is present, any attempt to write to a file under the user's Appdata\Roaming folder, whether to overwrite an existing file or create a new one will fail.

  Starting with the 1903 Runtime, VFS\AppData files and folders now redirect for Read and Write purposes. This Write redirection is performed by the OS using the same location as the one used by the Package Support Framework.

If your package might be used on down-rev systems, it may be safer to use the PackageSupportFramework in the package to ensure correct operation on any OS version.

- **LocalAppData** - On all versions (currently), files in the VFS\LocalAppData folder of the package are not visible to the application by VFS redirection alone.

  The Package Support Framework may be added to make this area Read and Write capable.

- **CommonAppData** - On all versions (currently), files in the VFS\CommonlAppData folder of the package are not visible to the application by VFS redirection alone.

  The Package Support Framework may be added to make this area Read and Write capable.

AppVPackageRoot

Maps from C:\

This acts as a catch-all for mappings that do not have another VFS mapping. Note that technically, the mapping is to the root folder of the drive letter that Windows is installed on, which is normally the C: drive.

Registry

The MSIX container for Win32 applications brings the App-V registry style of containerization support to the mainstream. This helps to solve some issues, like registry bloat and application interference with other applications, but isolation causes different issues that you might have to deal with as well.

The Windows Registry consists of a number of registry hives. Most technical people are already familiar with the HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER hives of the registry. As a user logs into a Windows desktop, their personal user hive is loaded into the logon session as HKEY_CURRENT_USER. MSIX containerized applications use a new type of hive known as the Application hive. The Application hive is only loaded within the container processes when the modern app is started.

The application hive includes two major branches, MACHINE and USER. Similar to how the VFS allows the runtime to layer certain package folders over the local file system, these two application hive keys act as an overlay to HKEY_LOCAL_MACHINE and HKEY_CURRENT_USER.

Having all these registry entries stored separately for each app means the end-user will get shorter boot and logon times, as Windows doesn't have to load the registry entries for all the apps. The application hive becomes mounted only when the container is started.

For more information on how a package's file and registry items work under MSIX, visit Microsoft's documentation[15].

[15] https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes

Registry Format in the Package

The MSIX package format stores the containerized registry of the package in standard form of the registry “dat” format. An MSIX package can have up to three .dat files.

- **Registry.dat**
- **User.dat**
- **User.Classes.dat**

Note. Pure UWP apps do not contain any registry .dat files in their packages. All of the registry entries for your package for both HKLM and HKCU will be stored in the **Registry.dat** file.

The **User.dat** file (when present) will contain a copy of the HKCU portion, and the **UserClasses.dat** file (when present) will contain a copy of the HKCU/Software/Classes area. It's a common assumption that these two extra files are used to boost performance for tools or components that only need to view those portions of the package registry. But, our experience has shown that the User.dat and User.Classes.Dat files only appear in the package when there are registry items in the package for the HKCU/Software/Classes area.

The .dat file format is a native registry file form, and if you need to view it, you can use a tool like RegEdit to import/export .dat files onto an existing key. The file is expected to contain a root key called “REGISTRY”, with sub keys for MACHINE and USER as needed for the equivalent to HKLM and HKCU on the native system.

Traditional Apps andApplication Registry Hive: Possible Issues

When delivered, the .dat files within the package are immutable, meaning that the registry values themselves can never be changed when a user runs the application. Instead, similar to App-V changes attempted directly to the application hive might be redirected to another safe location. Originally this redirection was not supported by App-V, but redirection for registry writes for many “Current User” registry items was added to the 2004 OS runtimes. The Package Support Framework may be able to support additional cases not supported by the runtime itself.

We can't disregard the possibility of human error when including logic or appropriately handling error returns to registry calls in some scenarios. Some programmers may have never encountered the kinds of issues which can occur when running inside a container and could lead to the programming logic request to be reinstalled, for memory corruption to occur, or the application to crash.

There are no hard rules as to when an application will have an issue in a registry request. But predominantly, it usually depends on the following aspects:

- **How the application makes the request**. When the application makes a request to operate on a registry key or item, it includes a request for the needed permissions. For example, those permissions can include Read, Write, and Delete permissions, or owner and ACL information. One of the most common issues we see happening with MSIX is related to applications that ask for “full access in registry calls". While this was always allowed (at least for HKCU), it is not under MSIX. In particular, Deletion, ACL and Ownership permissions are disallowed by the MSIX runtime.

- **The part of the registry the request is made for.** HKLM and HKCU requests with the same access permissions may behave differently under the MSIX runtime.
- **The version of the OS the app is running on.** Microsoft can change the MSIX runtime behavior from one version of the OS to another. For example, the restrictions were lessened slightly in the 2004 version of the OS.

If the application source code is not available for modification to remediate the situation, the Package Support Framework has a registry fixup that may solve the issue by performing changes to the API calls made by the application.

When permitted, all Writes to the application hive are copy-on-written to a private per-user, per-app location, the same way AppData is handled.

This allows the system to easily delete all the registry entries created by the app during its lifetime. This process happens upon uninstallation and avoids the all-known registry rot that has been slowing down Windows machines for decades.

All of these hives are virtually merged at run-time with the registry found on the OS, allowing the app to “see” the entire registry as a singular resource. This dynamic merging permits the application to view the modified package hive layered on top of the package hive, which in turn is layered on top of the system hives.

A note for those familiar with the App-V virtual registry. The App-V virtual registry works in a similar functional way, except for two aspects:

1. Under App-V, registry keys can contain a marker to indicate if layering should merge with the system hives,

or override (hide) the equivalent system key. The override method was used to hide keys and values of a potentially natively installed version of the same application. MSIX always uses the merge method.

2. Under App-V, registry keys and items can have deletion markers, indicating that the item should not exist as far as the app is concerned, even if it is present on a lower layer.MSIX (at this time) does not support any form of deletion markers.

Replacement of Registry Strings Containing File Paths

Typically, when a repackaging tool is used to generate an MSIX package from a traditional installer, the tool will automatically generate the .dat files for you. When these tools perform this action, they normally look for file string values that might be stored in the registry item values, either in Reg_SZ, Multistring, or Expandable string/multistring types. These file paths are usually altered to use file pathing relative to the package root folder.

As a case in point, the application might store the location of an important component upon installation, and then read that value to know where to look for it. On an x64 system, the file itself may have been written to “C:\Program Files” and is captured in the package under the “VFS\Program Files” folder. So, if the original registry value started with “C:\Program Files” , the reference string would change the beginning of the string to “VFS\ProgramFilesX64”.

Developers creating packages may use the same technique to help with portability.

Debugging the Containerized Registry

The isolation from the system's regular hives provided by the use of Application hives means that you'll have fewer problems with packages interfering with each other. However, in some cases, this isolation creates an impediment for inter-app communication through the registry. And, it also changes the way you debug your registry.

As an illustration, a ProcessMonitor trace will show registry references in different forms, depending on how the application determines the registry path. When the application starts with a hard coded base path in the application, you might see the ProcMon trace events using the "native path" (which gets redirected to the registry hive), but when the application uses a return path (taken from the registry) to form a new registry request, the path can appear in the ProcMon trace in the following form: `"\REGISTRY\WC\Silo{id}user_sid\Software…"` You can interpret this as an instance of the application hive for a current user path.

The Package Support Framework Trace fixup may also be added to the package to allow tracing of an exact Windows API call. This trace always shows the call as made by the application.

Either form of tracing can be used to determine when registry calls are failing, as well as remediation through application source code changes or other Package Support Framework based fixups.

It is often convenient to access the applications view of the registry using a tool like RegEdit. Accessing the registry using tools externally from the package container provides no access to the application hive of a package. These resources are only visible in the app's container, so, you will need to launch your registry editor or command prompt directly into the container. If you enable "developer mode" on the OS, you can do this with a simple PowerShell command:

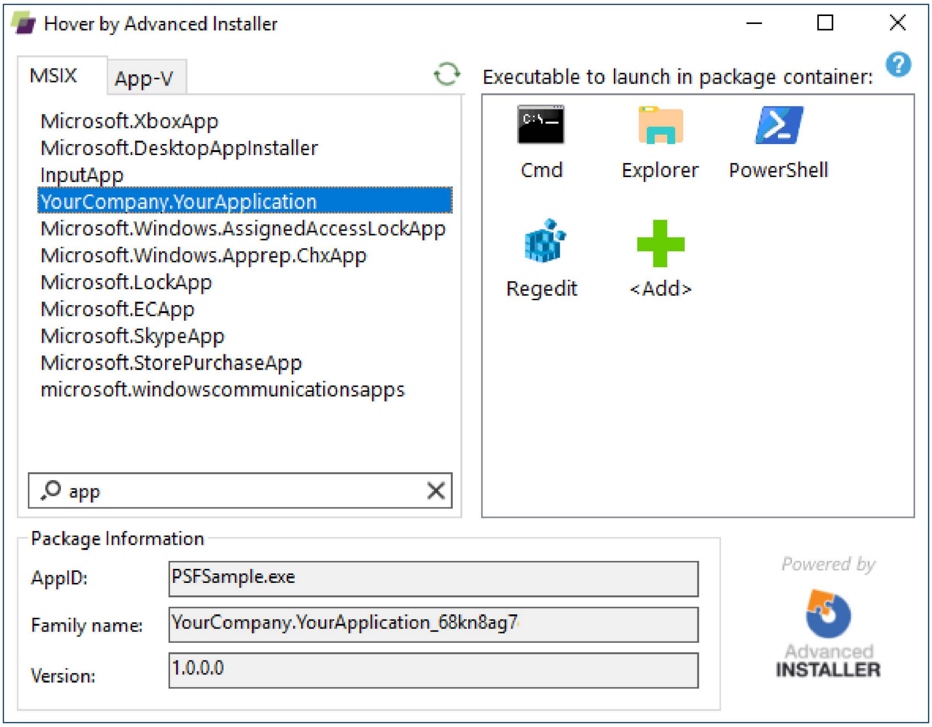
```
Invoke-CommandInDesktopPackage
```

Advanced Installer - MSIX PowerShell cmdlets[16] provides some other MSIX PowerShell commandlets that you might find useful in testing or debugging situations.

You can also use the free Hover[17] application built by the Advanced Installer team to speed up the process of running commands inside the package container environment.

[16] https://www.advancedinstaller.com/msix-powershell-cmdlets.html
[17] http://advancedinstaller.com/hover

Download it from: Advanced Installer - Hover[18]



Microsoft.XboxApp
Microsoft.DesktopAppInstaller
InputApp
YourCompany.YourApplication
Microsoft.Windows.AssignedAccessLockApp
Microsoft.Windows.Apprep.ChxApp
Microsoft.LockApp
Microsoft.ECApp
Microsoft.SkypeApp
Microsoft.StorePurchaseApp
microsoft.windowscommunicationsapps

app

Package Information

AppID:	PSFSample.exe
Family name:	YourCompany.YourApplication_68kn8ag7
Version:	1.0.0.0

Hover GUI

OS Integration

As mentioned in the beginning of this chapter, the registry entries of an MSIX-packaged app are visible only inside its container - which means

[18] http://advancedinstaller.com/hover

that other apps, including the OS cannot “see” the registry. Naturally, this characteristic changes some packaging habits.

Registry entries to define resources like FTAs, update PATH environment variables, or add COM components, are ignored. All of these resources must be declared in the Package Manifest for them to be integrated into the OS when the application is installed. Most repackaging tools will migrate these settings for you.

See section “AppxManifest.xml aka the Package Manifest” for more details.

Package Deployment Install Locations

By default, all MSIX packages are installed/extracted by the OS in the following folder:

```
%ProgramFiles%\WindowsApps
```

This is a system location, inaccessible by default, from Windows Explorer. Although there are methods to make it visible, we will not be going over them in this book.

Inside that folder, you will find subfolders for each app installed on the machine, including the OS built-in apps. The folder names here are not necessarily related to the MSIX filename, but are derived from fields within the package manifest files. All folders have their name following this pattern:

```
PublisherName.AppName_AppVersion_architecture_hash
```

Here’s an example of a folder name (Note: there are two underscore symbols "_" before the hash):

```
Microsoft.WindowsCalculator_10.1806.1821.0__x64__8wekyb3d8bbwe
```

The extracted MSIX package is inside the folder, just as you would see if you extract it with 7-Zip or other similar tools.

Only the OS can write in this location when installing your app. If your app is writing log files or other data inside the installation folder, it may crash or behave defectively. You need to either update your code to write outside of the package boundaries (and to %AppData% on OS 1903 and above) or fix it using the Package Support Framework for other OS versions and in the situations when you don’t have access to the code.

Appx Volumes

The installation path mentioned above is the default appx volume present on every Windows 10 machine.

You can define multiple volumes, move apps between them, or completely move all apps from a volume (such as the default one) to a new one, on a different drive.

The appx volumes can be managed using these dedicated PowerShell commandlets:

```
> Add-AppxVolume
> Remove-AppxVolume
> Dismount-AppxVolume
> Get-AppxVolume
> Mount-AppxVolume
```

You can find more details on the MSIX PowerShell commandlets at Advanced Installer - MSIX Powershell cmdlets[19].

MSIX App Attach

With the introduction of Windows Virtual Desktop (WVD), Microsoft also introduced the MSIX App Attach, a delivery engine of desktop applications for virtual environments.

The underlying technology supporting this engine is similar to the one used by the FSLogix solutions. It relies on filter drivers to deliver the MSIX package from a VHD.

The install location for apps deployed through App Attach uses the new VHD as a source. Launching the app on the virtual desktop requires one additional step. This step, executed every time the user logs in, will register the app's AppXManifest on the machine's local default appx volume, %ProgramFiles%\WindowsApps. At log off, all apps published through App Attach are automatically deregistered.
So in the end, you get a hybrid install path, with the application binaries present on the VHD (which is usually stored on a shared-storage server) but with the AppxManifest found on the default appx volume.

[19] https://www.advancedinstaller.com/msix-powershell-cmdlets.html

Applications delivered through App Attach promise great improvements for VDI-based infrastructures, but currently with a few limitations - not all MSIX packages can be deployed using App Attach.

The App Attach technology is less than a year old, it hasn't yet matured and currently lacks sufficient management. We'll come back with more details on its capabilities and limitations in a future edition of this book.

You can read more about Microsoft's introduction for MSIX App Attach here at Microsoft Documentation - Azure | Virtual Desktop | What is AppAttach[20] and within this book we will cover the topic in more detail within the "VDI Meets MSIX with App Attach" section.

Uninstall and Cleanup: MSI vs. MSIX

Most of the time, when you uninstall an MSI package the application files from AppData and the registry entries created by the applications during its lifetime are left on the machine, polluting the system with garbage. The famously called "Windows rot". This makes your PC slower every time you install an application.

MSIX packages, on the other hand, reduce machine clutter by simplifying the install and uninstall process.

Due to its containerized model, uninstalling an MSIX package will remove any files that the app has created while running (only the files redirected to its %LocalAppData%) by default. This includes all the application files installed under %ProgramFiles%WindowsApps, and any installation activities performed elsewhere. Additionally, any redirected

[20] https://docs.microsoft.com/en-us/azure/virtual-desktop/what-is-app-attach

registry writes to the application hive are removed. This helps to keep a clean machine and avoid the Windows rot.

Keep in mind that if the application creates files in other locations on the machine (*not recommended, but possible if it has the right permissions*), those files will not be deleted upon uninstallation. For example, a file saved in the Downloads or Desktop folder.

Generally, an update of a package will remove the previous version (if no other user on the system has it installed), but package data will be retained in the newer version.

The MSIX PowerShell commandlets provide you with options to automate your uninstalls:

```
        > Remove-AppPackage <package full name,
    possibly using wildcards>
```

Microsoft made two types of changes in OS 2004 that now can impact the removal of package-related data. These are new as the book is being written, so while we do not have experience in using them in production, we just want to mention them:

- There are now options on the cmdlet to prevent removal of package related data. More details on the MSIX PowerShell commandlets can be found at Advanced Installer - MSIX PowerShell cmdlets[21]

[21] https://www.advancedinstaller.com/msix-powershell-cmdlets.html

- There is a new option in the package manifest (UAP10 schema) called UpdateActions that appears to be related to the upgrade scenario but might also apply during an uninstallation.

Uninstalling Modification Packages

Important! When you uninstall an MSIX package, its related modification packages will be removed automatically.

As explained in the MSIX Modification Packages chapter, these are independently managed packages with regards to their deployment.

Make sure your deployment team receives correct and clear instructions for removing these packages.

Microsoft Documentation and Community Sites

If it is only one link you learn one from this book, it should be this one: "http://aka.ms/msix". This is Microsoft's landing page for all things related to MSIX.

From this page, Microsoft provides access to documentation, downloads, tutorials, videos, and the MSIX community known as "Tech Community" (the replacement of the old TechNet forums), and more.

Microsoft Documentation

Microsoft hosts its documentation on GitHub, and to help keep it up to date, they allow you to even mark up pages and make pull requests (the submission process used in git to request adding your changes into the documentation).

As it is typical with Microsoft Documentation, the Microsoft documentation for MSIX is filled with high level concepts, and low level details, but is commonly a little light on providing details to get the full picture.

This book focuses on some of the missing information, especially on helping you understand when and why you might use some of the information found in the detail-level document. It also explores the non-Microsoft options available out there.

MSIX Tech Community

The Microsoft Tech Community Site[22] hosts hundreds of new community hubs created by Microsoft. Once you log in, you may register to the MSIX community. Then, go to Microsoft Tech Community - MSIX[23] to have direct access to the MSIX-based Conversations and Ideas (forum posts), along with Blogs posted by the MSIX development team.

This is a highly active community, and currently the Microsoft development team is heavily engaged in the conversations and ideas. If you need help, or just want to learn from others who may have faced a

[22] https://techcommunity.microsoft.com/
[23] https://techcommunity.microsoft.com/t5/msix/ct-p/MSIX

similar problem to the ones you're facing today, you'll want to check it out.

Industry Blogs & Resources

Tim Mangan's Blog - Confessions of a Guru – Blogs from TMurgent[24].

MSIX Report Cards, by Tim Mangan - A series of research papers examining how MSIX is doing at various points in time - MSIX Report Cards[25].

Advanced Installer's Blog - Blog | The MSI(X) Experts Crib[26].

MSIX Ready Apps - A collection of apps evaluated by the Advanced Installer team - MSIX Ready[27].

Pascal Berger's Blog - MSIX Archives[28].

Social Media

On various social media sites, the hashtag #MSIX is used by folks sharing news and information about MSIX. At this time, Twitter seems to be the most popular for MSIX social media resources.

[24] http://www.tmurgent.com/TMblog/
[25] https://www.tmurgent.com/appv/en/resources/report-cards
[26] https://www.advancedinstaller.com/blog/msix-page-1.html
[27] https://www.advancedinstaller.com/msix-ready.html
[28] https://www.wpninjas.ch/category/msix/

OS Version vs MSIX Functionality

The public rollout of MSIX started with Windows 1709. Ever since, Microsoft has continuously improved the platform with new and updated features.

To get an always-up-to-date reference of all the features listed in this book, you can bookmark the following URL on your browser:

“advancedinstaller.com/msix-support-windows”

This online list will be constantly updated by the Advanced Installer team.

By the time you read the first edition of this book, most Windows 10 enterprise users should be running Windows 10 1803 (*considering Microsoft’s policy of supporting each update for only 18 months*).

Assuming you are running 1803, we will skip the MSIX features available by default in this version and only mention a few of the most important features that require a newer Windows 10 version. (*latest update: June 2020*)

Modification Packages	1809	

Windows 10 S	1809	*As a target device for deploying apps in your infrastructure.*
MSIX Bundles	1809	
“allowElevation”	1809	*Works if the Win32 app has the execution level set to “requireAdministrator“.*
“ForceUpdateFromAnyVersion”	1809	*Enables downgrade scenarios.*
Protocol Handlers	1909	

Shell Extensions	1909	Support for Context Menu shell extensions. Other shell extensions *might* work.
NT Services	2004	*The services actually run outside the MSIX container.*
User Registry Changes	2004	Support for user modification of HKCU items in the package. Only available if the package has a User.Reg file.
Fonts	2004	Support for fonts in the package. The fonts remain contained within the package but are made available both inside and outside the container.

MSIX Tooling for IT Pros & Developers

We're always looking for ways to simplify and smarten our job - and tools allow us to achieve that. As Microsoft MSIX Packaging Tool is the preferred tool for IT Pros that want to repackage into MSIX technology, we think it's important we go through it extensively, touching on the specific options and capabilities it brings.

Often, the Microsoft MSIX Packaging Tool is used in combination with a third-party tool, Tim Mangan’s PsfTooling, so we will be covering them both next.

Microsoft MSIX Packaging Tool

Microsoft offers a first-party tool to help IT Pros repackage existing applications into MSIX packages by capturing the installation. The Microsoft MSIX Packaging Tool (MMPT) offers a basic level of capabilities that is sufficient for many situations. However, they leave plenty of room for their tooling partners to be able to create tools that fill in the gaps, are easier to use, or address more sophisticated scenarios in your environment.

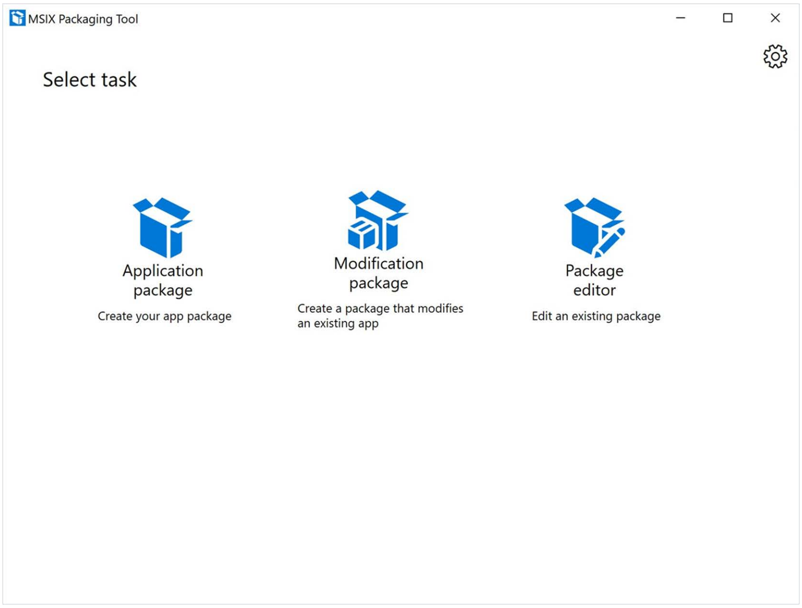
The MMPT[29] is free to download from the Microsoft Store, and currently gets updated around 4 times a year. Additionally, there is a special "Insiders Program" for this tool that offers preview builds in-between.

For recapturing purposes, the MMPT should be used on a clean virtual machine. It may also be installed on a secondary machine for controlling remote packaging. At this time, the tool provides support to identify and use a VM, but it does not manage the VM and snapshots itself.

MMPT supports the following three scenarios:

1. **Creating a new package by capturing an installation.** This uses a wizard to walk you through the process of entering information and performing the capture.
2. **Creating a modification package to another package**. This wizard requires you to have the primary package available.
3. **Package Editing.** This supports simple editing of file and registry entries from within the tool, and the ability to edit the AppXManifest file in an external editor of your choice.

[29] https://www.microsoft.com/en-us/p/msix-packaging-tool/9n5lw3jbcxkf



The MMPT is constantly adding new capabilities, so we won't go into details on what is supported, but right now, it requires full repackaging as there is no supported application upgrade scenario that involves an installation recapture mode.

The MMPT also includes the capability to convert existing Microsoft App-V 5 packages into MSIX without recapture -- this action does not require a clean VM.

This is the primary landing page[30] of the tool located on the Microsoft Documentation.

[30] https://docs.microsoft.com/en-us/windows/msix/packaging-tool/tool-overview

When comparing the MMPT against other packaging options, you should note that the MMPT does not directly support the inclusion of the Package Support Framework. It is easy to miss when reading the documentation as the PSF simply isn't referenced. Including the PSF when using this tool for packaging is left as a manual task to perform during the application installation, as well as for repackaging scenarios.

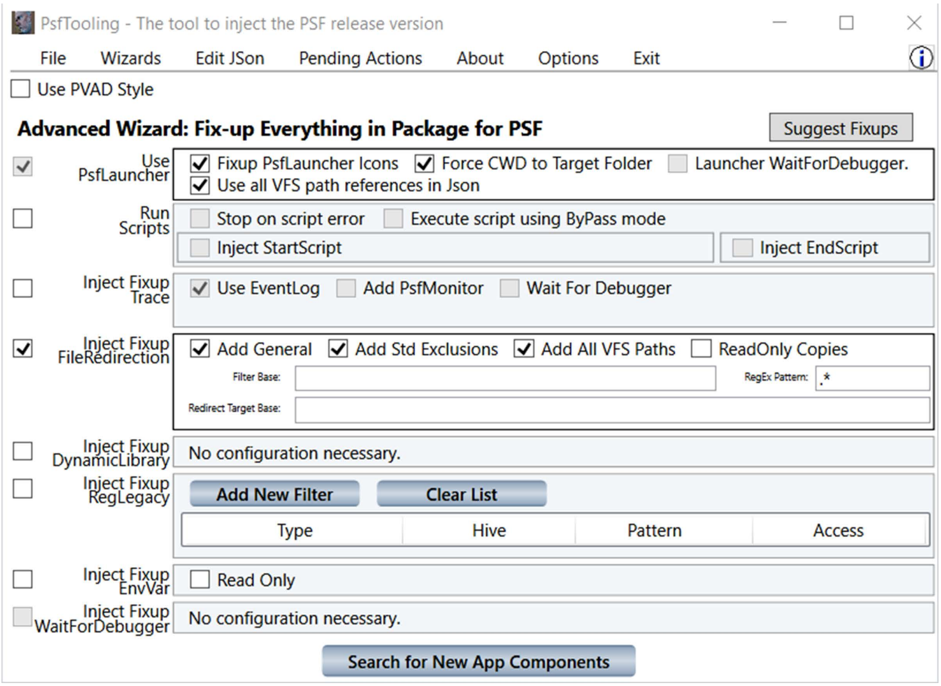
Note. The PsfTooling is typically the tool of choice to fix the manual tasks.

The current version of the MMPT is available for download here[31].

PsfTooling

Most likely, if you use the MMPT, you would need to use an additional third-party tool called PsfTooling developed by Tim Mangan to inject and configure the PSF. It allows to add and configure the PSF while still in the capture mode of the MMPT. This tool is free and available for download from the Microsoft Store.

[31] https://www.microsoft.com/en-us/p/msix-packaging-tool/9n5lw3jbcxkf



☐ Use PVAD Style

Advanced Wizard: Fix-up Everything in Package for PSF

✓ Use PsfLauncher: ✓ Fixup PsfLauncher Icons; ✓ Force CWD to Target Folder; ☐ Launcher WaitForDebugger.; ✓ Use all VFS path references in Json

☐ Run Scripts: ☐ Stop on script error; ☐ Execute script using ByPass mode; ☐ Inject StartScript; ☐ Inject EndScript

☐ Inject Fixup Trace: ✓ Use EventLog; ☐ Add PsfMonitor; ☐ Wait For Debugger

✓ Inject Fixup FileRedirection: ✓ Add General; ✓ Add Std Exclusions; ✓ Add All VFS Paths; ☐ ReadOnly Copies

Filter Base:

RegEx Pattern: .*

Redirect Target Base:

☐ Inject Fixup DynamicLibrary: No configuration necessary.

☐ Inject Fixup RegLegacy:

Type	Hive	Pattern	Access

☐ Inject Fixup EnvVar: ☐ Read Only

☐ Inject Fixup WaitForDebugger: No configuration necessary.

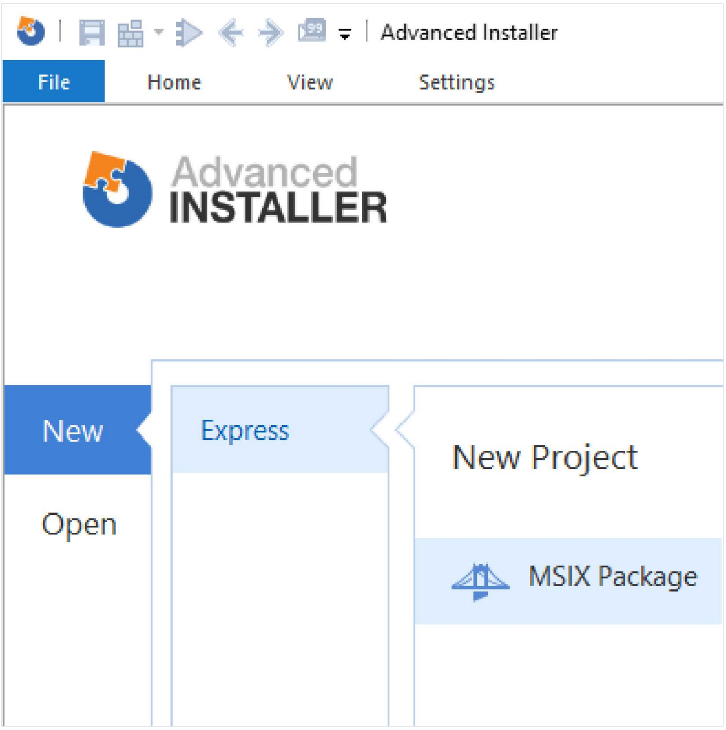
The tool is used while in the capture mode of the MMPT "Create New Package" wizard, after you have completed installing and configuring the target application. PsfTooling will make recommendations for PSF fixups that might be required for the application, add those components, and configure the PSF's json file for you. The tool will also modify the installed shortcuts and file associations to make the installed application compatible with the packaging tool features.

PsfTooling is available to download from the Microsoft Store here[32].

[32] https://www.microsoft.com/en-us/p/tmurgent-psftooling/9nc6k0q954jv

Advanced Installer Express

The Express edition from Advanced Installer was publicly announced in March 2018 by Kevin Gallo, Microsoft CVP, as the official release partner for MSIX. Until the release of the MSIX Packaging Tool, it was the only tool that supported MSIX.
Advanced Installer Express[33] is a free tool available to download from the Microsoft Store and gets constantly updated by the Advanced Installer team.



[33] https://www.microsoft.com/en-us/p/advanced-installer-express/9n4vqdj7ltb8

The powerful and easy to use GUI from Advanced Installer is the main characteristic of the Express edition. It helps create MSIX packages while educating the user. The tool is optimized to speed up MSIX adoption and brings considerable savings for enterprise packaging teams.

The Advanced Installer Express edition supports the following scenarios:

- Converting third-party packages to MSIX format.
- Creating MSIX packages from scratch (for your internal apps).
- Editing standalone MSIX packages using the advanced GUI.

In all of the above scenarios, you get the most powerful Package Support Framework integration on the market, including automatic fixup suggestions.

When you compare it with the MSIX Packaging Tool, the Express edition has practical capabilities that make MSIX packaging significantly easier.

- Automatic handling of command line (shortcut) arguments.
- Native interpretation of high-level constructs (FTAs, etc.) in the AppxManifest.
- Build-time validation of the package content for suitability.
- Project-based oriented workflow: reload, edit and rebuild your MSIX package in seconds.
- Package Support Framework integration, with automated fixes.

For advanced capabilities, the commercial Architect edition from Advanced Installer brings virtual machine management/integration, team repositories, and all the standard MSI/App-V packaging tooling. You can read more about it on Advanced Installer[34].

AdminStudio

AdminStudio is a traditional commercial repackaging tool that attempts to be a complete solution for the entire application lifecycle. As such, it is an extensive, expensive, and complex suite. Amongst other things, it supports repackaging of traditional installers using either inspection (when an MSI is available) or recapture.

Flexera has added support to the product for MSIX under the following categories:

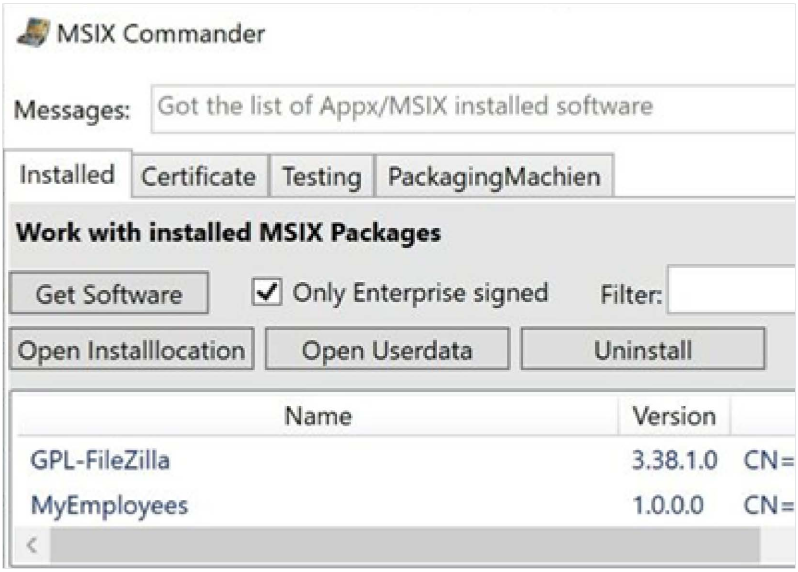
- Static Analysis of existing MSI installer for suitability.
- Transformation of captured output into MSI, App-V, or MSIX (and others).
- Manual inclusion of PSF based fixes.
- A stand-alone MSIX Package Editor.
- Integrations into test and distribution systems.

As products are constantly evolving, we recommend that you visit Flexera website[35] for updated information.

[34] https://advancedinstaller.com
[35] https://www.flexera.com/products/operations/application-management.html

MSIX Commander

MSIX Commander is a free tool built by Pascal Berger. This is a community tool that is used mainly in the package testing process.


MSIX Commander

Messages: Got the list of Appx/MSIX installed software

Name	Version	
GPL-FileZilla	3.38.1.0	CN=
MyEmployees	1.0.0.0	CN=

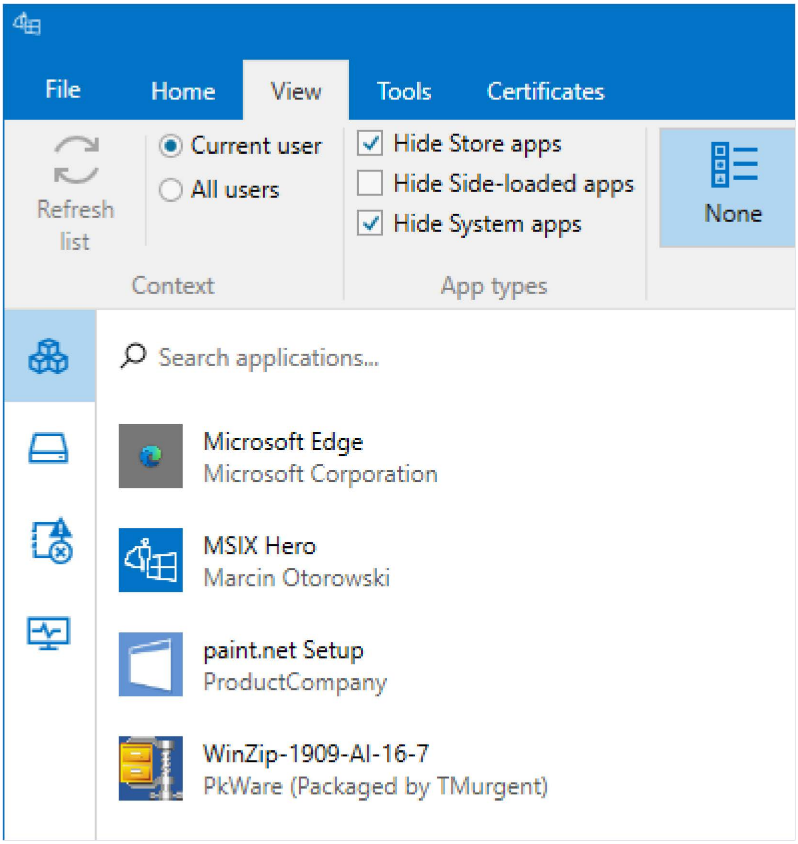
You may be surprised to learn that the tool you see in the image here is actually written in PowerShell. The tool source code and latest build as an MSIX package is available on GitHub - MSIX Commander[36].

The tool seems to be most useful during the smoke test phase when testing new packages. Using the tool, you have an easy button to access the hidden installed assets, such as the manifest or config.json files. And, you can also work with certificates, deployments, and simple edits and debugging.

[36] https://github.com/bergerpascal/MSIX_Commander

MSIX Hero

MSIX Hero is a free tool built by Marcin Otorowski. This is a community tool that is also mainly used in the package testing process.



Microsoft Edge
Microsoft Corporation

MSIX Hero
Marcin Otorowski

paint.net Setup
ProductCompany

WinZip-1909-AI-16-7
PkWare (Packaged by TMurgent)

Covering some of the same features as MSIX Commander, you would be safe to assume that this is also delivered as an MSIX package. The application does not seem to be open source. You can download the tool from MSIX Hero website[37].

[37] https://msixhero.net/get

The download is available in two forms -- one of them supporting the automatic detection of updates using the same web style we will probably see software vendors using in the future. Currently, the tool requires you to download msix.core as an additional download.

The tool seems to be most useful during the smoke test phase when testing new packages, but also has some simple package editing capabilities.

Additional MSIX Packaging Tools

There are many repackaging vendors out there that have announced or delivered MSIX support. Unfortunately, we can not cover all of them here, but we still tried to mention most of them here. These products usually have free trial versions for you to evaluate.

We previously covered Microsoft's first-party tool for packaging --the Microsoft MSIX Package Tool-- as well as tools from Advanced Installer and Flexera.

RayPack Studio

Based in Germany, RayNet[38] has been in the packaging business for quite a while.

[38] https://raynet.de/msix

Pace Suite

Pace[39], a division of Infopulse which is further owned by TietoEvry, provides conversion and repackaging tools.

InstallAware

Founded by a previous InstallShield alumni, InstallAware[40] has some MSIX Package editing capabilities, along with some other useful features..

Liquit Setup Commander

Liquit Setup Commander is a tool built by Liquit.com. It is available as a standalone product or part of the Liquit Release & Patch Management suite.

It enables you to prepare your own in-house applications for deployment by using either one of the connectors for Liquit Workspace, Micro Focus ZENworks, Ivanti EPM, WVD, and Microsoft Configuration Manager. Liquit also provides MSIX App Attach delivery via Liquit Workspace.

Additional MSIX Development Tools

The majority of the development vendors out there have announced or delivered MSIX support. Here are the most popular ones.

[39] https://pacesuite.com/convert-exe-to-msix
[40] https://www.installaware.com/

- Advanced Installer (Caphyon).
- InstallShield (Flexera).
- Wix Installer (FireGiant)
- InstallAware (InstallAware).

Additional MSIX Vendors

In this section, we will list some of the other vendors seen in the packaging space offering management, delivery, or support.

Management/Delivery

- Citrix
- VMWare
- AppVentiX
- Liquit

Support

- Access IT Automation
- Camwood
- Cloudhouse
- Rimo3
- SSH2

Fundamental Packaging Concepts

So far, we've covered the core concepts that we recommend you understand before you start building your first MSIX packages.

Going forward, we'll explore the options you have when application compatibility issues arise, how you can customize your enterprise deployments and what scripting options MSIX gives you.

Digital Signing

One of the main benefits of MSIX is its security-driven architecture. This is provided first by the app container and second, by the fact that each MSIX package carries a digital signature.

Over the last 20 years, no matter the packaging format (a simple zip, MSI, EXE, App-V, etc.), the use of digital signing was always considered an optional procedure by 99% of the packaging teams. This is different now.

Remember! All MSIX packages must be digitally signed. You will see that the OS has a setting called "Developer Mode" that allows for unsigned MSIX packages to be installed -- but, this setting is not intended for production use.

Why should you sign an MSIX package?

When deploying an MSIX package, the operating system uses the digital signature to confirm that it can trust the original owner or builder of the package. The digital signature is the standard method to ensure a package was not tampered with.

Digital signing involves attaching a hashed value of the contents of the package file to an “alternate stream” of the file to validate that the file contents have not changed since signing.

Note: When a file is digitally signed, applying the signature is always the last step.

Because the signing process uses the private key from your certificate as a secret for the encryption algorithm, nobody else can alter the file without invalidating the signature. Without your private key (and private key password), no one is able to re-sign the package file with your certificate.

If an attacker gets his hands on the package and adds a malicious file inside, it will break the signature. Without your certificate, the attacker cannot digitally sign the package and the system will automatically reject it.

This forces customers to manage which certificates they will support on their systems, a topic we will discuss shortly.

How to sign an MSIX package?

Generally, applying a digital signature is done by invoking a tool like Signtool.exe from the command line. In its command line, you can specify the files you want to sign, the certificate used for signing, and other configurations.

One important concept to understand is that Signtool reads a field called “Publisher” from the AppXManifest.xml file and compares it to the field called “Subject” in the certificate. The purpose of this comparison isn’t clear, but as we usually package many or all of our packages using the same certificate, this means that we must generate an AppXManifest file to match what is in the certificate.

Traditionally, this is in the form of a string like “CN=Company Name”. Microsoft’s own packages are signed using a certificate with a subject format resembling “CN=Microsoft Windows, O=Microsoft Corporation, L=Redmond, S=Washington, C=US”.

To make this process seamless, packaging tools similar to the Microsoft Packaging Tool and Advanced Installer, and most of the package authoring tools used by developers, provide a helpful GUI that enables package signing. In the end, they all abstract the command line options behind an easy to use GUI to save you time.

Configuring the signing tool is out of the scope of this chapter as this is documented in the user guide of your preferred packaging tool. What you need to understand is where to get your certificate from and how that affects your package. And we'll go into this now.

Where do I get a certificate from?

Windows recognizes certificates if it finds information about them in the system’s certificates manager (certmgr.exe).

You generally have two options to get a certificate to be recognized by the system.

1) You generate a self-signed certificate and pre-deploy it onto the machines where you want to install the application that you will sign. This certificate can be deployed by your IT team to ensure all the machines from your enterprise trust the packages signed with that certificate.

   In an enterprise environment this certificate creation and deployment may be controlled through PKI (Private Key Infrastructure). Sometimes, Group Policy is also used for the distribution.

2) You purchase a code signing certificate from a trusted Certificate Authority (CA) vendor, like Verisign, Comodo, Thawte, etc. Microsoft keeps an up-to-date list of the verified certificate vendors that are pre-deployed within the OS. Purchasing a certificate from one of these vendors guarantees that any Windows OS will see a valid digital signature applied to your package. This is how most software vendors digitally sign their applications (MSIs, EXEs, etc.).

   Unless you plan to distribute the package yourself, using a website, it is unlikely that you will need any level of "Extended Validation" when getting your certificate from a trusted vendor. So, the most economic option will suffice.

3) You can use the Azure Code Signing Service. Although it is a free service at this moment, you must have some kind of Azure tenant (including Office365) to be able to use it. Your Azure

administrator will need to enable the service and instruct you on how to use it. Certificate validation is also provided by Azure, but only in cases when the user is logged in with Azure credentials for your tenant.

How to protect the Certificate with a password

When you get your certificate, you will need to extract the signing format out to a file with a .pfx file extension. When you do this, it is very important to add a password to the file when performing the extraction. This ensures that the .pfx file cannot be used to sign any files without providing the required password.

The password itself should be kept safely; available only to those that need it and can be trusted. For example, you would typically not give the password to a third party packaging company that you work with. Instead, you should use a cross-signing technique to allow them to sign the package using an untrusted certificate, so that you can later re-sign it with a trusted one.

Certificate Timestamps

As you can probably imagine, certificates expire. While you can control the expiration date of certificates that you privately generate, those purchased from a public CA (Certificate Authorities) have a short life-span - typically ranging from 1 to 3 years, depending on the price.

Important! If you don't add a timestamp during the signing process, the package will be considered invalid when the certificate expires. This expiration would prevent any new installations of the package, although it shouldn't affect end-users that already have the package installed.

To prevent this, Signtool supports adding a timestamp to the signature. This involves having Signtool communicate with a public CA timestamp authority to validate the actual time of the signature, and storing that information in the file alternate stream with the certificate information.

This validates that the file contents have not changed since you signed it, and that the signing occurred while the certificate was still valid. This process requires the trust of a CA providing that timestamp, preventing you from just turning the clock back on the machine used to perform the signing.

With the timestamp applied, the package can be installed even after the certificate expires.

Currently, all public CAs that we have reviewed support using their timestamping service (although usually they include a warning about preventing Denial of Service attacks if you try signing too often).

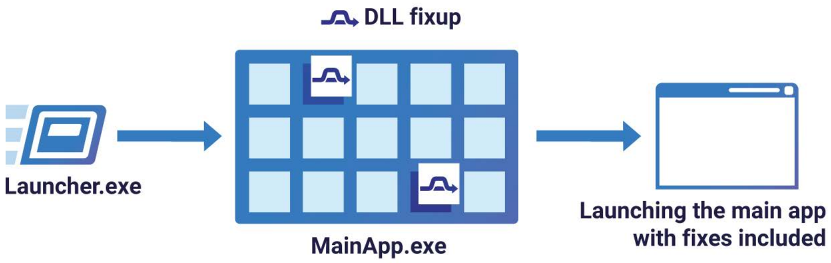
A common reference used is: http://timestamp.digicert.com.
Some other vendors will add “/rfc3161” to the end of their URL.

Package Support Framework

Running Win32/WPF applications inside a MSIX container is a different paradigm. While there are benefits, the container also can cause issues with applications when you simply repackage them without making changes to the application source.

To help with the MSIX transition, Microsoft has created the Package Support Framework (PSF) -- an open-source framework that enables you to apply runtime fixups for issues that might appear when migrating legacy applications.

Conceptually, the PSF works using fixups that are injected into applications to modify the application behavior, as shown in the following diagram:



Integrating the Package Support Framework

Inside the Package Support Framework (PSF)

The official Microsoft release for the Package Support Framework (PSF) is available on GitHub - MSIX Package Support Framework[41].

Under the hood, the Package Support Framework uses the Detours[42] technology, an open-source framework for handling API redirection and hooking. Aside from allowing you to move forward in instances when

[41] https://github.com/microsoft/MSIX-PackageSupportFramework
[42] https://www.microsoft.com/en-us/research/project/detours

you can't access the source code, this technology will also help make your application smarter.

With this framework, you can either choose to use predefined fixups or create a new fixup yourself. Since creating custom fixups is more of a developer matter, we will limit our focus on how to use the predefined fixups.

To start using PSF inside your MSIX package, you don't need the entire software that is in the GitHub repository for the PSF, as this includes additional content to help with testing the PSF itself.

These are the main resources that must be included into your MSIX package in order to use the PSF:

- **PsfLauncher** - *Necessary to use the PSF.*
- **The PSF runtimes** - *these should be included depending on your application's architecture*
- **A config JSON** - *specifies and configures the fixups used by your application*
- **The fixups DLL(s)** - *predefined or any custom ones you might build*

Commercial packaging tools like Advanced Installer, InstallShield, and RayPack provide direct integration with the Package Support Framework and take care of including the binaries listed above semi-automatically (you still need to perform configurations in their GUIs).

If you're using the Microsoft MSIX Packaging Tool, you will probably want to use a free third-party tool that we mentioned above called PsfTooling to provide a GUI tool to inject and configure pre-built PSF components.

Once you have defined the traditional app form, either by a recapture or by building an installation ingredient list, you would then add the appropriate PSF file to the package, followed by changing out a shortcut, and adding a configuration file that tells PSF what to do. Most of the tools mentioned above do this automatically.

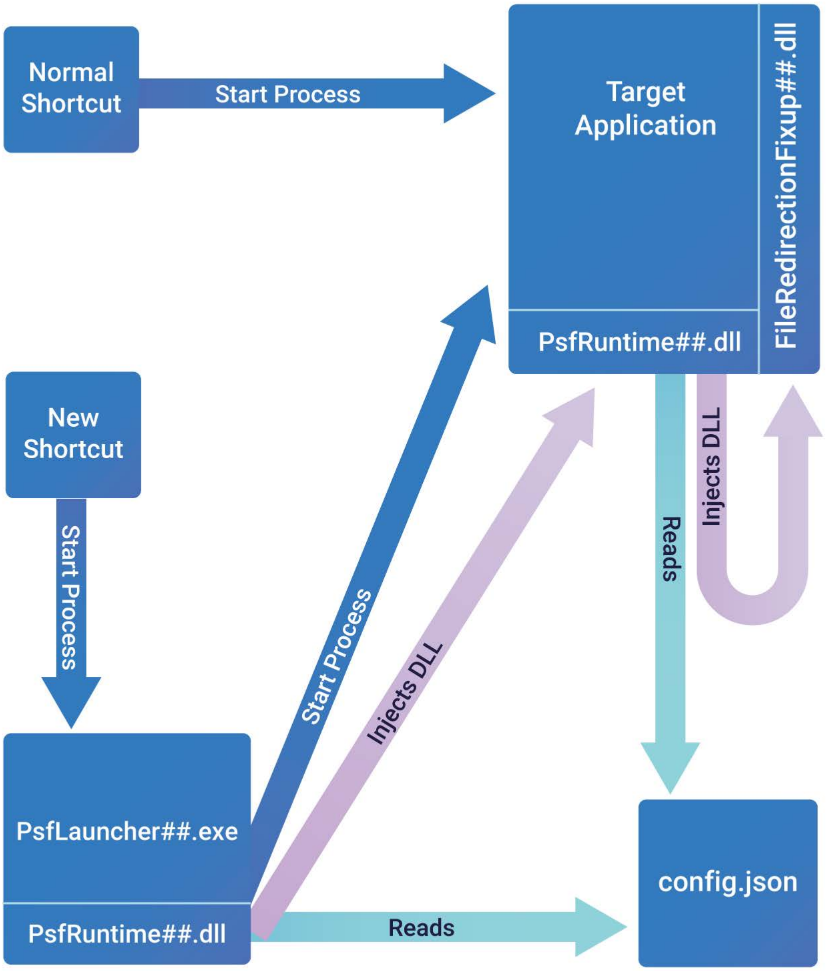
Using the PsfLauncher

In essence, you will need to replace the traditional shortcut .ink file added by the application installer with a new one. The old shortcut pointed to a target executable (possibly with command-line arguments, a working directory, and an icon). Your new shortcut will replace the target executable with a copy of the PSF launcher executable.

This PSF launcher can have any name you want. For example, the PsfTooling kit injects a file named like **PsfLauncher1.exe** while Advanced Installer injects a file called **AiStub.exe**. Sometimes, we need to use alternate when a package contains more than one shortcut as MSIX restricts any individual package from having two shortcuts to the same target file.

The configuration file (config.json) is used to inform PSF launcher what the real target application is and how to start it. The launcher starts the real target EXE and performs DLL injection for a fixup to be applied. This config file specifies what fixups should be injected into which target processes and also provides the behavioral configuration needed for the specific fixup. A utility DLL called “PsfRuntime” is used by both the launcher and injected into the target to make all of this work.

The startup sequence for this is depicted in the image below, where FileRedirectionFixup.dll is injected into the target executable process. For example, when an application is trying to write in its installation folder (an action forbidden by the MSIX standard), the runtime fixups redirect the call to a new location under AppData.



How Does Package Support Framework Work

In addition to launching the target application, the PsfLauncher may be configured in the Json to support command line arguments, the working directory, scripts, and PsfMonitor.

How to determine the needed fixups

The most difficult part of using the PSF is knowing when and what you need it for. Determining the fixups that your application requires can be a very time-consuming process.

Some of the third-party vendors include capabilities in their tooling to provide analysis of the traditional package before you start packaging for MSIX. Currently, none of these tools appear to provide a complete analysis, but they can provide useful information.

This means that you will have to invest in your own efforts to track your app's compatibility issues. And for the most part, that means creating packages and testing.

To help with that, you have a few options commonly used:

- Use the PSF tracing dll, PsfTraceFixup, inside your package to show what are the Windows API calls that your application makes.
    - This special fixup traces points that you might need to modify in application activities by using the other fixups.
    - The fixup supports two output forms.

        - One is the output to the debug console port -- and you need to use a tool like DebugView to capture and view the output.
        - The other form is to an event log -- and you need to use PsfMonitor (also part of the PSF) to capture and view the output.
        - These logs are also automatically parsed by Advanced Installer, simplifying the debugging process.
- Other commercial tools, like API Spy Studio, also target monitoring at the Windows API level. These tools generally have a wider coverage of the Windows API than the PsfMonitor has, however this extra detail may be either a blessing or a curse (as it could overwhelm you with information about API calls that the PSF fixups are not designed to fix).
- Process Monitor and ProcessExplorer (and similar).
    - Process Monitor shows the same file and registry captures as the API captures above, except that the capture is performed below the MSIX Runtime, which may have already made changes to the original request from the application.
    - Process Explorer is not a tracing utility, but it can be used to help understand command line arguments, process bitness, loaded modules, and kernel objects.

Depending on your needs, the pre-existing fixups can be added to your package, or you may have to create/code your own fixup.

Here are the most common types of issues, and how they are addressed via PSF are given here:

Issue Type	Issue Description(s)	Fixup Type
Shortcuts	Issues with supporting command line arguments, and specifying a working directory. Also, a package cannot have two shortcuts to the same .exe file (solvable by having separate copies of the launcher calling the same target .exe).	PsfLauncher
Scripts	Applications sometimes require modification either as part of the installation, or prior to use by the end-user. Generally, these modifications are based on the environment (Machine, OS, User, or other).	PsfLauncher
Missing Files	Although files in certain VFS folders may be present in your package, some like LocalAppData, will not be seen without a fixup.	FileRedirectionFixup
Writing to Package Files	The application cannot write to or modify files in the package without a fixup that makes a copy for the application.	FileRedirectionFixup
Dll Not Found	The application may have difficulty finding dlls that are part of the package.	DynamicLibraryFixup
Writing to	The application cannot write to or	RegLegacyFi

Registry	modify the machine portion of the registry. It may also have difficulty under the HKCU if the app asks for more permissions than it really needs.	xups
Missing Environment Variables	The application cannot see new or changed values to environment variables created or modified in the package.	EnvVarFixup

Predefined Fixups

In addition to the launcher, the Package Support Framework contains several predefined fixups described below.

ElectronFixup

Electron Apps is used to help apps originally developed as Electron Apps to operate in the MSIX container.

FileRedirectionFixup

The *File Redirection* is used to solve many file-related issues that could occur.

- The FileRedirectionFixup acts as an aid, allowing the containerized application process to have visibility of files that were captured under certain VFS folders.

  The MSIX runtime generally provides an overlay allowing to check the equivalent VFS path inside a package when an application requests a file of a certain path. Unfortunately, there are VFS folders in the package where this does not apply. Originally the AppData, LocalAppData, and CommonAppData folders were not mapped on this overlay until Microsoft added CommonAppData mapping.

  The FileRedirectionFixup can provide the additional missing mapping through the use of copy-on-access (described below).

- The FileRedirectionFixup allows the application to write to package files.

  MSIX protects the package files, preventing any corruption to the files: a feature sometimes referred to as "immutability". But, applications often need to write to files contained in the package, either to store settings or data.

  The FileRedirectionFixup can use copy-on-access to create a user private copy of the file and allow that copy to be written to.

- The FileRedirectionFixup implements a copy-on-access for package files and a redirection for access to the copied area. (Advanced Installer handles this automatically)

By default, the files are copied to a package's specific location in the user's LocalAppData\Packages folder, but this location can be specified using a mapped driver or server share location (specifying the user's AppData\Roaming folder is not possible due to MSIX limitations).

This allows the application to see areas of the package that are normally hidden, and to make file changes as needed.

Additionally, with the FileRedirectionFixup in use, any future attempt by the application to reference the file, will first detect the copied file as a new mapped layer on top of the packaged one.

DynamicLibraryFixup

Dynamic Library Loading - Used to make sure that the application can find dlls that are in the package.

The MSIX runtime does not implement two of the most important methods for an application installer to specify additional folders to find dll files:

- Modification to the system PATH environment variable.
- AppPaths registration.

The DynamicLibraryFixup allows for the registration of package dlls in the Json based configuration, letting the intercept to cause dll loading to always find the dll in the package.

RegLegacyFixups

RegLegacyFixups is intended to be a place for several types of fixups involving the application registry access. As it is relatively new, we expect that it will be updated over time. Currently, it has two types of fixup rules that may be applied as needed.

- *Registry Permissions* - Used to modify application registry requests to the permissions allowed under MSIX.

  Often an application might open a registry key or item with more permissions than it really needs. Especially for HKCU based items, this could be done in traditional applications, but not under MSIX due to immutability implementations in the MSIX runtime. This intercept rule can target certain types of SAM permission requests made when accessing the registry.

  Assuming the application was asking for permissions it did not need, the Registry Permissions solves the problem. If the app later attempts, in a subsequent call, to use those unavailable permissions on the object, the call will fail. But even so, this might still solve your problem as developers tend to specifically check for permissions when executing such requests, so they handle any eventual errors better.

- *Fake Delete* - Used to trick the application into thinking that a registry item or key has been deleted.

  We have found some apps that use the registry as a scratch pad for temporary storage, and then try to delete the entries as part of shutting down. As the MSIX runtime does not permit these deletions, using this tool to "lie" to the app could solve the issue.

EnvVarFixup

Environment Variables - Used to turn User Session or System scoped environment variables into Application scoped ones. This affects only processes running inside the container, so it does not help with things such as Path variable changes.

The fixup supports registering the environment variable name and the value inside the JSON configuration, or alternatively, registering the name in the JSON and the value in the package registry. Both methods allow processes inside the container to see the value, and the alternative method supports apps that want to update the value at runtime.

Applying the PSF

Although Microsoft offers the Package Support Framework as an open-source kit on its GitHub page, they do not provide any additional tools to make it easy to integrate it into your packaging and debugging workflow -- and you are left with cobbling up your own workflow using independent pieces, or resorting to third party tools.

Third party packaging tools tend to have a more integrated and holistic approach to simplifying and speeding up this process. Be aware that third party products may either use an older version of the PSF than the one available on GitHub, modified versions of prebuilt fixups, or could include additional kinds of fixups that are not available in the GitHub source.

Applying PSF for MSIX Packaging Tool

If you are using the MSIX Packaging Tool, the best way to leverage the Package Support Framework within your MSIX package (without having to do everything manually) is to use the free PsfTooling kit[43], built by Tim Mangan.

This tool can apply and configure all of the prebuilt fixups described previously, with the exception of the ElectronFixup. It also supports a standard fixup called *WaitForDebugger,* which is useful for developers that need to debug their code or for debugging the PSF itself.

Additionally the tool can detect and modify the installed application to improve compatibility by aligning the application component registration to methods supported by the MSIX Packaging Tool.

Applying PSF with Advanced Installer

Advanced Installer Express - the free, featured-limited edition of Advanced Installer - offers support for additional fixups and replacements for predefined fixups: these can be easily configured and integrated into your package by using the "Trace App" functionality.

The following equivalent replacements should be considered:

- Trace App
- WorkingDirectory

[43] https://www.tmurgent.com/appV/en/resources/tools-downloads/msix-tools

- Command-line arguments management
- Run custom PowerShell scripts

If you encounter any situations for which a fixup is not available, contact the Advanced Installer Support team over email at support@advancedinstaller.com. They will analyze the problem and try to provide a fixup for the community, if possible.

Advanced Installer Express edition has a built-in debugger that traces your app and tells you what known fixups are required. Check out this video demo at Advanced Installer - Package Support Framework[44].

Applying PSF with other Packaging Vendors

Flexera AdminStudio and RayNet RayPack Studio are tools that also provide an integrated approach.

Config.JSON and Config.XML

The Package Support Framework uses the Config.Json file inside the package in order to configure the Psflauncher and Fixup components.

The choice of using json inside the package appears to have nothing to do with traditional json usages. There are no web servers or rest interfaces in use. It is just a structured file with configuration items and values.

[44] http://www.advancedinstaller.com/package-support-framework

One important rule you must follow in json is that item names (not values) are always in CamelCase, which means it starts with a lowercase letter and uses an uppercase letter only when it would be a new word if spoken out loud (e.g. “workingDirectory”).

The config files make a heavy use of RegEx in the configuration values. RegEx is a pattern matching syntax, that instructs the reader to use a configured value string as a pattern to match against. The RegEx syntax is different from the pattern matching syntax used by the Windows command processor. If you need to further understand the RegEx syntax, you can start with the information available here[45] and look for the Ecml variation of RegEx.

In addition to the Json format, Microsoft defines an xml version of the file that may be used externally by tools responsible for producing the package. Those tools would transform the xml format into Json as part of the packaging process. To many, xml may be a better choice as it is self describing and makes it easier to implement validators against a schema. This is the main reason Microsoft specifically defined it to be used by these tools, even if the PSF doesn’t use this form itself.

The config file format consists of two major sections, **Applications** and **Processes**:

Config Applications Section

The Applications section is used to configure the launcher. The launcher is usually configured elsewhere to become the target of an entrypoint into the container. Generally, we consider this the

[45] https://en.wikipedia.org/wiki/Regular_expression

replacement target of a shortcut, but it can also be the target of other modified registrations in the AppXManifest, such as file type associations or URL protocol handlers.

The **Applications** section consists of an array of application items.

The identification of the application is performed by matching the name of the launcher process against the id field of the application item “id".

When using the Microsoft MSIX Packaging Tool with PsfTooling to add the PsfLauncher as defined on GitHub, the launcher uses an undocumented algorithm against the process name to perform this comparison. Generally speaking, it drops the “.exe”, turns it into uppercase, and changes any numeric character into the alphabetic English language equivalent (in Pascal Case). For example, the process name “PsfLauncher1.exe” would match the id “PSFLAUNCHEROne”.

Note. Other packaging tools may use a different algorithm in their copy of the PSF.

When there is a match found, the rest of the fields of the application item will be implemented by the launcher process.

This includes the following items:

- *monitor* - An item with subitems to specify a monitor program, like PsfMonitor, to launch first.
- *scripts* - An item with subitems to specify a start script to run before launching the target application, and an end script to run when the target application ends.

- *executable* - The package relative path to the ultimate target application file. This may be an .exe file, but any other file type that has a local file type association may be used.
- *workingDirectory* - A full or package relative folder to use as the working directory for the ultimate target application. If specified as an empty string, it will use the path containing the target application.
- *arguments* - Additional command line arguments to be appended to the command line when launching the ultimate target application.

Config Processes Section

The **Applications** section is used to configure PSF Fixups on a per-process basis. The section consists of an array of process entries, with two subitems:

- *executable* - The value is a RegEx pattern string to match against process names.
- *fixups* - The value is an array of fixups. Each fixup has additional subitems:
    - *dll* - The value of this item is the name (without path) of the dll fixup to be injected. If not found by that name, it will attempt to add a 32 or 64 (depending on the bitness of the running process) to the base name and try again. Thus “FileRedirectionFixup.dll” can match “FileRedirectionFixup64.dll”.
    - *Config* - The value is a collection of further subitems, dependent on the needs of the particular fixup.

The best way to begin to understand the syntax further might be to look at some example config files of existing packages.

Inspecting Fixups that are in a Package

Do you want to know if a package uses PSF fixups? Here are a couple of options to inspect it.

The first one is simple, extract the MSIX package using 7-Zip or a similar tool. In the package contents, you should notice the PSF-specific resources (psf launcher, dlls, & the config.json file).

Note. Opening up the package in a zip utility will break the package if saved, so remember to always use a copy of the package!

To understand the exact configuration of the fixups, if present, open the config.json file. Often, this file will be at the root folder of the package, but it could also be stored in the folder with the primary executable (eg. under VFS\ProgramFilesx64\VendorName).

Here is an example:

A Json file that causes the FileRedirectionFixup to be injected into the target process and configures the file redirection for all files with the extension ".log" will look like the one below.

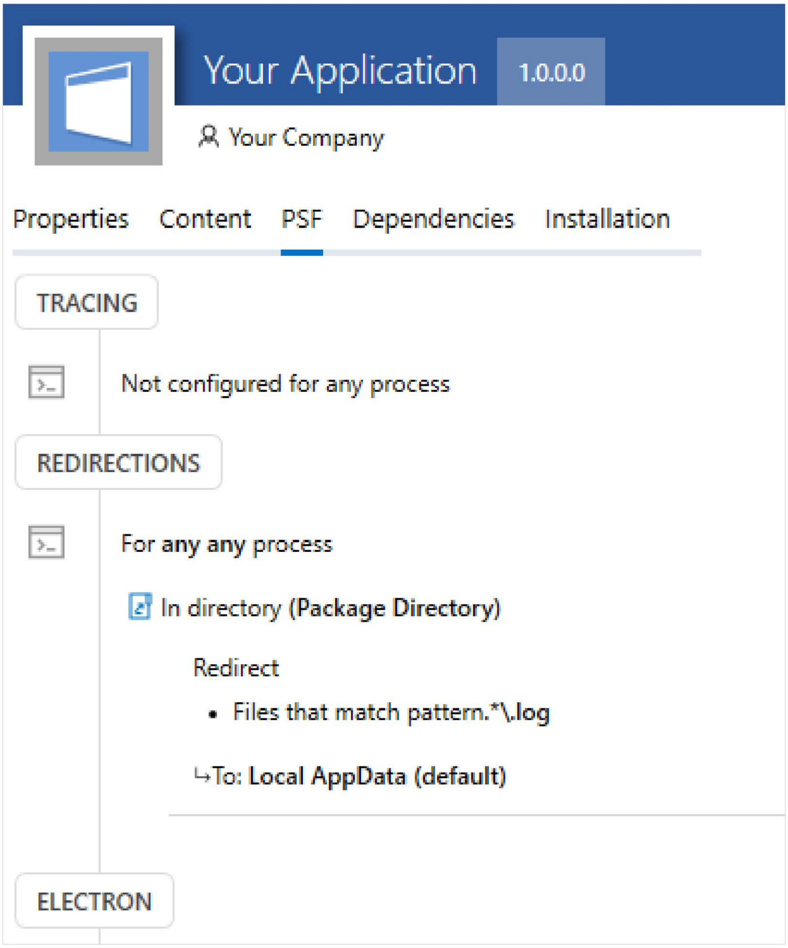
```
{
    "applications": [
        {
            "id": "PSFLAUNCHER1",
            "executable": "SampleApp/Sample.exe",
            "workingDirectory": "SampleApp/"
        }
    ],
    "processes": [
        {
            "executable": "Sample",
            "fixups": [
                {
                    "dll": "FileRedirectionFixup.dll",
                    "config": {
                        "redirectedPaths": {
                            "packageRelative": [
                                {
                                    "base":
"SampleApp/",
                                    "patterns": [
                                        ".*\\.log"
                                    ]
                                }
                            ]
                        }
                    }
                }
            ]
        }
    ]
}
```

A sample configuration JSON file

The JSON file uses *Regular Expression* (RegEx) patterns in many of the fields. The flavor used is known as the ECML variety that is also used in JavaScript. So the “.*\\log” in RegEx means a file that starts with anything, but ends with a case sensitive ".log" file extension. Microsoft describes the RegEx language in this documentation[46].

Another useful tool for inspecting fixups is "MSIX Hero". This free application informs you about any PSF fixups present in the package. Here is how it looks when inspecting the same MSIX package.

[46] https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference



TRACING

Not configured for any process

REDIRECTIONS

For any any process

In directory (Package Directory)

Redirect

- Files that match pattern.*\.log

↳To: Local AppData (default)

ELECTRON

The same sample configuration file - interpreted by MSIX Hero

For the latest updates on the Package Support Framework, join the [Microsoft PSF community](https://techcommunity.microsoft.com/t5/package-support-framework/bd-p/Package-Support)[47].

[47] https://techcommunity.microsoft.com/t5/package-support-framework/bd-p/Package-Support

Scripting

A traditional installer unpacks file and registry items and spreads them around the system as required. Under MSIX, these items are unpackaged into an isolated area, and the package Manifest describes any integrations that are required to be present outside of this isolated area (such as shortcuts and file associations).

Traditional installers also tend to perform customizable actions, investigating the environment and making changes based on what it finds. Even if the application vendor's installer doesn't do this, quite often the larger organizations that distribute applications to many users tend to either repackage these customizations, or add scripts to do so.

These customizations may be used to perform licensing operations, configure appropriate back-end servers (especially for dev/test/prod), or determine if integrations with additional applications are required.

Out of the box, MSIX does not support installation time customizations, but when you use PSF, you enable scripting. PSF Scripting is configured and controlled via the Json configuration file and implemented by PsfLauncher.

Scripts may only be triggered by two events: **the start** and **the end** of a process. Although there is no "install" script trigger, the "RunOnce" feature of the start trigger provides the effect of an installation script. The RunOnce feature writes a small marker in the Application Registry Hive so that subsequent launches will not run the script again.

The scripts for your packages must be written in PowerShell as PS1 files, and if you need to use the “wait” and “stop on error” features., the script must return a “0” to indicate success.

In a script triggered by the start event, the "wait" feature makes the launcher wait for the script to run to completion before starting the target application. When "wait" is requested, there is a timeout value that must be provided (in milliseconds) and is used as the maximum wait time. If this time is exceeded, it is considered an error, and the application will only be launched at that time if “stop on error” was not specified.

Note: You may want to reference file locations in your script that are dependent on the package installation location (which is dependent on the package name and AppX Volume mounting) or where the redirected user profiles are (dependent on the package name and user name).

To help you with this, PsfLauncher supports two pseudo-variables that you can use in the Json file as script arguments. These are:

Pseudo-variable	**Meaning**
%MsixPackageRoot%	The root folder of the installed package.
%MsixWritablePackageRoot%	The user’s LocalAppData is for

	file redirection for this package.

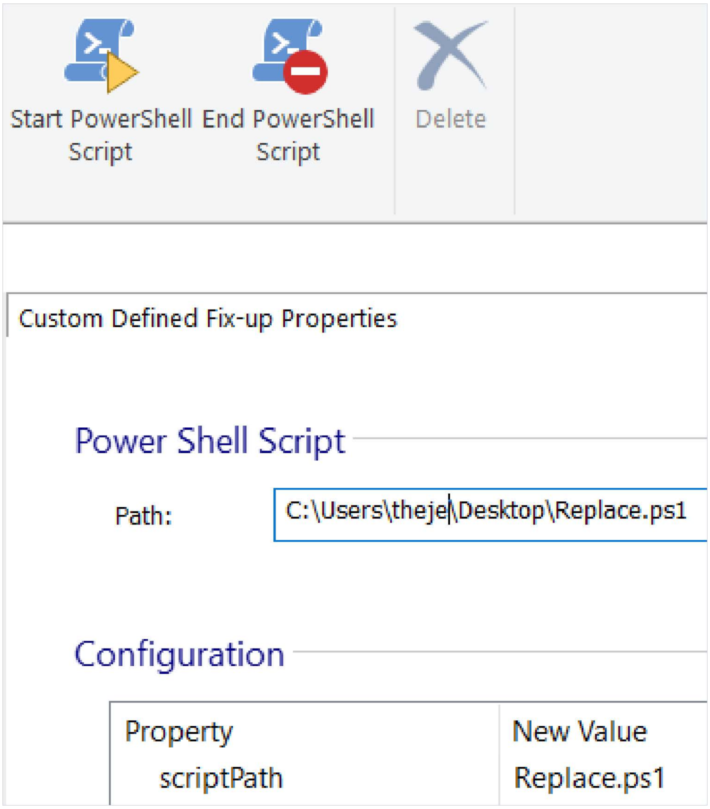
The PsfLauncher will decode these arguments which can be passed into your script. However, your script should not contain these syntaxes as "environment variables" because they are not; the PsfLauncher must do that decoding and pass the equivalent path as the argument to your script.

PsfLauncher scripting requires an additional file to be placed in the package. This file named “StartingScriptWrapper.ps1”, is part of the PSF and should be placed in the folder alongside the copy of the PsfLauncher. This Ps1 file is used for both the start and the end trigger scripts and its name can not be changed. Also, the Json configuration will only reference the additional Ps1 file that you provide.

PsfLauncher will call StartingScriptWrapper.ps1 with your script filename and arguments from the Json added as command line arguments.StartingScriptWrapper will manage the run of the script for you.

To allow these PowerShell scripts to be executed, you need to adjust the execution policy. You can do this through a Group Policy Object and set it to Unrestricted or RemoteSigned.
There is also an option to request ByPass mode on the script, however, you should not use this if the GPO is in place.
System changes to the execution policy must be performed on both x64 and x86 PowerShell executables.

Some packaging tools, like Advanced Installer, provide you with a predefined support to visually add and configure the scripts in your package, as part of their integration with the Package Support Framework.



Custom Defined Fix-up Properties

Power Shell Script

Path: C:\Users\theje\Desktop\Replace.ps1

Configuration

Property	New Value
scriptPath	Replace.ps1

Adding PowerShell scripts inside an MSIX, with Advanced Installer

MSIX Modification Packages

MSIX Modification Package is a type of MSIX package format introduced by Microsoft with the goal of decoupling customizations from the main application. Launched with the release of their MSIX packaging format, the MSIX Modification Package is an MSIX package meant to store the customizations of the application.

Windows 10 1809 is the first version to support MSIX Modification Packages.

Note. Make sure to use Windows 10 1809 or higher if you want to take advantage of the MSIX Modification Package. You can find more details on MSIX supported platforms here[48].

Is the MSIX Modification Package the new MST equivalent?

If you are familiar with Windows Installer technology, then you know that, just as with the MSIX Modification Packages, Transform files (MSTs) are meant to store the customizations of an MSI package. But that is as far as the similarities go for them.

MST files can be applied to the base MSI only at the time of installation and through the same msiexec command[49].

[48] https://docs.microsoft.com/en-us/windows/msix/supported-platforms
[49] https://www.advancedinstaller.com/user-guide/msiexec.html

For a broader perspective, let’s compare MSIX Modification Packages with Windows Installer Patches (MSPs). Although an MSP is applied to the MSI in the same way that an MSIX Modification Package is applied to the MSIX - there are two differences:

- It cannot be installed on its own
- It can be applied at any time (at installation time or later).

MSPs are meant to store other resources (bug fixes, security updates, and hotfixes), but not customizations.

In summary, the MSIX Modification Package is similar to MST because they both are meant to store customizations and, because they both are packaged separately from the target package.

A deep dive inside the manifest

This manifest section is always located in the AppxManifest.xml file of a modification package.

```
<Properties>

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

</Properties>
```

Going forward, there is a `MainPackageDependency` entry which is included within the **Dependencies** section of the MSIX Modification

Package manifest file, but not in the manifest file of the target application. It is called “TargetAppSample” in our example below.

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Desktop"
                      MinVersion="10.0.17701.0"
                      MaxVersionTested="12.0.0.0"/>

  <uap4:MainPackageDependency Name="TargetAppSample"
                             Publisher="CN=Contoso Software,
                             O=Contoso Corporation, C=US"
/></Dependencies>
```

This comes with two attributes:

- `Name` – it must match the identity Name of the target MSIX application.
- `Publisher` – it does not have to match the identity Publisher of the target MSIX application and this information could be missing if the publisher is the same.

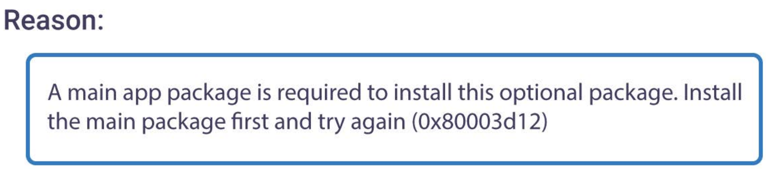
Also, since the MSIX Modification Package does not contain a full application, it does not have an entry point. In turn, there is no Application node within the manifest file of the MSIX Modification Package.

How does an MSIX Modification Package work?

One of the nice perks of decoupling the customization from the main application is that you will have two separate packages with no explicit relationship between them. And, since it is a separate package, you will not have to recreate the MSIX Modification Package with the same customizations each time there is a new update of the main application – obviously, as long as the customizations stored in the MSIX Modification Package are still applicable.

Not only do you not need to recreate the MSIX Modification Package, but there is no need to remove and reinstall it when a new update of the main application comes in. The customization stored in the MSIX Modification Package will apply successfully to the new MSIX package for the main application once the updated target MSIX package gets installed.

However, there's a small catch. The MSIX Modification Package doesn't contain a full application, thus it can't be installed on its own. The MSIX for the main application is required to be installed on the device prior to the MSIX Modification Package installation. If you mix-up the steps, you will be presented with an error like the one shown below:


Reason:

A main app package is required to install this optional package. Install the main package first and try again (0x80003d12)

An MSIX Modification Package cannot be installed before the target application

When an MSIX Modification Package is installed, its content will overlay (at runtime) on top of the resources from the target MSIX in a way that it will look like one application for the operating system and the main app.

Important! The overlay does not work for files outside the VFS folder. Any changes outside the VFS are ignored.

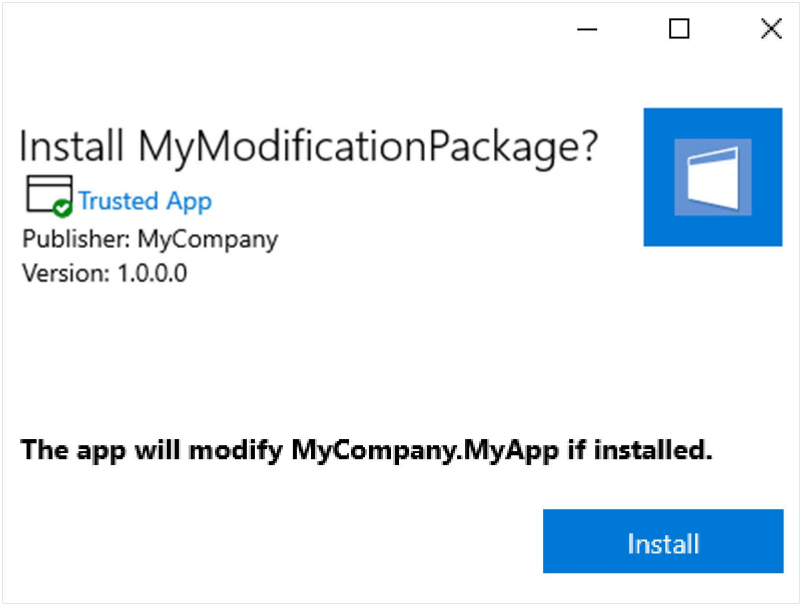
Let's do a small exercise. Imagine we have an executable file that reads the content of a .config file placed next to it and displays the outcome on the screen. Our example application is being packaged into MSIX.

Now, we want to replace the .config file with a new one (customized for our enterprise environment) using an MSIX Modification Package.

Note. For step-by-step instructions on how to create an MSIX Modification Package, you can watch our video tutorial on creating an MSIX Modification Package using the free Advanced Installer Express edition[50].

Now, take the MSIX Modification Package created with the new .config file included within and install that on a device where the target MSIX package is installed. App Installer will prompt a message to let you know that it will modify the target application.

[50] https://www.advancedinstaller.com/msix-modification-package.html


Install MyModificationPackage?

Trusted App

Publisher: MyCompany

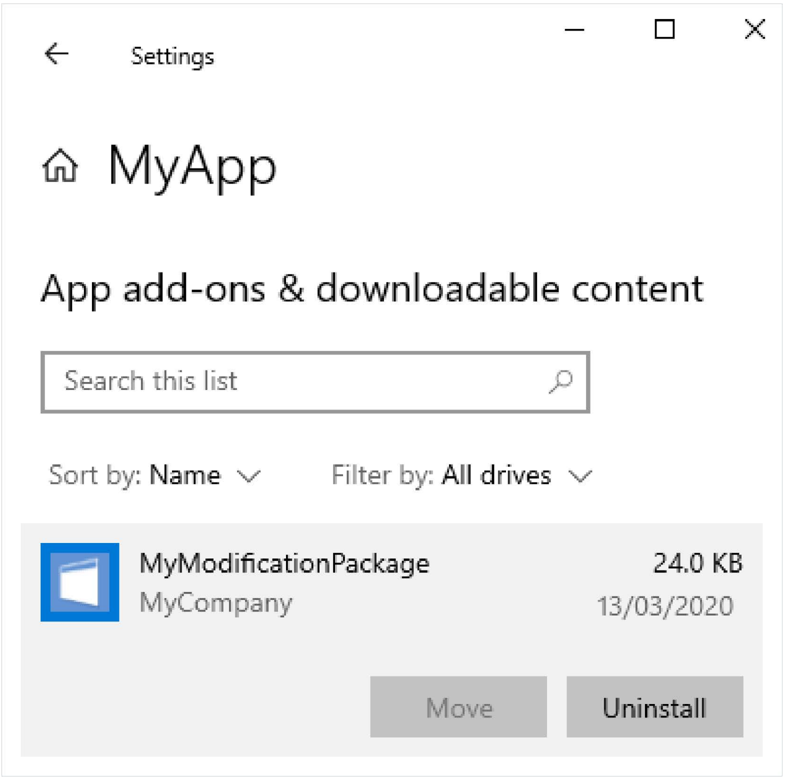
Version: 1.0.0.0

The app will modify MyCompany.MyApp if installed.

App Installer message when installing an MSIX Modification Package

Once the MSIX Modification Package is installed, if we launch our application, the executable will now read the content of the new .config file included within the MSIX Modification Package.

If you go to “App Settings” for the MSIX Package, you should find the MSIX Modification Package listed under the “App add-ons & downloadable content” section, here is an example for a target package called “MyApp” and its modification package “MyModificationPackage”:



List of MSIX Modification Packages installed on the device for a specific application("MyApp")

Or you can achieve the same output by using `Get-AppXPackage` PowerShell cmdlet:

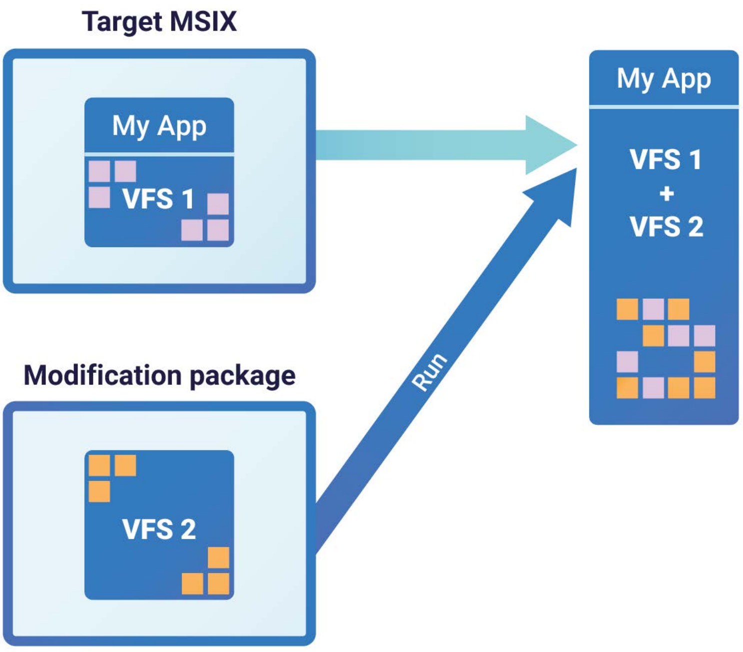
```
PS C:\Windows\system32> get-appxpackage -name "My
Company. MyApp"

Name                    : My Company.MyApp
Publisher               : CN=Horatiu Vladasel
Architecture            : X64
Resourceid              :
```

```
Version                  : 1.0.0.0
PackageFullName          : My Company.
                         MyApp_1.0.0.@_x64_r21nowlrc5s2y
InstallLocation          : C:\Program Files\WindowsApps
     WMy Company. MyApp_1.0.0.0_x64_r21nⓇwlrc5s2y
IsFramework        : False
PackageFamilyName        : My Company.
MyApp_r21n@wirc5s2y
Publisherid              : r21now1rc5s2y
IsResourcePackage : False
IsBundle                 : False
IsDevelopment Mode       : False
NonRemovable             : False
Dependencies             :
                         {MyCompany.MyModificationPackage
                         _1.0.0.0_neutral _r21n@wirc5s2y}
IsPartiallyStaged        : False
Signaturekind            : Enterprise
Status                   : OK
```

Additionally, you can get a list of all installed MSIX Modification Packages using the following PowerShell cmdlet:

```
Get-AppPackage -PackageTypeFilter Optional
```

The main MSIX and the MSIX Modification Package are installed in separate folders under “C:\Program Files\WindowsApp”. When the application is launched, it will consider (“see”) both packages within a single container.



How an MSIX Modification Package is loaded by the main application

Reminder: Please note that C:\Program Files\WindowsApp folder is read-only and system protected.

For debugging, you can launch an external EXE (eg.: regedit.exe or cmd.exe) inside an MSIX container using Invoke-CommandInDesktopPackage[51] PowerShell cmdlet.

[51] https://docs.microsoft.com/en-us/powershell/module/appx/invoke-commandindesktoppackage

Alternatively, you can use Hover[52] - a free tool, built by the Advanced Installer team, which allows you to launch natively installed applications inside an MSIX container with just a double-click.

Basically all you have to do is to launch a cmd.exe process either using the above PS cmdlet or Hover and then navigate to the folder of the target MSIX package and execute a dir command to see the contents of the virtual folder, just as your application would “see it”, i.e. with the contents of the modification package merged inside.

Microsoft did a great job here, but there is still room for improvement. The MSIX Modification Package is an excellent opportunity for enterprises to handle the customizations of each of their MSIX packages separately.

What makes it valuable is that these customizations are now decoupled from the main application. This will save a lot of time, effort, and budget for IT professionals when packaging and deploying applications.

Application Updates

MSIX deployment utilizes the concept of matching the Package Name, as referenced in the internal AppxMainifest.xml file, along with version numbers to enable good application upgrade and downgrade scenarios.

An upgrade package may be created using one of two methods:

[52] https://www.advancedinstaller.com/hover.html

Type	Description
Edit existing package	The original package can be opened for edit, modified, and saved using a higher Version Number. Currently, tooling vendors are supporting this scenario by using a strict editor, which prevents recapture of an traditional upgrade/patch installer. Edit scenarios involving recapture (similar to how it is done in the App-V Sequencer) is not precluded by the MSIX technology and may become available in the future as well.
New package	A completely new package can be created using the same Package Name as the original package and a higher Version Number.

When creating an upgrade package, some packaging tools use some form of file naming that involves the Package Name. Some include the Version Number in the package name by default and some do not. Care should be taken to properly identify the package filename so that each version is unique.

The MSIX installation methods in the OS will recognise a situation where a different version of the same Package Name is being installed. It is capable of performing both upgrade and downgrade operations.

By default, an upgrade or downgrade will cause the user preferences data to be retained. After installation of this version, the previously installed version will be removed for this user, and if no other user profile on the system has the previous version published it will be removed from the system completely.

Thanks to single-instance download and storage, there are significant performance benefits for a version installation. As these benefits are based on file block hashes, they should be roughly the same no matter which method for packaging the new version is used.

User Settings and Data Associated with a Package

As end-users utilize the applications that we package and distribute, they often make changes to the application settings, sometimes altering or creating new data files.

With or without the Package Support Framework, MSIX implements a strategy for redirecting file and registry activity (when supported) to an area of the user's local profile. This strategy is generally limited to the scope of the structure of the package itself. In practice, writes to areas outside of the package boundary are not redirected at all.

This allows for updates to the settings of a package file to be redirected, even if the app writing a file to the Documents folder is not redirected (unless your package includes files in the Documents folder).

The MSIX redirection is located in the end-users AppData\Local folder, under a subfolder called Packages. Each package installed to the user's device will have a subfolder using the package name. Under this folder, there will be a large number of subfolders, and all the written settings/data.

MSIX redirected file data will be placed under the *LocalCache\Local\Microsoft\WritablePackageRoot* folder. Notice that even if the app attempts to write to the end-users AppData\Roaming folder, if that folder is included in the package coverage -- MSIX will rewrite that file to this folder.

MSIX registry data will be placed in the *SystemAppData\Helium* folder in the form of .dat files.

Keep in mind that UWP Programs (which can also be MSIX) use the package folder structure and may place additional settings and data files under the RoamingState folder of the package redirection area.

Getting Settings and Data Off the Local Profile

Scenarios like WVD, VDI, and SBC, require that user settings and data be migrated to an area other than the local profile. Additionally, some companies like to have physical desktops set up to support an instant swap-out. This also requires ensuring that these settings and data are redirected to off-box storage.

Traditionally, Roaming Profiles and/or Folder Redirection policies have been used to help with these scenarios. More recently, additional User Environment Management (UEM) products like Microsoft UEV, AppSense, and FsLogix have become popular.

Such additional products can be effective in managing user data and settings when MSIX is in use. There will be situations where the vendor configuration for the MSI version of the application is identical to that of the repackaged MSIX app, and other situations where the UEM product rules will need to be altered to look for the files in this redirected area.

Fundamental Deployment Concepts

One of the big achievements of the first iteration of the Windows operating system was the ability to easily install applications on a desktop computer. By allowing non-technical consumers to install applications, Microsoft helped set in motion an era of consumer software development that still powers so much of our lives today.

The story behind the Windows application installation and deployment has been evolving ever since. One could characterize the rough and tumble marriage between Windows and applications as a struggle between ease of use (for the end user) on one hand, and unsustainable complexity of the underlying technologies and management practices on the other.

From the development of basic installers, to the MSI framework, and then application virtualization and containers, Windows has responded to the rising tide of problems attributed to application deployment and management. The most recent chapter in the story of application deployment, as already mentioned, is the MSIX application package format ("MSIX").

One of the great improvements of MSIX is that the software installation process is almost entirely isolated from the device state, which significantly reduces the number of dependencies as well as the unintended consequences experienced with older installers.

According to Microsoft, MSIX delivers an impressive 99% installation success rate, making it one of the most predictable ways to deploy applications in modern environments.

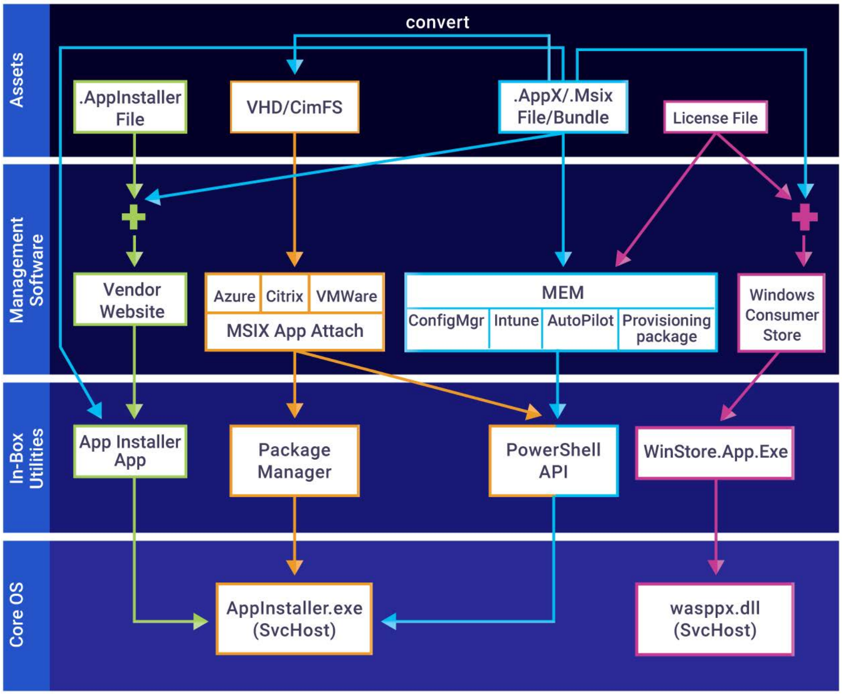
The increased installation success rate and predictability of MSIX comes with the added complexity of properly designing your unique MSIX deployment package. Microsoft has invested heavily in the technology to make sure it is suitable for a wide range of use cases, and as a result, there are a lot customization options you need to be familiar with.

Fortunately, for those of you that are new to MSIX, the options can be highly managed, whereas for those of you with more experience, you have access to power user features through more hands-on methods.

MSIX packages support not only installation and uninstallation, but also additional features:

- Support upgrade/downgrade scenarios
- Support for Add-on packages (Modification packages containing application configuration or plug-in modules).
- Support for referencing required external packages, such as framework or vc library packages.
- Support for referencing needed external drivers that can be triggered for installation by installing this package.

MSIX application package installation implementation support is built directly into the operating system, however we may use different methods to achieve package distribution and installation depending upon the customer requirements:



MSIX Deployment

While the core of the Windows Operating System includes system level components that ultimately are responsible for installing the package, the OS also comes with many different built-in utilities and tools that are used during deployment.

Different approaches to deploy MSIX packages will use different combinations of these components and utilities, and the previous diagram attempts to show the most common scenarios.

Some of these scenarios use new file types that we have not previously discussed, so before we start covering the deployment scenarios, we should discuss these file types.

Deployment File Types for MSIX

- **".msix" and ".appx".** Previously[anchor link to "The MSIX Package Layout"], the MSIX file format for ".msix" files was discussed. The overall format is shared by both .appx and .msix packages with differences in the AppXManifest file that make the package be a UAP or MSIX package. In fact, most of the utilities used to install the package don't care which of these two formats are used.

- **".msixbundle" and ".appxbundle".** Additionally, these OS utilities also support the "bundle" formats ".appxbundle" and ".msixbundle". The bundle is also a compressed file with a manifest, but also contains multiple .appx or .msix packages within the bundle. The Manifest provides information used by the installer to determine which of the internal packages to deploy. For example, the two packages might be present for x86 and x64, or for seperate localization (languages), or operating systems (windows versus android).
- **The "Store License" file.** Packages in the Microsoft Store require a license file. This even includes the free packages in the store. The license file comes in two forms, both xml and a binary format. When an end-user is acquiring apps from the Microsoft Store directly, the license file comes down with the package in the background and is not seen by the end-user. In scenarios where these packages are acquired centrally for deployment, the Administrator may need to deal with this file also. In addition to providing store-based licensing protections, this file also is used to help with the detection and application of updates to the package version.

- **The "App Installer" file.** The App Installer File is an additional Xml file used for certain types of deployment, especially when hosting of the MSIX package occurs on websites. This file usually points to the MSIX file, but also enables updates when used to deploy the package outside of the Microsoft Store.
- **VHD/CimFs**. Used with MSIX AppAttach, MSIX files can be converted to VHD files and mounted for fast deployment in VDI like scenarios. CimFs, a new read-only disk format for better performance, is expected to replace the use of VHD at some point.

MSIX Package Installations

Some general principles apply to all forms of MSIX deployments.

Upgrades and Downgrades

MSIX identifies packages via the package family name field of the package Manifest. For packages which use the same family name, the version string is also used for identification.

The MSIX installation process recognizes upgrades of newer versions, and will remove the older version when present automatically as part of the installation.

Originally MSIX did not support downgrades, however this was added to the OS support, and packages may now be similarly downgraded. To better understand this behavior the following table will illustrate some key points.

Installed (version)	Upgrade or Reinstall version	Behavior	Result
x86 (1.0)	x86 (1.0)	Reinstall	Supported
x86 (1.0)	x86 (3.0)	Upgrade	Supported
x86 (1.0)	x64 (1.0)	Reinstall	Not Supported
x86 (1.0)	x64 (3.0)	Upgrade	Supported
x86 (3.0)	x86 (1.0)	Downgrade	Supported
x86 (3.0)	x64 (1.0)	Downgrade	Supported

The first scenario in the table illustrates what would happen if the application needed to be re-installed in place. This sometimes happens to fix applications and is a supported action.

The next scenario shows an upgrade skipping application versions in-between. With MSIX the application upgrades do not need to be applied in sequence to upgrade an application.

In the next line we see the first consideration for changing the bitness of the application from 32-bit (x86) to 64-bit (x64) as a reinstall. This sort of upgrade or downgrade scenario is not supported as an reinstall. The new version of the application would be considered a completely new application installation that must be installed separately. The line below illustrates how the upgrade is possible between bitness but versions can also be skipped.

The last two lines illustrate the downgrade behavior, versions can be skipped in a downgrade plus bitness changes to the application are also supported.

Per User Installations

At the target system, MSIX package installations are performed on a per-user basis. This differs from traditional MSI and Exe based installers which supported the concept of either per-user or per-system installations.

Even though all installs for MSIX are per-user, MSIX does have something equivalent for the system level installation. MSIX packages may be pre-provisioned on a system-wide basis. This adds all of the package assets onto the system without performing the per-user integration aspects, and then registers a trigger that will complete the remaining installation integrations for the user whenever any user logs in. Such pre-provisioning is appropriate for adding MSIX packages to a standard image deployed to desktops.

Standard User Installation

Due to the isolation of the MSIX format and strict control over installation activities, including not allowing MSI Custom actions, MSI installs may normally be performed without elevation by a user with standard access rights.

The exception to elevation occurs if certain special actions are requested, exposed through the Capabilities settings of the AppXManifest. Currently only the inclusion of Windows Services requires this elevation.

When elevation is not required, all of the local system interfaces used to install the MSIX packages that are available to end-users, including PowerShell and the App Installer App, do not prompt for elevation.

While this simplifies software installation activities when using manual methods, it may also be of concern in the enterprise which depended upon authorized installations to ensure compliance with vendor licensing agreements, or to keep certain apps from only being accessed by authorized personnel that need access to the data exposed by the app.

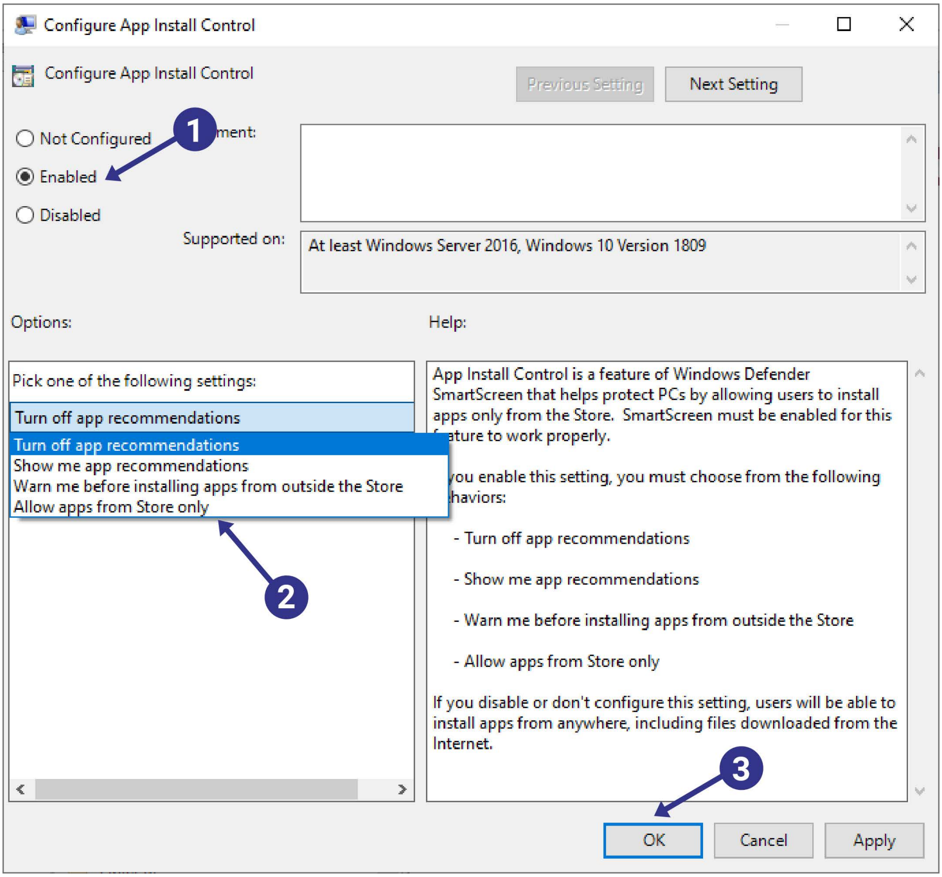
Organizations making use of open file shares as part of the deployment processes may need to review and update their deployment procedures accordingly.

Machines can use Group Policy to restrict the installation of applications to only be from the Windows Store. Starting with Windows 10 1703 the policy resides in the following location:
`Administrative Templates\Windows Components\Windows Defender SmartScreen\Explorer\Configure App Install Control`

Beggining with Windows 10 2004 this setting moved to its current location:
`Administrative Templates\Windows Components\Windows Defender SmartScreen\Explorer\Configure App Install Control`

To configure the policy, simply open a Group Policy editor and navigate to the policy and after opening it follow the instructions below:

1. Enable the policy
2. Select how strict to enforce Store apps
3. Click OK to accept the changes.



Supported on: At least Windows Server 2016, Windows 10 Version 1809

Pick one of the following settings:

Turn off app recommendations

- Turn off app recommendations
- Show me app recommendations
- Warn me before installing apps from outside the Store
- Allow apps from Store only

App Install Control is a feature of Windows Defender SmartScreen that helps protect PCs by allowing users to install apps only from the Store. SmartScreen must be enabled for this feature to work properly.

If you enable this setting, you must choose from the following behaviors:

- Turn off app recommendations

- Show me app recommendations

- Warn me before installing apps from outside the Store

- Allow apps from Store only

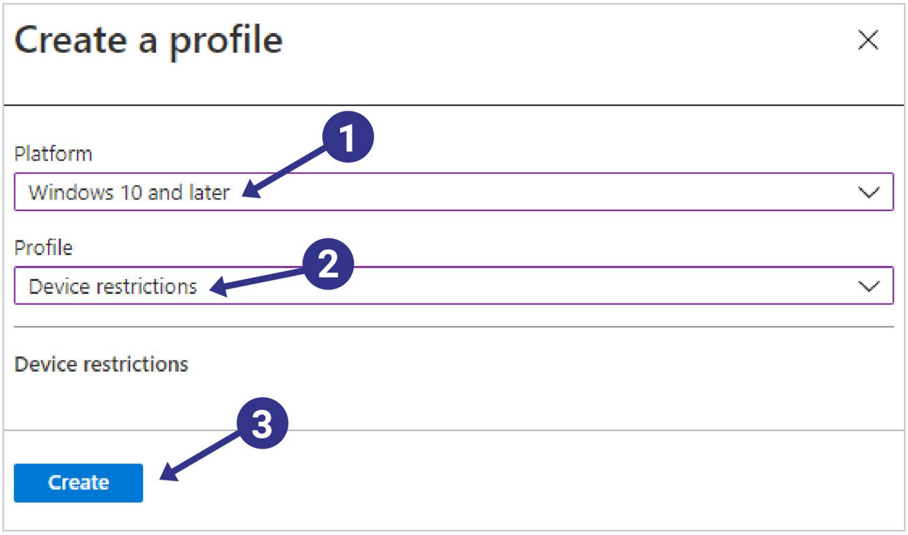
If you disable or don't configure this setting, users will be able to install apps from anywhere, including files downloaded from the Internet.

Intune supports this configuration and would be best configured using a device restriction configuration profile. To access it go to the Endpoint manager console location[53].

53 https://endpoint.microsoft.com/#blade/Microsoft_Intune_DeviceSettings/DevicesWindowsMenu/configProfiles

When creating a new configuration profile be sure to create a device restriction profile using the following procedure:

1. Change the platform to Windows 10 and later
2. Edit the Profile field to be Device restrictions
3. Click Create



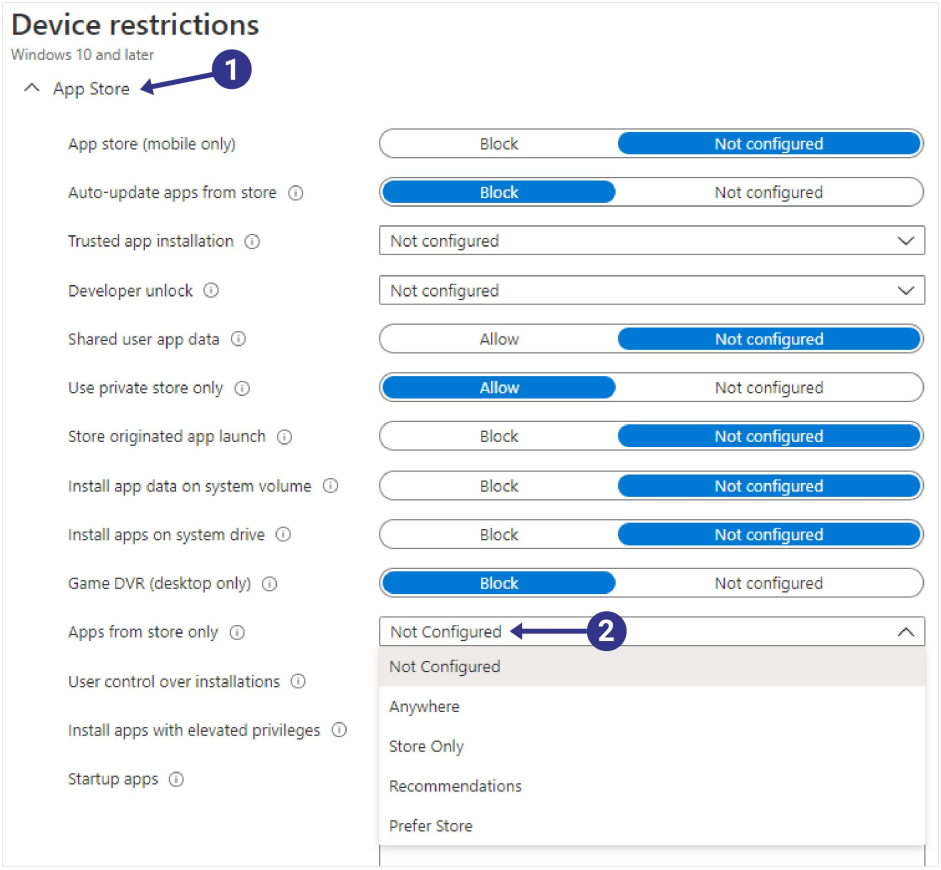
Create a profile

Platform: Windows 10 and later

Profile: Device restrictions

Device restrictions

In the App Store portion of the profile there is a setting labeled Apps from store only. Use the drop down menu to configure the Smart Screen settings for applications.



Device restrictions

Windows 10 and later

App Store

Setting	Value
App store (mobile only)	Block [✓ Not configured]
Auto-update apps from store	[✓ Block] Not configured
Trusted app installation	Not configured
Developer unlock	Not configured
Shared user app data	Allow [✓ Not configured]
Use private store only	[✓ Allow] Not configured
Store originated app launch	Block [✓ Not configured]
Install app data on system volume	Block [✓ Not configured]
Install apps on system drive	Block [✓ Not configured]
Game DVR (desktop only)	[✓ Block] Not configured
Apps from store only	Not Configured
User control over installations	
Install apps with elevated privileges	
Startup apps	

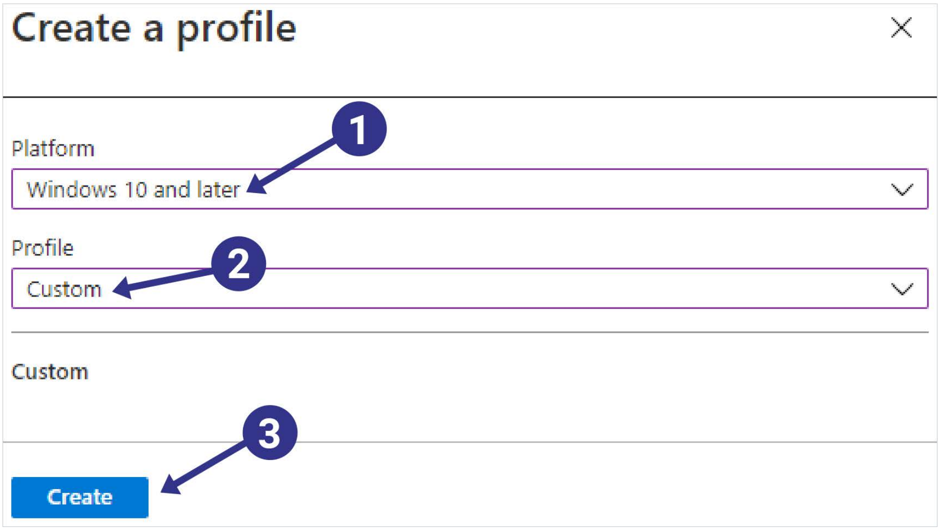
While this setting is helpful for locking down the application installs on the device it has an effect on administrator rights that may not go over so well in a production environment. By using this setting, the administrator of the device cannot install traditional Win32 applications if the configuration is set to Store Only.

For Intune managed devices starting with Windows 10 2004, the device can be configured to only allow local administrators to perform application management tasks. The user can still install from trusted sources such as the Windows Store or the Intune Company Portal app. A custom policy can be built in the Intune console for turning on this behavior.

To enable this configuration you will need to navigate to the Endpoint Manager console for Windows device policies[54].

Once there, you can create a custom policy to configure the setting.

1. Change the Platform to Windows 10 and later
2. Edit the Profile field to be Custom
3. Click Create


Create a profile

Platform: Windows 10 and later

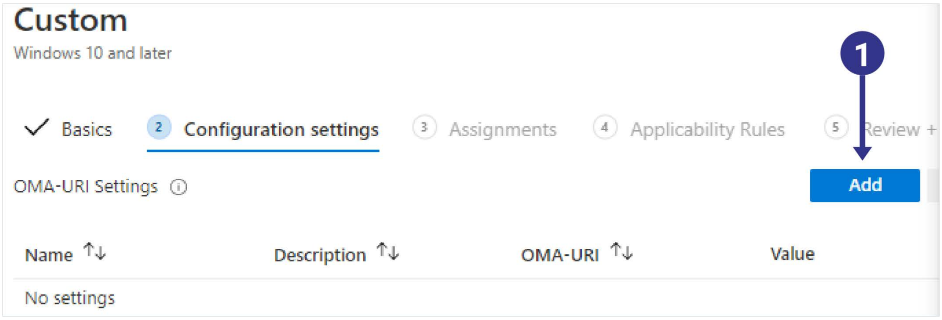
Profile: Custom

Custom

After you have created the policy and supplied a name for it you can move on to the settings. When using custom policies there is the potential for error since many of the settings need to be manually

54 https://endpoint.microsoft.com/#blade/Microsoft_Intune_DeviceSettings/DevicesWindowsMenu/configProfiles

inputted. As you can see with a new policy the contents are empty. Click Add to add settings to the profile.



Custom

Windows 10 and later

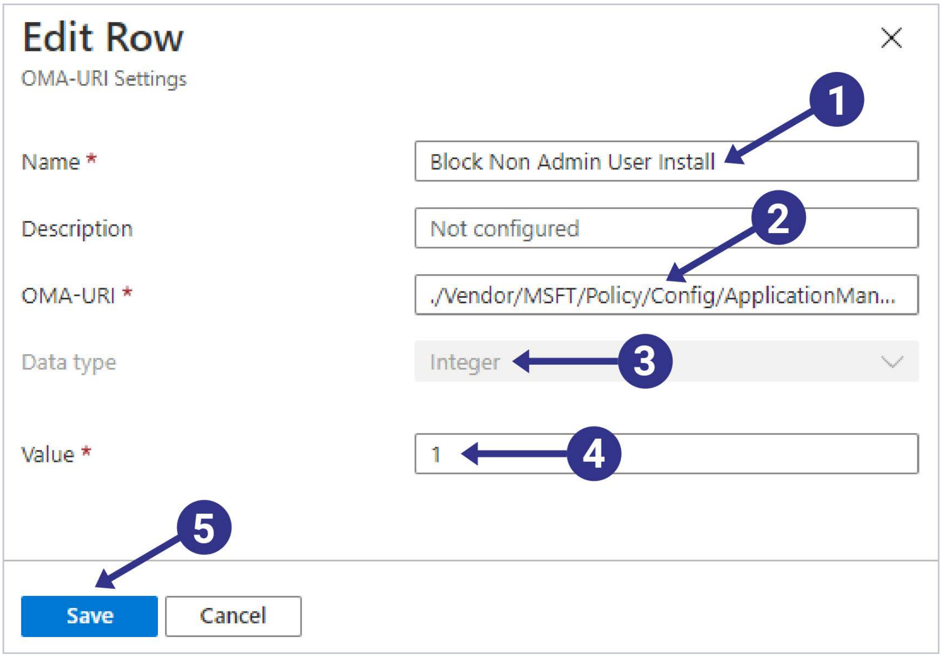
✓ Basics | 2 Configuration settings | 3 Assignments | 4 Applicability Rules | 5 Review +

OMA-URI Settings

Name	Description	OMA-URI	Value
No settings			

To configure the custom setting follow the steps below.

1. Give the setting a name
2. Put in the OMA-URI (this is case sensitive) ./Vendor/MSFT/Policy/Config/ApplicationManagement/BlockNonAdminUserInstall
3. Set the Data type to Integer
4. Set the value to 1
5. Click Save when complete


Edit Row

OMA-URI Settings

Name *	Block Non Admin User Install
Description	Not configured
OMA-URI *	./Vendor/MSFT/Policy/Config/ApplicationMan...
Data type	Integer
Value *	1

For more information on the BlockNonAdminUserInstall setting see the following Microsoft document[55].

Application Package Signing

Each MSIX package that you want to distribute must be signed with a code signing certificate that is trusted by the device or the installation will fail. This behavior (set by design) is meant to prevent untrusted code from being installed. While we covered the details on how to sign packages in the Digital Signing section of the Fundamentals chapter,

[55] https://docs.microsoft.com/en-us/windows/client-management/mdm/policy-csp-applicationmanagement#applicationmanagement-blocknonadminuserinstall

those involved with deployment still need to be involved with digital signatures.

In particular, if you have a business requirement to distribute an MSIX package to people outside of your organization, make sure you obtain a code signing certificate for your package from a major certificate authority.

Otherwise, you can use your existing certificate server infrastructure to issue a code signing certificate from your issuing certificate authority.

It takes a lot of work to set up a code signing process for your applications and that can pose a significant problem when you prototype MSIX applications and need to be more agile. While signing your packages is the recommended best practice, there is another way to install MSIX applications without code signing.

In Windows 10, you can enable developer mode[56] on the device which allows you to install unsigned MSIX packages. Many enterprise environments will not enable developer mode on production systems as a security precaution, so you may have to request a security policy exception to use developer mode on a device.

The key point is that there must be a valid code signing certificate that the device will use to trust the application. The application must then be signed using the same certificate to complete the chain of trust.

Ideally, in-house packages will be signed with a digital certificate that will be distributed to all appropriate organizational computers either as part of the base image or applied via PKI. MSIX Packages obtained from reputable third party vendors will most often be signed using a

[56] https://docs.microsoft.com/en-us/windows/uwp/get-started/enable-your-device-for-development

certificate obtained from a public Certificate authority, which might automatically be trusted by your PKI.

When the certificate is not trusted, it is still possible to add the certificate signature thumbprint to a system by right clicking on the MSIX file for properties. From the Digital Signatures tab it is possible (using elevation) to view and install the certificate on your system. The certificate must be placed in the “Trusted Root Certification Authorities” store of the Local Machine.

Using the App Installer App

Installing App Installer App

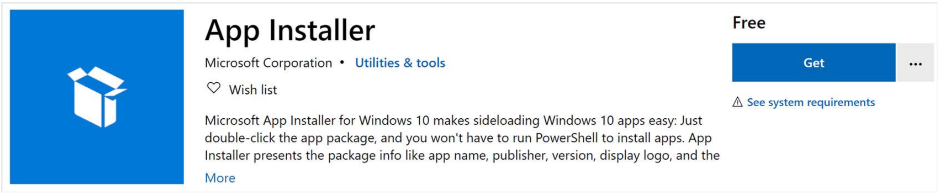
The App Installer application is now included with the operating system, however earlier versions of the OS required it to be obtained and installed from the Microsoft Store[57].

The App Installer app was built to simplify the installation of MSIX applications. With it, you can access a user-friendly double click method to install applications and avoid the complexity of PowerShell commands.

Another great feature of the App Installer app is that you get meaningful error messages to help you diagnose installation issues. If there are installation issues, the application is designed to provide error messages that help diagnose the application installation issue.

For earlier versions of the OS, the App Installer App may be obtained and installed from the Microsoft Store.

[57] https://www.microsoft.com/en-us/p/app-installer/9nblggh4nns1


App Installer

Microsoft Corporation • Utilities & tools

Wish list

Microsoft App Installer for Windows 10 makes sideloading Windows 10 apps easy: Just double-click the app package, and you won't have to run PowerShell to install apps. App Installer presents the package info like app name, publisher, version, display logo, and the

More

Free

Get

See system requirements

App Installer app available in the Microsoft Store

Alternatively, you can open the Windows Store app and search for “app installer”. It should be the first application in the results.

Click **Get** to install the application. You may be prompted to sign into the store if you have not configured the Windows Store with your Microsoft account.

The application will download and install and no further action or configuration is required to use the features of the App Installer.

Application Management via App Installer App

Manual Software Installation

With the App Installer present on the machine, it is possible to double click both MSIX and AppX applications as well as application bundles and get them to install (similar to an offline installation). There also is the .AppInstaller file that helps define more complicated application installations while providing a framework for light management of application updates.

The App Installer App uses a small GUI that aids the user performing the installation:

- To verify whether the package is validly signed with an acceptable certificate.
- Distinguish new installation, upgrade, and downgrade scenarios.

- Understand dependencies, including modification package installation situations.

App Installer App from Web Sites

The App Installer also allows users to initiate an installation from web accessible URLs that point to either an MSIX/AppX application, application bundles or an .appinstaller file. To successfully host MSIX packages from the web, the following needs to be in place.

Web Server Configuration

- Support for HTTP/1.1: The web server needs to support byte range requests from the client to fulfill the communication requirements from App Installer.
- Required MIME Types: New MIME types are needed to support MSIX

File Extension	MIME Type
.appx	application/appx
.msix	application/msix
.appxbundle	application/appxbundle
.msixbundle	application/msixbundle
.appinstaller	application/appinstaller

Required MIME types to host MSIX packages for web consumption

The MIME types in the table above are needed to configure the web server's response to the incoming request. For example, if you use IIS as a web server to host your MSIX packages, the following settings would be added to the web.config file.

```
<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>
```

The process is similar for other web hosting platforms.

Create a Clickable Link on a Web Page

If you are using a web page to provide an easy way to distribute the application installation or to share a common link through Emails, the current best-practice is to prefix the URL path to the MSIX package with the App Installer handler.

The App Installer handler is "`ms-appinstaller:?source=`". Below is an example of the final string.

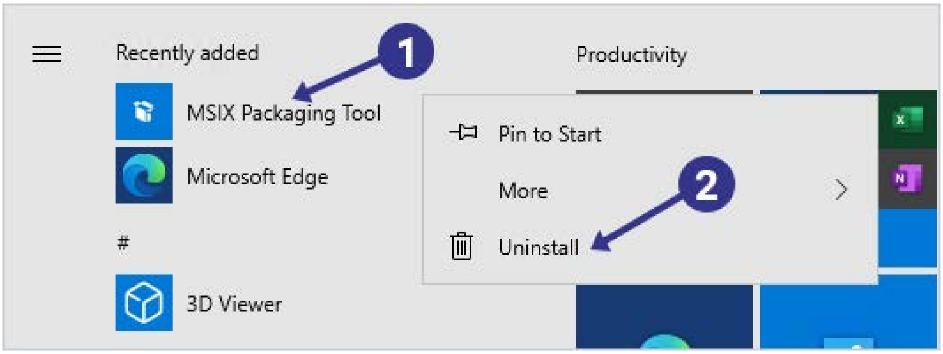
```
ms-appinstaller:?source=
https://msixdemo1.azurewebsites.net/MySample.msixbundle
```

When you edit the web page that references the MSIX application, add the **App Installer handler** (`ms-appinstaller:`) to the relevant HTML link using the source parameter (`?source=`). You can see an example below where the App Installer handler is added to the href property of an anchor element (<a>), so it contains the full reference.

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Install My App</title>
  </head>
     <body>
       <a href="ms-appinstaller:?
          source=https://msixdemo1.azurew
          ebsites.net/MySample.msixbundle">
             Install My App</a>
     </body>
</html>
```

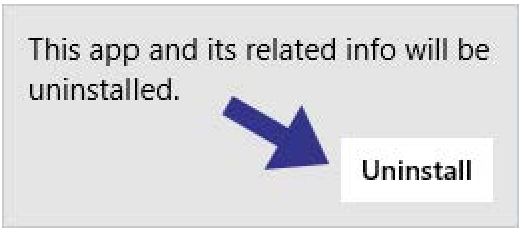
Uninstall an MSIX Package using App Installer App

The process to remove an MSIX package is straightforward. Simply locate it in the **Start Menu** and right click, then select **Uninstall**.



Uninstall MSIX package from a device

You will be prompted to uninstall the software. Confirm the uninstall action by clicking **Uninstall** in the prompt.


This app and its related info will be uninstalled.

Confirmation of the Uninstall of an MSIX package prompt

While quite simple for the end-user, the admin should be aware of the following:

- Like installation, in most cases an uninstall may be performed without elevation.
- By default, this method of installation removes not only the installed package, but also removes any user settings stored in the container and/or application registry hive.
- Removal of a package also automatically removes any Modification package to the package. If you wish to remove the modification package only, this is accomplished via the Windows Settings application by locating the application, exposing the advanced details to see the modification packages and uninstalling from there (or via PowerShell).

Installing MSIX with PowerShell

Before we dive into using PowerShell to manage MSIX installations, here's a list of the primary PowerShell cmdlets used for managing packages:

PowerShell cmdlet	**Description**

Add-AppXPackage	Used to install a signed *.msix or *.appx application along with related and dependent packages.
Get-AppXPackage	This cmdlet is used to provide a list of *.msix and *.appx applications on the device.
Remove-AppXPackage	Used to remove a signed *.msix or *.appx application from the device.
Get-AppXPackageManifest	Can be used to read the manifest of an installed application as an XML object.

PowerShell cmdlets for managing MSIX applications

Note: Each of these commandlets support appx and msix styled packages and are aliased, such that Add-AppPackage (without the X) is the same as Add-AppXPackage. We will show the name using the X in this book to highlight that we are talking about MSIX.

With these four cmdlets, you can handle most of your MSIX management needs. Make sure to familiarize yourself with the above cmdlets as they are frequently used.

The Add-AppXPackage Cmdlet

The Add-AppXPackage cmdlet has several use cases.

Installing a simple package

The most common use case of Add-AppPackage is to install a simple package. As a general best practice, file paths should be enclosed in double quotes to account for spaces in the path and allow for the use of variables in the file path (if needed).

```
Add-AppXPackage
  -Path “C:\Packages\MyPackage\MyPackage.msix”
```

Handling package dependency packages

Packages may include a listing of dependency packages in their manifests. These dependencies, which are generally packages with commonly used components like runtimes and frameworks, must be available for the package to be installed. Dependencies of a package may be installed first, prior to this package, or may be installed at the same time using the **DependencyPath** parameter.

```
Add-AppXPackage
  -Path
  “C:\Packages\MyPackage\MyPackage.msix”
  -DependencyPath
  “C:\Packages\MyPackage\MyPackageDependency.msix”
```

Handling External (Modification or Optional) packages

Modification Packages are examples of how the ExternalPackage option may be used. While the modification package may be installed using the first Add-AppXpackage after the primary package is installed, it may also be installed with the primary package this way. It is important to note that ExternalPackages is an atomic operation which means that if the additional packages fail to install, the whole installation operation is aborted.

The ExternalPackages parameter specifies an array of one or more strings that contain file paths.

```
Add-AppXPackage
  -Path
  "C:\Packages\MyPackage\MyPackage.msixbundle"
  -ExternalPackages
  "C:\Packages\MyPackage\OptionalPackage.msix",
  "C:\Packages\MyPackage\OptionalPackageBundle
  .msixbundle"
```

Handing RequiredContentGroups

One feature that a software developer can use to improve installation performance of a large package is to install only the required components and trigger the rest of the components to install later. Microsoft refers to this as application streaming, although it is unrelated to the App-V style of streaming that MSIX does not implement. An example of this may be a game where additional levels are not needed until the user finishes early levels, but optional features that can be triggered by a user-interface component could also be implemented by the developer this way. When such a package is built, the installation can use the **RequiredContentGroupOnly** parameter, which installs only the required components of the application, leaving the optional components registered in a way that the application can later trigger the download of optional content.

```
Add-AppXPackage -Path
  "C:\Packages\MyPackage\MyPackage.msixbundle"
  -RequiredContentGroupOnly
```

The Get-AppXPackage Cmdlet

The Get-AppXPackage cmdlet allows you to view which applications are installed on a device and is the primary method to identify which applications are installed on a device so that you can manage them using the other PowerShell cmdlets.

Getting the user's package list

It is common to start with the following command to generate a list that covers applications installed for the logged in user.

```
Get-AppXPackage
```

The resulting list will include all kinds of applications, including those that are part of the system, store apps, and developer signed ones. The output will be a list of packages, one of which is shown here:

```
Name                  : windows.immersivecontrolpanel
Publisher             : CN=Microsoft Corporation,
                        O=Microsoft Corporation,
                        L=Redmond, S=Washington, C=US
Architecture          : Neutral
ResourceId            : neutral
Version               : 10.0.2.1000
PackageFullName       : windows.immersivecontrolpanel
                _10.0.2.1000_neutral_neutral_cw5n1h2txyewy
InstallLocation       : C:\Windows\ImmersiveControlPanel
IsFramework           : False
PackageFamilyName     : windows.immersivecontrolpanel
                        _cw5n1h2txyewy
PublisherId           : cw5n1h2txyewy
IsResourcePackage     : False
IsBundle              : False
```

```
IsDevelopmentMode : False
NonRemovable      : True
IsPartiallyStaged : False
SignatureKind     : System
Status            : Ok
```

Sample output generated for an application by Get-AppXPackage

Eliminating System and Store Apps

While the cmdlet directly supports certain types of filters, we can also pipe the output to an external filter to be more creative. For example, the following line would filter only those packages signed by a developer:

```
Get-AppXPackage   | % {if($_.SignatureKind -eq
“developer”){$_.name}}
```

This produces a list of just the names, but changing “$_.name” to “$_” would show the entire object.

```
PascalBerger.MSIXCommander
MSIXHero
NotepadPlusPlus-O2004-M2020.824-P430-F
UltraEdit-Class102-O2004
Avogadro-O2004-M2020.824-P430-P
AppPersonalization-Class102
TMurgent-LotsOfFonts-M2020.1006-P4433
23572TimMangan.TMurgent-PsfTooling
```

Sample output of a filtered list shown package names only

Getting all applications

If you want to see all applications, for each user account registered on a device, so that you can perform a cursor assessment of the applications that are installed on a device, the following cmdlet parameter is available.

```
Get-AppXPackage -AllUsers
```

Note: To use the -AllUsers parameter, the PowerShell process must have been elevated using RunAsAdmin. This is true for any AppX PowerShell command that affects more than the current user.

When you run that command, the console output will list all the applications on the machine, including all the specified parameters. The output for a single application is presented in the figure below.

```
Name                             :
Microsoft.MicrosoftEdge.Stable
Publisher                        : CN=Microsoft Corporation,
                             O=Microsoft Corporation,
                             L=Redmond, S=Washington, C=US
Architecture                     : Neutral
ResourceId                       :
Version                          : 85.0.564.63
PackageFullName                  : Microsoft.MicrosoftEdge.
               Stable_85.0.564.63_neutral__8wekyb3d8bbwe


InstallLocation                  : C:\Program
Files\WindowsApps\
                            Microsoft.MicrosoftEdge.
                            Stable_85.0.564.
```

```
                           63_neutral__8wekyb3d8bbwe
IsFramework                      : False
PackageFamilyName          : Microsoft.MicrosoftEdge.
                            Stable_8wekyb3d8bbwe
PublisherId                      : 8wekyb3d8bbwe

PackageUserInformation : {S-1-5-21-250224505-1589582600-
2170371117-1001 [kkaminsk]: Installed}

IsResourcePackage                : False
IsBundle                         : False
IsDevelopmentMode                : False
NonRemovable                     : False
IsPartiallyStaged                : False
SignatureKind                    : Developer
Status                           : Ok
```

Sample output generated for an application by Get-AppXPackage -AllUsers

Additional Filtering Options

Once the big picture is assessed, the next step is to begin filtering the application data returned by Get-AppXPackage. You can filter for "application name" by including the **Name** parameter with a wildcard character.

In the next example, you can see how to use the Get-AppPackage command to filter the results for applications named beginning with “Microsoft.”.

```
Get-AppXPackage -Name Microsoft.*
```

You can add in multiple filters to the Get-AppPackage cmdlet. So, for example, most of the time we want to know which applications are assigned to a particular user on a device. Add the **User** parameter to the previous example along with a user account and Get-AppPackage will return a list of applications published to that user.

```
Get-AppXPackage -Name Microsoft.ScreenSketch -User
 LocalUser
```

In some instances, you may need to specify the domain that the user is a part of when a device is joined to Active Directory. However, the domain name is optional when using this parameter.

```
Get-AppXPackage -Name Microsoft.ScreenSketch -User
   Contoso\DomainUser
```

The Remove-AppXPackage Cmdlet

Once you understand how to use Get-AppXPackage and Add-AppXPackage, removing a package is straightforward. The Remove-AppXPackage cmdlet is the equivalent to the uninstall feature of the start menu seen in the previous section.

The first step to removing a package with the cmdlet is finding the **PackageFullName** assigned to the package, which is a composite of other parameters.

You can look up the **FullPackageName** property for an application using the Get-AppPackage cmdlet (see the above figure: **The console output generated for an application by Get-AppXPackage**).

The important thing to remember is that the **Remove-AppXPackage** cmdlet expects the **PackageFullName** to be passed to the **-Package** parameter.

```
Remove-AppXPackage -Package
Microsoft.ScreenSketch_10.2008.22.0_x64__8wekyb3d8bbwe
```

Using Get-AppPackageManifest

The application manifest contains a great deal of metadata about the package. For example, there are times that you need to assess and understand what capabilities the application has registered with the operating system. To search the application manifest for assigned capabilities, try the following:

```
(Get-AppXPackage -Name "*ZuneMusic*" | Get-
AppXPackageManifest).Package.Capabilities)
```

The Get-AppXPackageManifest cmdlet generates an array of strings with that application's capabilities.

```
Capability
----------
{internetClient, privateNetworkClientServer,
musicLibrary, removableStorage...}
```

With a more complicated analysis, it is possible to review other elements that have an effect on user experience such as which applications have startup tasks. For example, if you need to know what applications have registered startup tasks the following bit of PowerShell will collect that information.

```
 # List all the app startups

 $startuptasks =
   get-appxpackage -pv app | get-appxpackagemanifest |
 %
 {
 if ($_.package.Applications.Application.
        Extensions.extension.startuptask.taskid)
     {
       [pscustomobject] @
       { PackageFamilyName = $app.PackageFamilyName
         TaskID = $_.package.Applications.Application.

 Extensions.extension.startuptask.taskid
       }
     }
   }
$startuptasks
```

The output should list off the PackageFamilyName and the tasks that are registered.

```
PackageFamilyName                     TaskID
-----------------                     ------
Microsoft.SkypeApp_kzf8qxf38zg5c      SkypeStartup
Microsoft.549981C3F5F10_8wekyb3d8bbwe CortanaStartupId
SpotifyAB.SpotifyMusic_zpdnekdrzrea0  Spotify
Microsoft.Todos_8wekyb3d8bbwe         ToDoStartupId
AppleInc.iTunes_nzyj5cx40ttqa
            {AppleMobileDeviceProcess, iTunesHelper}
```

Other MSIX/AppX Cmdlets

Additional, less frequently used, PowerShell cmdlets are part of the AppX module. These include cmdlets for provisioned packages (also available using the DISM command described in the next section), and those for Volumes (the location for the MSIX/AppX packages on any given disk partition):

- Add-AppXProvisionedPackage
- Get-AppXProvisionedPackage
- Remove-AppXProvisionedPackage
- Set-AppXProvisionedDataFile
- Optimize-AppXProvisionedPackages
- Add-AppXVolume
- Get-AppxVolume
- Mount-AppXVolume
- Unmount-AppXVolume
- Remove-AppXVolume
- Set-AppXDefaultVolume

Deployment with DISM

When talking about package deployment, Microsoft uses the term "offline" simply to indicate that the installation activity is disconnected from the end-user. For those instances where you need to perform an offline installation, you may deploy MSIX applications with the use of DISM[58], or alternatively you may use the Add-AppxProvisionedPackage PowerShell cmdlet. Both of these methods inject the packages into

[58] https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/mount-and-modify-a-windows-image-using-dism

Windows images when they are mounted offline. The functionality of these two methods appear to be the same, and where we show examples in this section using one or the other method you should be able to use the other.

Pre-provisioning allows for the application package content, and licensing file if needed, to be added to the Windows image as part of a preparation phase of imaging prior to sealing off the image for distribution. Microsoft uses pre-provisioning themselves within the OS images provided to you for many of the “in-box” UWP and MSIX application packages, such as Calculator.

When you pre-provision the application package in this way, the app is not yet fully installed, but all contents needed are present in the image. Installation will automatically complete when a user logs into the booted image after image deployment. In a multi-user operating system, this means that each user logging onto the OS will have their own installation completed. This portion of the per-user installation consists of fully integrating the application into the user profile, including important integrations such as to the Start Menu and file type associations.

Offline Installation

Application packages acquired from the Microsoft Store include licensing that must be applied. For offline installation to work, you need to acquire and add the licencing file when adding the package.

Currently, the Windows Store interface does not support separation of the license file, however organizations may use their Windows Store for Business, or Store for Education, to obtain the msix file or bundle and license file. When requesting the license file, you should download the xml version of that file when using the method shown here.

The **Add-AppxProvisionedPackage** commandlet adds a package to the image that is installed per user when they log into the device. You can customize the MSIX package further by including additional packages, for example, a license file with the use of the **LicensePath** parameter.

```
Add-AppxProvisionedPackage
  -Path C:\offline
  -PackagePath C:\Packages\MyPackage\MyPackage.msix
  -LicensePath C:\Packages\MyPackage\MyLicense.xml
```

Each application will have unique dependencies that will need to be included in the offline package, some will be required, and some will be optional. In most use cases (where there is an Internet connection), dependencies are simply downloaded when needed, but in the case of offline installations, required dependencies must be specified so that they are present to complete the application install.

Dependent packages are specified using the **DependencyPackagePath** parameter.

```
Add-AppxProvisionedPackage
  -Path c:\offline
  -PackagePath C:\Packages\MyPackage\MyPackage.msix
  -DependencyPackagePath
  C:\Packages\MyPackage\MyDependencyPackage.msix
  -LicensePath C:\Packages\MyPackage\myLicense.xml
```

Online Installation

Another scenario is to use DISM to install applications into online operating systems. This could be part of an MDT task sequence or some other customization script that runs after the operating system is installed.

In the following example, an application, its dependencies, and a license file are all contained in the folder “C:\MyPackage”. In order for DISM to load all the required components from the same folder, use the **FolderPath** parameter.

```
Add-AppxProvisionedPackage -Online -FolderPath
                           "C:\MyPackage"
```

Provisioning Packages

Provisioning packages are used to transform the default configuration of an off-the-shelf Windows device into the configuration an organization needs to provision the device to a user.

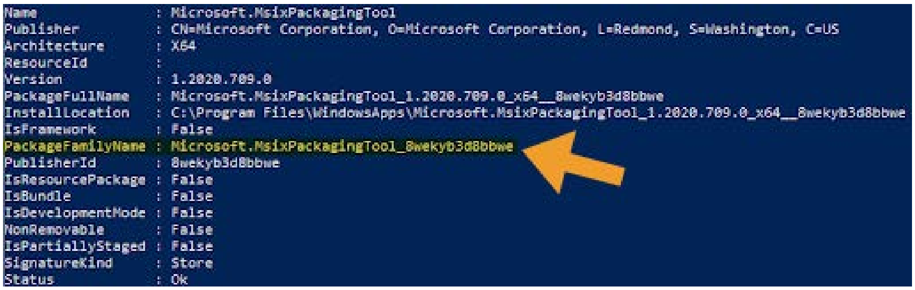
The provisioning package is built using the Windows Image Configuration Designer found in the Windows Automated Deployment Kit (ADK). Using this tool administrators can build a provisioning package file that can be hosted on SharePoint, websites, email or USB keys.

The provisioning package is typically for environments that require the quick provisioning of hardware that may not be owned by the organization. It is possible to use provisioning packages to install applications, however, the application installation requirements will determine if it is a possible option in your specific scenario.

The Provisioning Package can contain a copy of the application package and it’s dependencies for installation onto the device when it is applied. Installation can be triggered by adding a USB key to the device as part of Windows setup, in the Windows Settings app or double clicking the provisioning package file.

When you install an application with a provisioning package, you need two pieces of information. First, you need the path to the *.msix or *.msixbundle file that you want to install. Second, you need to know the package family name (PackageFamilyName) to complete the process.

You can use the Get-AppPackage cmdlet (in PowerShell) to look up this value by installing the application on a test machine and then running the cmdlet.



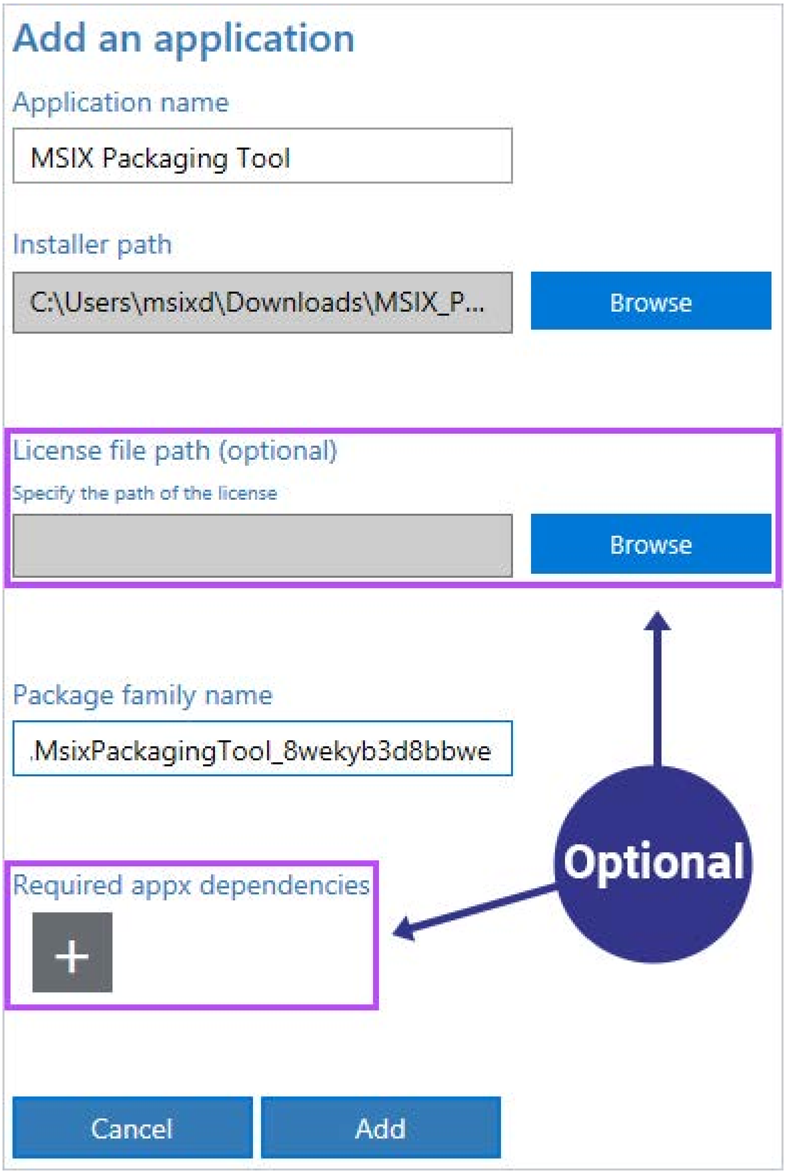
```
Name              : Microsoft.MsixPackagingTool
Publisher         : CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
Architecture      : X64
ResourceId        :
Version           : 1.2020.709.0
PackageFullName   : Microsoft.MsixPackagingTool_1.2020.709.0_x64__8wekyb3d8bbwe
InstallLocation   : C:\Program Files\WindowsApps\Microsoft.MsixPackagingTool_1.2020.709.0_x64__8wekyb3d8bbwe
IsFramework       : False
PackageFamilyName : Microsoft.MsixPackagingTool_8wekyb3d8bbwe
PublisherId       : 8wekyb3d8bbwe
IsResourcePackage : False
IsBundle          : False
IsDevelopmentMode : False
NonRemovable      : False
IsPartiallyStaged : False
SignatureKind     : Store
Status            : Ok
```

```
Name                              :
Microsoft.MicrosoftEdge.Stable

Publisher                         : CN=Microsoft
Corporation, O=Microsoft Corporation, L=Redmond,
S=Washington, C=US
```

```
Architecture                  : Neutral
ResourceId                    :
Version                       : 85.0.564.63
PackageFullName         :
Microsoft.MicrosoftEdge.Stable_85.0.564.63_neutral__8wek
yb3d8bbwe
InstallLocation               : C:\Program
Files\WindowsApps\Microsoft.MicrosoftEdge.Stable_85.0.56
4.63_neutral__8wekyb3d8bbwe
IsFramework                   : False
PackageFamilyName       :
Microsoft.MicrosoftEdge.Stable_8wekyb3d8bbwe
PublisherId                   : 8wekyb3d8bbwe
IsResourcePackage       : False
IsBundle                      : False
IsDevelopmentMode       : False
NonRemovable                  : False
IsPartiallyStaged             : False
SignatureKind                 : Developer
Status                        : Ok
```

Console output of Get-AppPackage and PackageFamilyName identified

The license file and dependency packages must be specified if required, otherwise, those values are optional. In the following picture, you can see the **Add an Application** screen, with the required data for the application.



Add an application

Application name: MSIX Packaging Tool

Installer path: C:\Users\msixd\Downloads\MSIX_P...

License file path (optional)

Specify the path of the license

Package family name: .MsixPackagingTool_8wekyb3d8bbwe

Required appx dependencies

Add an application to a provisioning package

MSIX and the Windows Store

It’s important to understand that Windows Store is more than just a consumer app store, it is a focal point for purchasing hardware and software for your organization.

The Windows Store is a cloud-based application store where organizations can curate a list of approved software that users can install themselves. Windows Store for Business and Windows Store for Education are the primary portals for organizations to access the Windows Store as their own private store. Unfortunately, you cannot push MSIX packages to devices using this technology because the Windows Store is a self-service oriented experience and users must trigger the application installations.

Instead, you need to assign applications using Azure AD security groups to publish them. The user simply has to "go to the store", "search for the application" to "install it".

With administrators, the process can be more complicated, not all enterprise software distribution scenarios require an integration with Windows Store. If the application has been published to the public store, then the process of adding the application to your organization is straightforward and done through the Windows Store for Business portal.

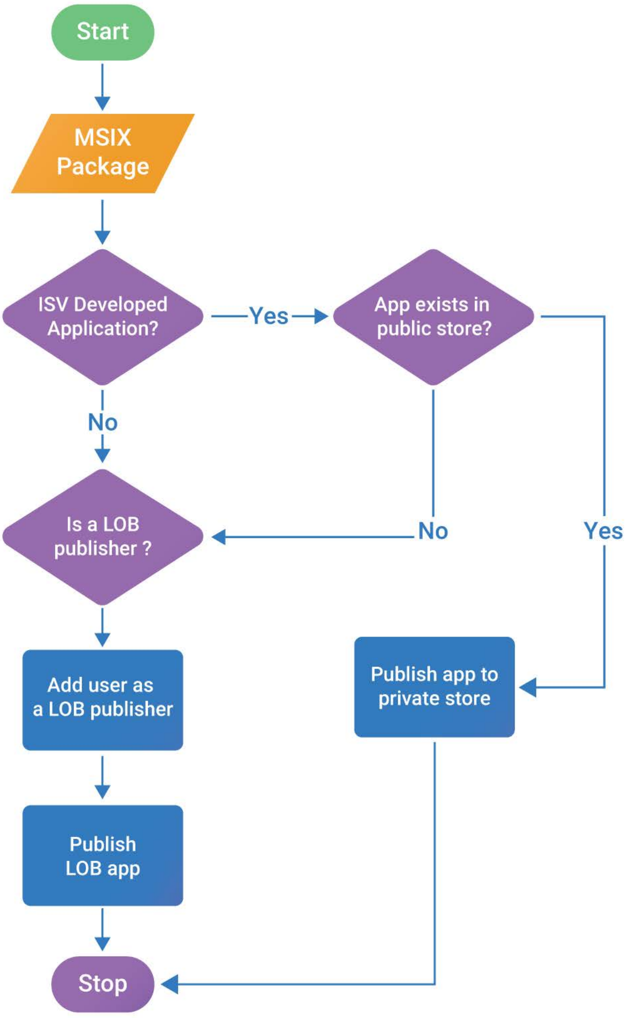
On the other hand, for repackaged MSIX applications and software not published through the public store, you can use a private Windows store. Under this approach, publishers are given access to the private Windows store as an LOB publisher for the organization. The publisher then submits applications and application updates using this mechanism.

All application changes are approved before the application is published, and the time it takes to synchronize private Windows Store applications is up to 5 days.

When LOB Publishers are used, and you're ready to send your custom applications to the private store, you need to keep some considerations in mind when estimating the timeline to get an application ready for deployment.

This process has prerequisites and several steps that require time until the next step can be performed.

- First, you need to determine the onboarding procedure that the publisher must go through to distribute applications with Microsoft. All LOB publishers must have a developer account with Microsoft before they can deploy applications to private Windows Stores.
- Next, you need to be aware of various events to take into account each time an application is published (see figure below).

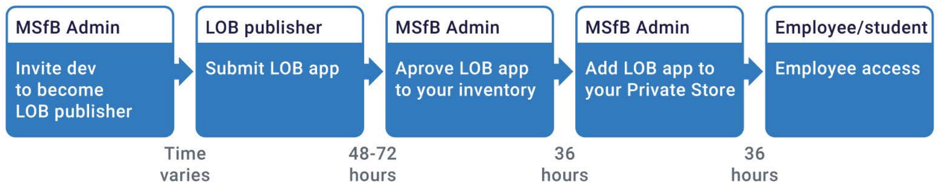

Start
MSIX Package
ISV Developed Application?
Yes
App exists in public store?
No
Is a LOB publisher ?
No
Yes
Add user as a LOB publisher
Publish app to private store
Publish LOB app
Stop

Onboarding diagram for LOB published applications

- Now, to put it into context, let’s assume that the supplier of the MSIX package is set up as an LOB publisher - the process needed to push applications to the private Windows Store will still take a significant amount of time to complete.

For example, when an application is submitted to a private Windows Store, the application can take 48-72 hours to appear in the customer's application inventory. Only after an application is available in the application inventory,the application is added to the private Windows Store by an administrator of the customer's Windows Store for Business tenant.

It can take another 36 hours for the application to become available to all users once it is available in the private Windows Store.



There are some points to consider in regard to deploying applications through a private Windows Store are:

1. To understand that it is an involved and time-consuming process to onboard a developer who is not already a certified Microsoft developer.
2. To be aware of the considerable delay when following the steps to publish an application and its updates to a private Windows Store and have it available to end users.
3. To know that deciding to use a private Windows Store should not be taken lightly. It may not be the best path for some of your in-house application packages because of the delays in publishing.

Note. In this last case, you might consider using an alternative to a private Windows Store, you can try researching a device management solution such as Microsoft Endpoint Manager, which can deploy MSIX packages with or without the store.

MSIX and the App Installer File

The MSIX package contains the AppXManifest file, which is an XML structured text file that contains information about the package as set by the developer/creator of the package.

An App Installer File is an additional external XML file that contains all the installation settings needed to deploy and manage the application(s) in the package. It typically contains dozens of critical management settings, such as enabling update management -- which makes the App Installer file scope to reach beyond application deployment.

The scope of the App Installer File becomes an important piece you need to account for when you begin designing your MSIX package and deployment workflow in some environments. In some cases, software vendors may choose to build website deployments using the MSIX packages and App Installer files. But you may also find this a useful technique to manage updates from within your organization. Deciding to use the App Installer File to help manage updates is a major decision that will impact your overall application management practices.

Before diving into the key components of the App Installer file, we need to mention one technical requirement for the App Installer file to function correctly on end user devices. Currently (end of 2020), the device must have Windows 10 1709[59] (or newer) installed and have the necessary APIs[60] for reading and modifying App Installer packages.

Once that requirement is out of the way, the first step in working with App Installer files is to choose a code editor that supports XML markup, autocompletion, and autoformatting. A proper code editor will help eliminate hard to catch syntax errors and efficiently focus on the core elements and their attribute values.

Note. We use Visual Studio Code, but there are lots of great alternatives available, so just use one you are comfortable working with.

Here's an App Installer sample file you can download for easy reference[61]. Alternatively, you can copy sample content from Microsoft's documentation[62] or start from scratch by creating a new XML file in your code editor with the ".appinstaller" file extension.

It is strongly recommended that you familiarize yourself with the official App Installer XML schema[63]and regularly review the core elements and

[59]https://docs.microsoft.com/en-us/windows/msix/app-installer/app-installer-file-overview
[60] https://docs.microsoft.com/en-us/windows/msix/app-installer/app-installer-documentation
[61] https://portalfuse.io/community/msixbook/appinstaller.html
[62] https://docs.microsoft.com/en-us/windows/msix/app-installer/how-to-create-appinstaller-file#app-installer-file-example
[63] https://docs.microsoft.com/en-us/uwp/schemas/appinstallerschema/schema-root

their attribute definitions because Microsoft changes which core elements are required as well as attribute data types from time to time.

When you create an App Installer file, make sure the standard XML schema declaration is present and you're good to go.

The first core element you need to include in the App Installer file is the **MainPackage** element declaration and the schema required attributes listed below.

```
<MainPackage
    Name="Contoso.MainApp"
    Publisher="CN=Contoso"
    Version="1.0.0.0"
    ProcessorArchitecture="x64"
    Uri="http://mywebservice.azurewebsites.net
         /mainapp.msix" />
```

The **URI** attribute contains a web accessible link where the application files are stored. If this value is different from the current location of the App Installer file, Windows will attempt to retrieve the latest copy of the application from the **URI** location.

The URI accepts connections through HTTP, HTTPS, or file schemes. HTTP is generally discouraged on public networks and should be avoided (if possible). Instead, use HTTPS to ensure the trust of your hosting site.

In the next example, the URI attribute contains a file share path.

```
    <MainPackage
        Name="Contoso.MainApp"
        Publisher="CN=Contoso"
        Version="1.0.0.0"
        ProcessorArchitecture="x64"
        Uri="file://server/share/mainapp.msix" />
```

For new applications, it is a good practice that you use the **xmlns** and **Version** attributes listed in the below table. The Version attribute specifies the current application version, so make sure you verify the version value before you deploy.

Attributes	**Constraints**	**Required**
xmlns	*Must be static (http://schemas.microsoft.com/appx/appinstaller/2017/2)*	*Yes*
Name	*A string between 3 and 50 characters with alpha-numeric, period, and dash characters allowed.*	*Yes*
Publisher	*A string between 1 and 8192 characters that fits the regular expression of a distinguished name. The string must be compliant with CertNameToStr Windows API implementation of X.500 rules.*	*Yes*
Version	*Quad notation (Major.Minor.Build.Revision)*	*Yes*
ProcessorArchitecture	*x86 | x64 | arm | neutral*	*Yes*
Uri	*Web Uri to main package*	*Yes*
HoursBetweenUpdateChecks	*0-255*	*No*
ShowPrompt	*true/false*	*No*

UpdateBlocksActivation	*true/false*	*No*
HoursBetweenUpdateChecks	*numeric*	*No*
ForceUpdateFromAnyVersion	*true/false*	*No*

App Installer file schema attribute details

In situations where you need to distribute an MSIX file, the App Installer file needs to contain a **MainBundle** element declaration with the details of the MSIX application to install.

The Name and Publisher attributes are both required, and the publisher value must be a valid X500 distinguished name.

If you are using an *.msixbundle or an *.appxbundle, then you must include the **MainBundle** element to define the settings. The URI for the MainBundle would point to an *.msixbundle or an *.appxbundle for installation.

```
    <MainBundle
        Name="Contoso.MainApp"
        Publisher="CN=Contoso"
        Version="1.0.0.0"
        ProcessorArchitecture="x64"
        Uri="http://mywebservice.azurewebsites.net
             /mainapp.msixbundle" />
```

For deployment scenarios that require more granular control over application deployment, or have additional dependencies to consider, use the **OptionalPackages** element or, in some instances, the ModificationPackages which will be discussed later.

With OptionalPackages, you can select which portion(s) of an application to install. It's a good idea to experiment with this technique to improve performance for your end users, especially with large applications.

When building your App Installer file, use the OptionalPackages element to describe the dependent application details.

In the following example, the application details are contained in a **Package** element because these examples are designed for a single MSIX file deployment. For those use cases where you need to deploy multiple MSIX files, use the **Bundle** element instead.

The attributes of the Package element have the same constraints as the MainBundle element. The URI points to an MSIX file. The example shows an MSIX package as an optional package.

```
    <OptionalPackages>
          <Package
              Name="Contoso.OptionalApp1"
              Publisher="CN=Contoso"
              Version="2.23.12.43"

Uri="https://mywebservice.azurewebsites.net
                /OptionalApp1.msix" />
     </OptionalPackages>
```

The next example uses only an *.msixbundle package as a dependency. As a result, the **Bundle** element is required instead of the **Package** element.

```
    <OptionalPackages>
        <Bundle
            Name="Contoso.OptionalApp2"
            Publisher="CN=Contoso"
            Version="1.32.13.63"
            Uri="http://mywebservice.azurewebsites.net
                 /OptionalApp1.msixbundle" />
    </OptionalPackages>
```

For larger organizations, it is strongly recommended to get to know the **ModificationPackages** element because it is where you define unique application customizations for the MSIX container that get applied to the application as it is installed.

The ModificationPackages element helps decouple customizations to the MSIX package into a separate package, while still giving admins the ability to fine-tune the application experience for the end user.

In the next example, the URI attribute contains a link to a modification package where the modification files are located.

```
  <ModificationPackages>
    <Package
        Name="Contoso.Customization"
        Publisher="CN=Contoso"
        Version="1.0.0.0"
        Uri="https://mywebservice.azurewebsites.net
            /ContosoCustomization.msix" />
  </ModificationPackages>
```

Dependencies, such as Visual C++ runtimes, are defined in the **Dependencies** element. The next example shows how to use the Dependencies element.

```
    <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="x64"
            Uri="http://mywebservice.azurewebsites.net
                 /fwkx64.appx" />
    </Dependencies>
```

The **UpdateSettings** element defines important parameters that control how updates are managed.

You can control whether updates happen independently from the application launch or only when the user launches the application. You can also choose whether the user is prompted or if the update is silent and happens in the background.

For example, including the **AutomaticBackgroundTask** element instructs the application to check for updates in the background.

```
    <UpdateSettings>
        <AutomaticBackgroundTask/>
    </UpdateSettings>
```

Note. The **AutomaticBackgroundTask** element and many other update settings are optional.

In the next example, find two useful update settings. First, the **OnLaunch** element, which forces the application to check for updates each time it is launched by setting the **HoursBetweenUpdateChecks** attribute to zero.

Next, we need to make sure that the user receives a prompt about the update by setting the **ShowPrompt** attribute to “true”.

Finally, a typical approach would be to force the user to update the application before it launches by setting the **UpdateBlocksActivation** attribute to “true”.

Then, you need to include the **ForceUpdateFromAnyVersion** element to ensure that I can rollback an app’s version in case there’s an issue. Without this setting configured, you can only increase the app’s version.

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

It’s important to have your app team make these decisions ahead of time and document the installation update settings.

The last thing to do with your App Installer file is to make sure you close the **AppInstaller** element in the file.

```
</AppInstaller>
```

And there you have it, make sure to review the complete demo file[64] as a reference, then try making your own App Installer file from scratch.

[64] https://portalfuse.io/community/msixbook/appinstaller.html

Configuration Manager and MSIX Deployment

Configuration Manager has a long and distinguished history of managing Windows devices with application management as one of its many capabilities.

For sure, many of you are aware that managing applications on Windows 10 devices comes in different forms. Configuration Manager on its own provides multiple ways to install MSIX applications on a device, and it is important to point out that there are also cloud-based MSIX application management options that you can explore.

When Configuration Manager is in a hybrid-cloud configuration, known as co-management, it is possible to have Intune handle MSIX applications on the same device.

This section focuses exclusively on distributing MSIX packages with Configuration Manager and its underlying infrastructure. There are two main paths you can follow to publish an MSIX package through Configuration Manager.

However, whichever path you choose depends on how the application source is acquired. This is because with MSIX, if the application is in the Windows Store, then it makes sense to use that solution with Configuration Manager. Otherwise you might have a bunch of application source files that need to be loaded directly into the Configuration Manager.

With that caveat out of the way, the first way to distribute an MSIX package assumes that you have an offline copy of the application that can be loaded into the Configuration Manager for distribution.

While using Windows Store is the preferred method for handling the distribution of modern applications, there are times (e.g., distributing internally repackaged software) when the store model isn't suitable because it is more rigid and that can add significant delays to the publishing process.

Having the MSIX package file offers more control over the delivery of MSIX applications, but considering the following.

Paid apps from the store are not supported for offline installation via Configuration Manager. For repackaged software this is likely not an issue but for purchased software you may need to integrate Configuration Manager with your Windows Store for Business.

Capability	*Offline apps*	*Onlin e apps*
Synchronize app data to Configuration Manager	*Yes*	*Yes*
Create Configuration Manager applications from store apps	*Yes*	*Yes*
Support for free apps from the store	*Yes*	*Yes*
Support for paid apps from the store	*No*	*Yes**
Support required deployments to user or device collections	*Yes*	*Yes*
Support available deployments to user or device collections	*Yes*	*Yes*
Support line-of-business apps from the store	*Yes*	*Yes*

Provision a store app for all users on a device Note 2[65]	*Yes***	*Yes***

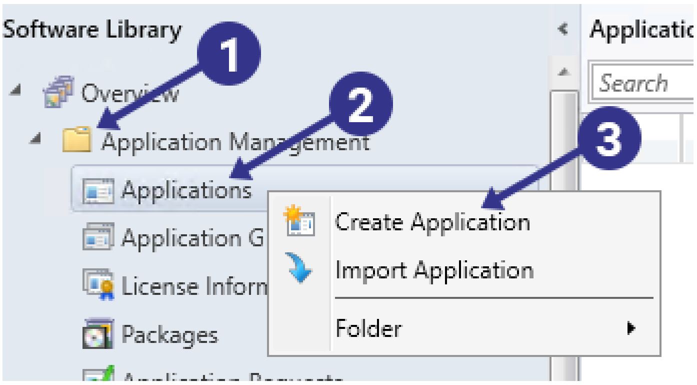
* Support begins with Windows 10 1703
** Requires a minimum of Configuration Manager 1806

Comparison between online and offline support for MSIX capabilities in Configuration Manager

In situations where I need to distribute repackaged software within the organization, you could use the offline installation approach when loading the package directly into the Configuration Manager as an application object with source files.

With this approach, youI can take advantage of Configuration Manager's distribution point servers and trigger the distribution of the package across the network.

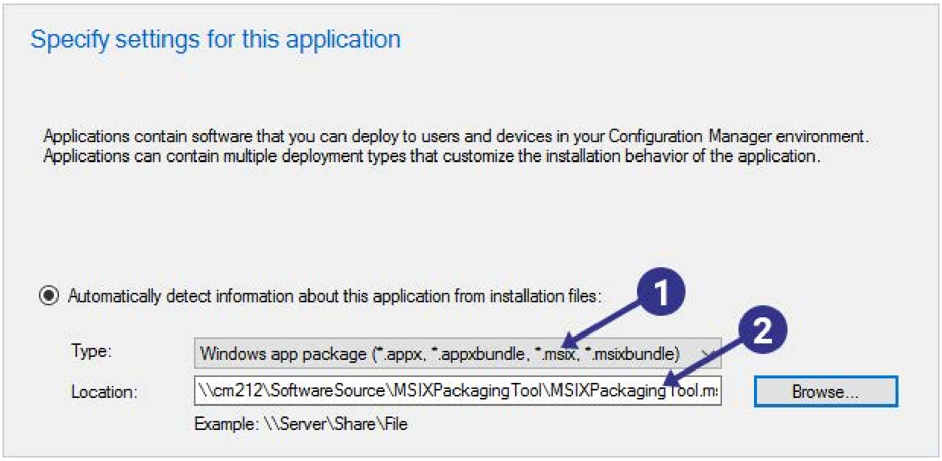
First, go to the Software Library workspace in the Configuration Manager console. Next, expand Application Management, and right click Applications. Finally, select Create Application.



Create an application from the Software Library workspace

[65] https://docs.microsoft.com/en-us/mem/configmgr/apps/deploy-use/manage-apps-from-the-windows-store-for-business#bkmk_note2

On the General page, change the Type field to "Windows app package" and locate the MSIX package you want to install by clicking the "Browse…" button.


Specify settings for this application

Applications contain software that you can deploy to users and devices in your Configuration Manager environment. Applications can contain multiple deployment types that customize the installation behavior of the application.

(●) Automatically detect information about this application from installation files:

Type: Windows app package (*.appx, *.appxbundle, *.msix, *.msixbundle)

Location: \\cm212\SoftwareSource\MSIXPackagingTool\MSIXPackagingTool.m:

Example: \\Server\Share\File

Configure the application Type and MSIX package Location fields

The process is simple for standalone applications. But, for applications that have dependencies, it is more complicated because you need to individually add application dependencies to the Configuration Manager and then link them to the main application object.

Another important thing you need to consider is whether the application is installed for a specific user or for all users.

Note: Be sure to look for the "**Provision this application for all users on the device"** checkbox further on in the creation process.

Traditionally, Configuration Manager uses the application deployment targeting to demine if the application should be installed for all users of a machine or only for a specific user. With MSIX this behavior is controlled by the checkbox.

Many organizations are interested in knowing if they can distribute packages to machines outside of the corporate network. The quick answer is yes -- if they are managed using the Cloud Management Gateway or Intune. The Cloud Management Gateway is where Configuration Manager uses Azure Services to manage clients that are not connected to the corporate network.

Intune can complement Configuration Manager for deploying applications in a Co-Management configuration, which is going to be covered in the next chapter.

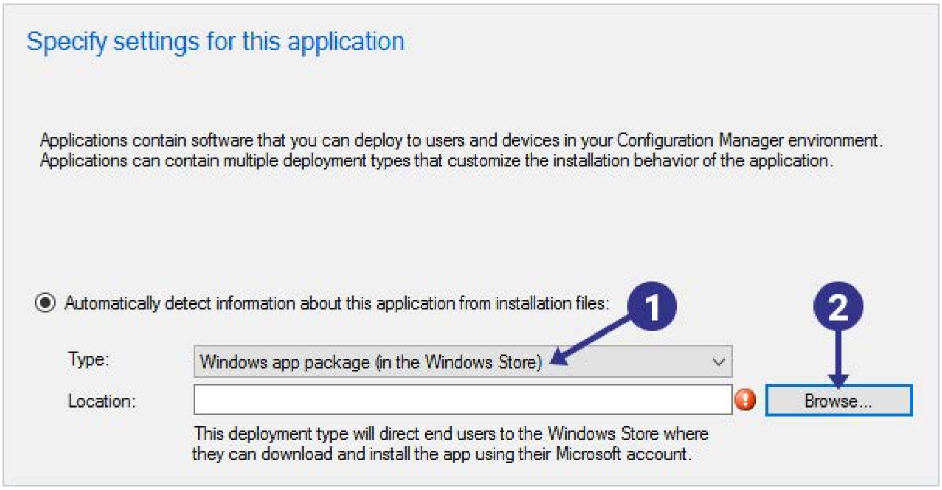
The second way to install an MSIX package with Configuration Manager is to link directly to an application in the public Windows Store. From there, you can download an application with all its dependencies straight into your device using the Windows Store. But this delivery method offers very light management of the application installation.

The application is then delivered to devices by leveraging the Windows Store client on the device and the Windows Store cloud service to distribute the applications to devices.

Here's how to achieve that:

- In the Configuration Manager console, create a new application object.
- Then, from the General page, select “Windows app package (in the Windows Store)” in the Type field.
- Next, select the “Browse…” button and log in with a Microsoft account (not your work account) and search for the application

.


Specify settings for this application

Applications contain software that you can deploy to users and devices in your Configuration Manager environment. Applications can contain multiple deployment types that customize the installation behavior of the application.

Automatically detect information about this application from installation files:

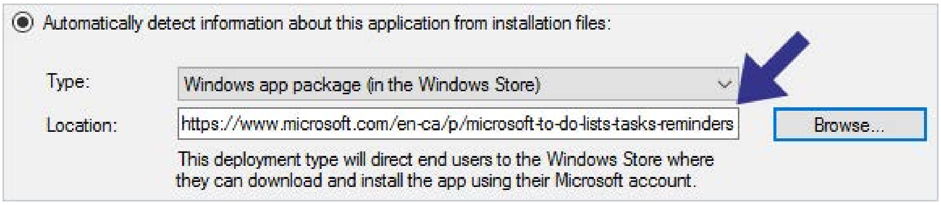
Type: Windows app package (in the Windows Store)

Location:

This deployment type will direct end users to the Windows Store where they can download and install the app using their Microsoft account.

Install an application from the public Windows Store

Once you've selected the application and returned from the Windows Store, the Location field will have a link to the application.


Automatically detect information about this application from installation files:

Type: Windows app package (in the Windows Store)

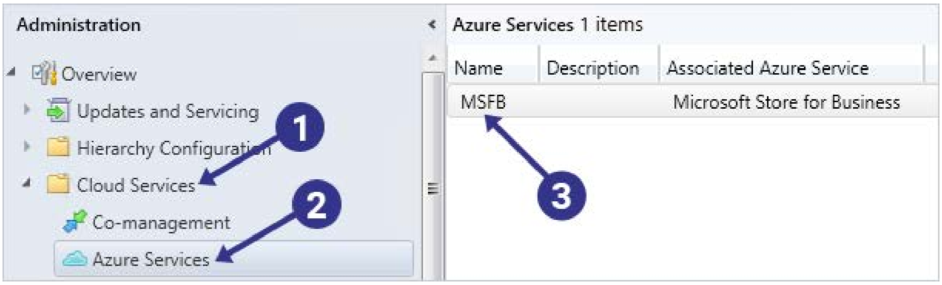
Location: https://www.microsoft.com/en-ca/p/microsoft-to-do-lists-tasks-reminders

This deployment type will direct end users to the Windows Store where they can download and install the app using their Microsoft account.

The Application Location field contains link to Windows Store

In the third case, you can use the private Microsoft Store for Business to distribute an MSIX package. In this scenario the Windows Store for Business is used to acquire and load applications for deployment with Configuration Manager. There are a number requirements that you must adhere to in order to successfully deploy through a private Microsoft Store for Business.

First, Microsoft Store for Business must be added as an Azure Service to Configuration Manager. Second, the synchronization must be active and error free before you can begin. In the image below, Microsoft Store for Business has been configured as an Azure Service[66].



Configure private Windows Store for Business as an Azure Service

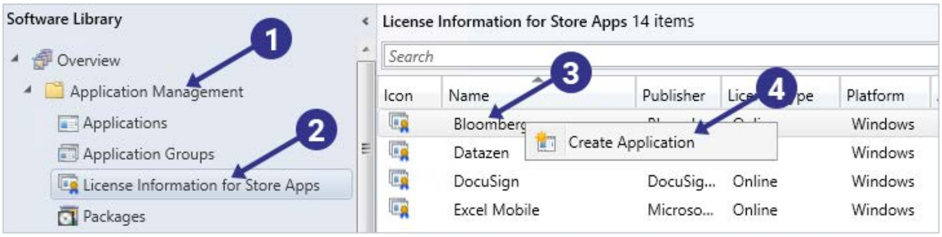
Applications will be brought across into the Configuration Manager Console upon successful synchronization with the store.

Note. When you initially synchronize an application through Microsoft Store for Business, it is classified as a License. Therefore, you look for them in Application Management > License Information for Store Apps.

To finish the process, you need to create an application from the license.

- First, right click a license and select Create Application.

[66] https://docs.microsoft.com/en-us/mem/configmgr/apps/deploy-use/manage-apps-from-the-windows-store-for-business#bkmk_setup



Create an application from an available license in License Information for Store Apps

- The create application wizard will walk you through the steps of creating the application, which is mostly inputting metadata about the application.

Now that you have completely synchronized an application from Windows Store to a private Microsoft Store for Business, expect Configuration Manager to synchronize every 30 minutes.

While this interval is sufficient for most situations, there may be occasions where you need to manually initiate a synchronization for troubleshooting or prototyping purposes. When you need to manually synchronize an application, use the following workflow:

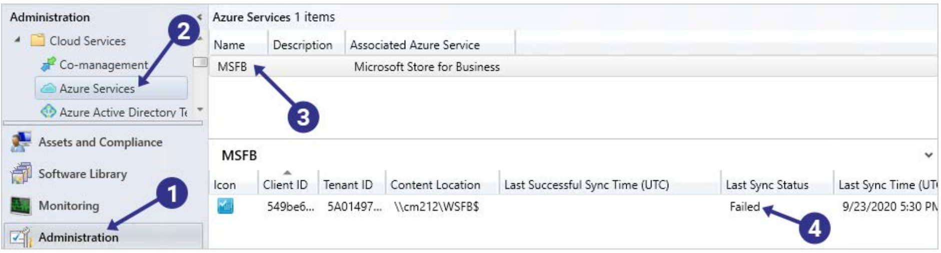
1. Open the Configuration Manager console.
2. Go to the Administration workspace.
3. Expand Azure Services.
4. Right click the Microsoft Store for Business node you already configured.
5. Select **Synchronize with the store**.

Note. Configuration Manager limits manual synchronization to once every ten minutes. If you attempt another synchronization before the 10 minutes have passed, the request will be denied.

Once you have configured applications in Microsoft Store for Business, you are going to need to troubleshoot the synchronization between Windows Store and your Microsoft Store for Business regularly.

The first step is to locate the synchronization status for your Microsoft Store for Business in the Configuration Manager console.

Under Administration > Azure Services, select your Microsoft Store for Business service. The details section will then display the properties of that service, one of which is “Last Sync Status”. Below, we can see a “Failed” synchronization status.



Azure Services 1 items

Name	Description	Associated Azure Service
MSFB		Microsoft Store for Business

MSFB

Icon	Client ID	Tenant ID	Content Location	Last Successful Sync Time (UTC)	Last Sync Status	Last Sync Time (UT
	549be6...	5A01497...	\\cm212\WSFB$		Failed	9/23/2020 5:30 PM

Identify a failed application synchronization in Microsoft Store for Business

If there is a synchronization problem, start your investigation by evaluating the following logs on the site server. You can use the order presented below:

1. WSfbSyncWorker.log
2. SMS_CLOUDCONNECTION.log

It is important to point out that in this scenario with the Windows Store the content for the applications is downloaded to the site server then replicated to distribution points for installation by client devices. The other method that uses MSIX packages with distribution points is when you supply the MSIX file directly to a Configuration Manager application object.

As mentioned at the onset, when the Configuration Manager is in a co-management configuration, applications can be delivered using Intune. In this configuration, the client device can leverage the application management investment that was made with Configuration Manager while having the option of performing application management functions from Intune.

So, let’s proceed to the next chapter where we can explore MSIX packages delivered via Intune.

Using Intune with MSIX

Before diving straight into Intune, let's illustrate the broad support MSIX receives across a wide range of enterprise use cases which includes more purpose-built devices such as the Surface Hub and Hololens.

App type	*LOB: APPX/MSIX*	*MSFB Offline*	*MSFB Online*	*Store Link*
Home	*Yes*	*Yes*	*Yes*	*Yes*
Pro	*Yes*	*Yes*	*Yes*	*Yes*
Business	*Yes*	*Yes*	*Yes*	*Yes*
Enterprise	*Yes*	*Yes*	*Yes*	*Yes*

Education	*Yes*	*Yes*	*Yes*	*Yes*
S-Mode	*Yes*	*Yes*	*Yes*	*Yes*
HoloLens1	*Yes*	*Yes*	*RS4+*	*Yes*
Surface Hub	*Yes*	*Yes*	*No*	*Yes*
WCOS	*Yes*	*Yes*	*Yes*	*Yes*
Mobile	*Yes*	*Yes*	*Yes*	*Yes*

Modern application formats and Intune capabilities

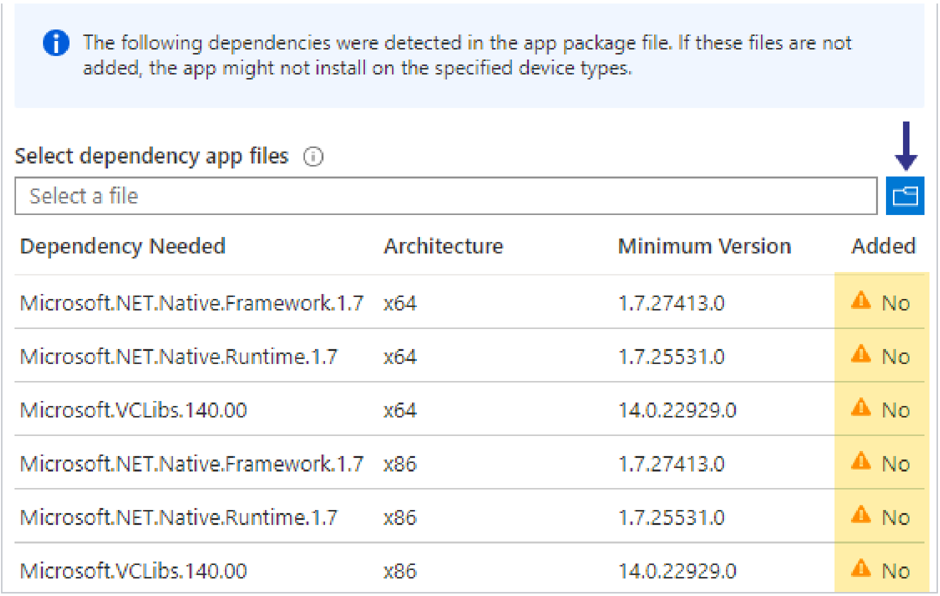
The good news is that Intune provides similar options to distribute MSIX applications as Configuration Manager. In this section, I will cover three key scenarios for installing MSIX applications with Intune:

- MSIX uploaded to Intune
- MSIX from a public store
- MSIX from a private store

That being said, before you can load an application to Intune, you need to consider how the limitations of Intune affect the way you deploy the application. Intune is a cloud-based service with its own unique limitations that must be understood.

The main limitation is that package file size must be less than 30GB. This was a recent improvement over the previous limit of 8GB.

For those cases where you need to upload and distribute your own MSIX package to Intune, you can create a Line-of-business app object in the Intune console. When the package is loaded, it is scanned for dependencies which are then listed in the Intune console.



The following dependencies were detected in the app package file. If these files are not added, the app might not install on the specified device types.

Select dependency app files

Select a file

Dependency Needed	Architecture	Minimum Version	Added
Microsoft.NET.Native.Framework.1.7	x64	1.7.27413.0	⚠ No
Microsoft.NET.Native.Runtime.1.7	x64	1.7.25531.0	⚠ No
Microsoft.VCLibs.140.00	x64	14.0.22929.0	⚠ No
Microsoft.NET.Native.Framework.1.7	x86	1.7.27413.0	⚠ No
Microsoft.NET.Native.Runtime.1.7	x86	1.7.25531.0	⚠ No
Microsoft.VCLibs.140.00	x86	14.0.22929.0	⚠ No

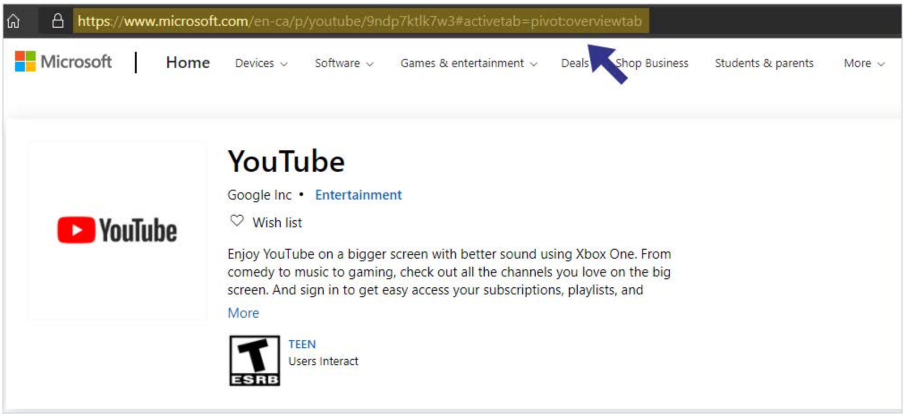
Uploading an MSIX package to Intune automatically detects dependencies

If the reported dependencies are not present in the main package, you should upload them at the prompt, or the application may not run correctly when delivered to the user.

For those cases where you need to create an MSIX application from the public Windows Store, Microsoft has developed a prescribed process[67] for adding a public store application to Intune.

The first step is locating the application in the public store[68] through a browser. Once you've located the application, copy the URL to the application from your browser's URL field and paste it into the Intune console.

[67] https://docs.microsoft.com/en-us/mem/intune/apps/store-apps-windows
[68] https://www.microsoft.com/en-ca/store/apps


https://www.microsoft.com/en-ca/p/youtube/9ndp7ktlk7w3#activetab=pivot:overviewtab

YouTube

Google Inc • Entertainment

Wish list

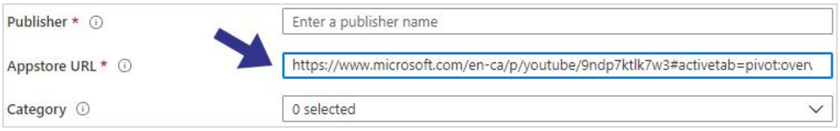
Enjoy YouTube on a bigger screen with better sound using Xbox One. From comedy to music to gaming, check out all the channels you love on the big screen. And sign in to get easy access your subscriptions, playlists, and

More

TEEN
Users Interact

Enter the application URL into the Intune Console

This URL must also be entered into the Appstore URL field of the Intune application object.



Publisher *	Enter a publisher name
Appstore URL *	https://www.microsoft.com/en-ca/p/youtube/9ndp7ktlk7w3#activetab=pivot:overv
Category	0 selected

Enter the Windows Store application URL into the Appstore URL field

Also enter in the publisher field as well since it is a required field.

You can synchronize Intune and Windows Store with Microsoft Store for Business, which makes regular publishing less fragile because it eliminates a lot of data entry errors that occur when creating the application object from the public store.

Additionally, paid applications can be purchased by bulk through the synchronization mechanism, a feature which greatly simplifies license management.

Microsoft has published a workflow[69] to integrate Intune and Microsoft Store for Business and it comes with considerable effort up front to get it working. The good news is that once the configuration is complete, applications that exist in Microsoft Store for Business will automatically replicate to your Intune tenant.

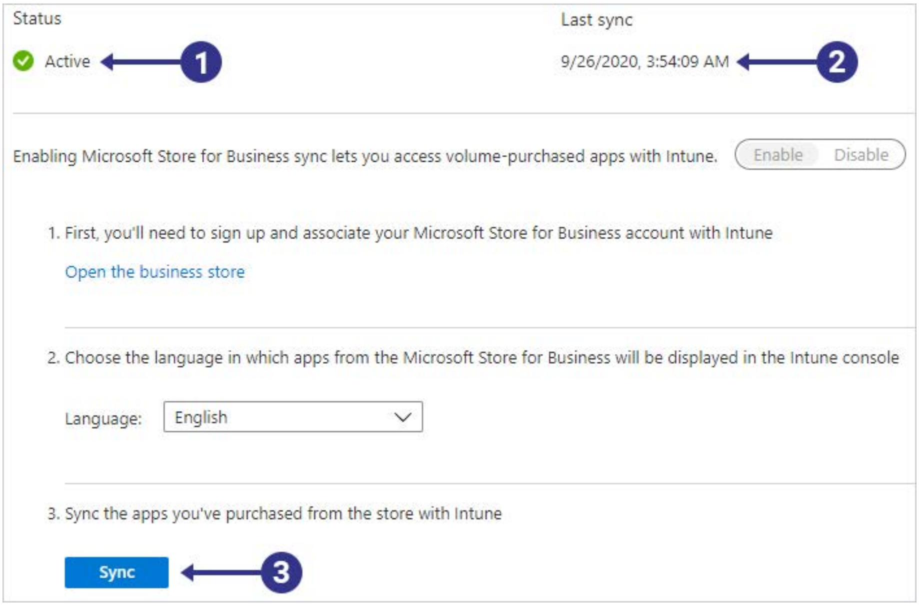
For those situations where you need to manually trigger the synchronization, you can trigger it in the Intune console, like you can with the Configuration Manager.

To trigger a synchronization, open the Microsoft Store for Business blade[70] located in the Tenant administration portal and then open the "Connectors and tokens" page. From there, you can view the status of the synchronization and the last synchronization timestamp (see the figure below).

To sync with Microsoft Store for Business, click the Sync button.

[69] https://docs.microsoft.com/en-us/mem/intune/apps/windows-store-for-business

[70] https://devicemanagement.microsoft.com/#blade/Microsoft_Intune_DeviceSettings/TenantAdminConnectorsMenu/msfb


Status

Active

Last sync

9/26/2020, 3:54:09 AM

Enabling Microsoft Store for Business sync lets you access volume-purchased apps with Intune.

1. First, you'll need to sign up and associate your Microsoft Store for Business account with Intune

Open the business store

2. Choose the language in which apps from the Microsoft Store for Business will be displayed in the Intune console

Language: English

3. Sync the apps you've purchased from the store with Intune

Manually synchronize Intune applications with Microsoft Store for Business

As with Configuration Manager, Intune supports Delivery Optimization in Windows 10 to allow for peer sharing of application content. To enable this functionality, a device configuration profile must be created in the Intune console and targeted at client devices to allow content sharing. Devices on the same local network can then use each other to speed up download times while offloading the Internet connection.

VDI Meets MSIX with App Attach

A common use case seen in some customers is to use datacenter hosted operating systems, either in private data centers or in the cloud. Often these operating systems are set up generically and applications are added dynamically based on the logged on user. The implementation may be either VDI or a shared operating system, but in either case after signing into the OS the end-user must wait for the apps to be ready. MSIX App Attach significantly reduces this wait time, getting the applications into a usable state more rapidly.

App Attach delivers the fastest provisioning experience for MSIX applications in a stateless VDI environment. Some preliminary performance numbers on provisioning time between scripted standard installation of MSIX packages versus pre-release versions of MSIX AppAttach may be seen at Tim Mangan Blog[71].

Keep in mind that the operating system, the packages, plus the user and application state information stores are managed through different techniques to allow for the dynamic composition of a virtual machine when a user logs in, and all parts need to be in place for the user to become productive.

More specifically, App Attach mounts MSIX applications at logon without requiring a full application installation, instead, the application shell integrations are performed to appear installed to the end-user.

When the application is in use, only the required blocks of data are copied to the virtual machine - bypassing a lengthy installation process of copying all the application payload to the virtual machine. Furthermore, the block-level single instance recognition of MSIX avoids

[71] https://www.tmurgent.com/TmBlog/?p=3139

streaming and storing application blocks that are common to other packages.

This approach is recommended because it lowers costs by reducing data storage and improves governance practices by providing a uniform way to install applications across all virtual machines in a pool.

The MSIX applications are attached as *.vhd or virtual hard disk files meaning that the application host operating system must have the Hyper-V feature installed to do this action. Hyper-V can easily be enabled with the following PowerShell command:

```
Enable-WindowsOptionalFeature -Online -FeatureName
Microsoft-Hyper-V -All
```

Importantly, you need to disable four update services that affect applications. The first is Windows Update, which you can disable with:

```
sc config wuauserv start=disabled
```

Second, you need to disable Windows Store updates. To do that, use the “reg” command:

```
reg add HKLM\Software\Policies\Microsoft\WindowsStore /v
AutoDownload /t REG_DWORD /d 0 /f
```

Third, disable the Automatic app update scheduled task with the following two commands:

```
Schtasks /Change /Tn
"\Microsoft\Windows\WindowsUpdate\Automatic app update"
/Disable
```

```
Schtasks /Change /Tn
"\Microsoft\Windows\WindowsUpdate\Scheduled Start"
/Disable
```

And finally, the application host also needs to have Content Delivery auto download disabled.

```
reg add
HKCU\Software\Microsoft\Windows\CurrentVersion\ContentDe
liveryManager /v PreInstalledAppsEnabled /t REG_DWORD /d
0 /f

reg add
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\ContentDe
liveryManager\Debug /v ContentDeliveryAllowedOverride /t
REG_DWORD /d 0x2 /f
```

Once you have completed the above steps, the host operating system is configured, and you can prepare your applications for App Attach. While this can be done through a command line[72], you can use MSIX Hero[73], a freeware utility. The MSIX Hero team have created a graphical MSIX to VHD package conversion utility that works great.

Publishing requires the correct certificates to be present on the application host virtual machines. As such, it is a best practice to place these certificates in the operating system image so that they are immediately present when applications mount.

The virtual machines will require SMB file share access to the VHD files and the computer accounts will require read-only rights. Always try to

[72] https://docs.microsoft.com/en-us/azure/virtual-desktop/app-attach#generate-a-vhd-or-vhdx-package-for-msix
[73] https://msixhero.net/

run the latest version of the SMB protocol to ensure the best performance and security.

To tie everything together requires four final PowerShell scripts that manage the following activities with App Attach.

- A startup script that runs the stage[74] script
- A logon script that runs the register[75] script
- A logoff script that runs the deregister[76] script
- A shutdown script that runs the destage[77] script

Microsoft has guidance on customizing[78] these files to suit your configuration. Once these files have been tested, create a GPO and add the PowerShell files to the various script events and target your virtual machines with it. Because of the special needs of VDI machines they rarely share many of the same Group Policy objects that desktops and laptops would use.

Usually virtual machines predetermined for this role would exist within their own organizational unit in the Active Directory, where all the relevant policies for the device are targeted.

[74] https://github.com/Azure/RDS-Templates/blob/master/msix-app-attach/1.stage-a.ps1
[75] https://github.com/Azure/RDS-Templates/blob/master/msix-app-attach/2-a.register.ps1
[76] https://github.com/Azure/RDS-Templates/blob/master/msix-app-attach/3.deregister-a.ps1
[77] https://github.com/Azure/RDS-Templates/blob/master/msix-app-attach/4.destage-a.ps1
[78] https://docs.microsoft.com/en-us/azure/virtual-desktop/app-attach#prepare-powershell-scripts-for-msix-app-attach

MSIX and App Center

The intention of including App Center in this book is to build awareness for the IT Pro because, especially when developers drive internal application processes, it may not always be clear what tools should be used to prototype applications in an enterprise environment. Sometimes, developers are not necessarily aware of the App Center and how it can be used to help enhance their experience in building a line of business applications.

Microsoft's App Center[79] is a solution for rapidly building, deploying, and testing MSIX and other modern applications. The framework is designed for developers who need to quickly prototype beta code in a production environment while gaining access to important analytics from the application.

At a first glance, many of the management tools for MSIX installation appear to overlap our present goal of managing MSIX packages. Unfortunately, the App Center is not intended as a production solution for application management and should only be used with limited users.

Development and testing with the App Center is encouraged with prototyping application releases. But when your application releases are stable, it's good to sign your code and use one of the other delivery methods available for MSIX packages.

[79] https://appcenter.ms/

How Tos

In this chapter we provide answers to some topics we are frequently asked about but have not yet covered.

How To: Setting up a recapture VM

The recapturing of an application installation needs to be performed in a special well-prepared environment, which is typically a virtual machine (VM) with a snapshot. The VM is always reverted to this snapshot prior to any recapture, with the purpose of ensuring that it is a well-known environment.

The VM should be as clean as possible. This is important because application installers will detect whether dependencies already exist on a machine, so specific components may be left out of the package and not reapplied. Any software added to the VM beyond the operating system could potentially add dependencies. Even though recaptured packages may work fine on initial test systems (with dependencies) - as time passes, it is possible for a package to break when landing on a system without that dependency. Current practices usually try to keep this image to hold just the OS and the required repackaging tools.

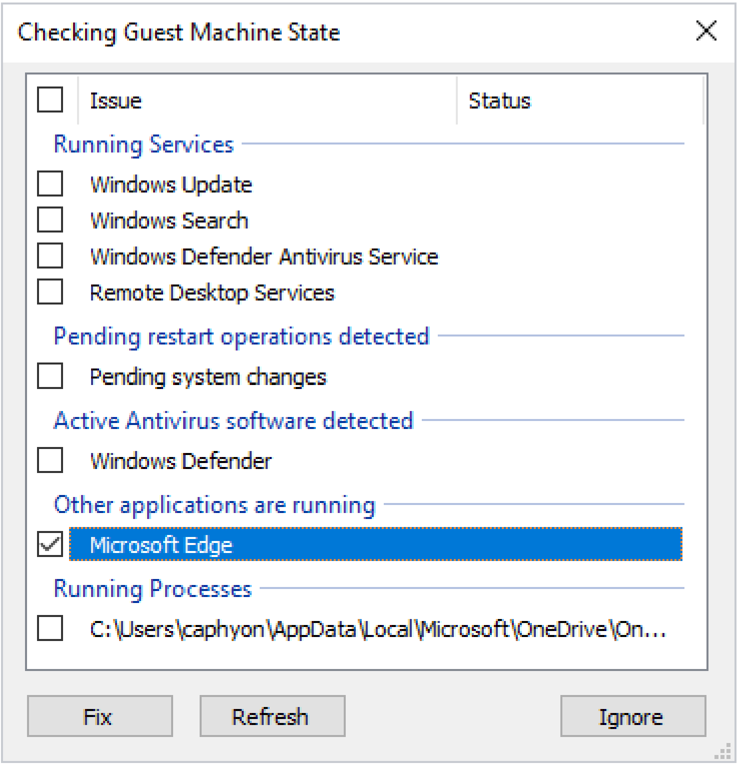
Note: The vendors of these tools generally limit their dependencies.

Often, organizations require anti-malware software to be present on all systems. And unfortunately, anti-malware software adds many dependencies that change every year. So, it is better to use the anti-malware software that is already built into the OS while packaging, then use your preferred vendor scanning tool to detect anything that shouldn't be in the future packages.

Additionally, the OS should be tuned to reduce background activity that could lead to capturing unnecessary (or dangerous) components. This varies depending on the packaging purpose and the versions of the OS, and often, it includes the following steps:

- Turning off unnecessary services, such as Windows Updates or Windows Search, as well as other services that you will not use, (i.e. bluetooth).
- Not joining the OS to the domain, or exempting it from the Group Policy.
- Turning off Windows Apps updates.
- Disabling anti-malware definition updates.
- Pending system changes

Some repackaging tools, like Advanced Installer (and others), automatically check for these rules and assist you in enforcing them - while also managing the VM snapshot states automatically for each new package.



Checking Guest Machine State

	Issue	Status
	Running Services	
☐	Windows Update	
☐	Windows Search	
☐	Windows Defender Antivirus Service	
☐	Remote Desktop Services	
	Pending restart operations detected	
☐	Pending system changes	
	Active Antivirus software detected	
☐	Windows Defender	
	Other applications are running	
✓	Microsoft Edge	
	Running Processes	
☐	C:\Users\caphyon\AppData\Local\Microsoft\OneDrive\On…	

Checking a VM state - Advanced Installer

How To” Common packaging scenarios

Installing a font

Starting with the 2004 version of the Operating System, MSIX packages may now include fonts for use on the system the application package is deployed to. These changes are supported by changes to the AppXManifest schemas as part of the Uap4 schema extensions.

Deploying fonts, or any other system resource, with an MSIX package, requires entries in the AppxManifest.xml for each of the fonts. Even when you have a recaptured and fully installed font in the package, the font will not be available to use by any application without a declaration in the AppXManifest file.

Installing a package that contains a font declaration instructs the OS to register the font using the Windows Font Manager, making it available for any application installed on the system.

Note. Removing that package will remove the font. Thanks to single instance storage, if the same font is added by two different packages, this is correctly handled.

For example, to add a font named “Radiant”, the font file must be present in the package, and the following declaration has to be included in the manifest as part of one of the “Application” elements:

```
<Applications>
   <Application Id=...............................>
      <Extensions>
         <uap4:Extension Category="windows.sharedFonts">
            <uap4:SharedFonts>
               <uap4:Font File="VFS\Fonts\Radiant.ttf"/>
            </uap4:SharedFonts>
         </uap4:Extension>
      </Extensions>
   </Application>
</Applications>
```

Although the fonts are added on the basis of the package, you must still add this within one, and only one, of the Application elements. It does not matter which application you apply the font references to.

Currently, the MSIX Packaging tool ignores captured fonts contained in the package. When using this tool, PsfTooling can detect and provide the syntax that you would need to manually add it to the Manifest.

With Advanced Installer Express, this can be done in just a couple of clicks. Just go to Declarations view and configure your font, this will ensure your package manifest is correctly generated.

For full details and a tutorial video, check Advanced Installer Blog - Installing Fonts in MSIX Package[80].

[80] https://www.advancedinstaller.com/install-fonts-msix.html

“Path” Environment Variable replacement

The most common scenario for using an environment variable is when an application wants its main executable path to be added to the “Path” environment variable.

A newer replacement technique, which is also used to help a process find folders containing DLLs, is the App Paths registration in the Windows Registry. These techniques allow a process to be started or DLLs to be loaded without direct knowledge of the folder they are in.

MSIX packages have no direct support for any environment variables, and the App Paths registration is not recognized either. But for scenarios like the one above, to help with the DLLs, the Package Support Framework has the DynamicLibraryFixup. There is a new EnvVarsFixup to support environment variables when inside the container, but this might not be appropriate for the Path variable changes.

Fortunately, there is another predefined solution to allow finding an EXE by name only introduced by the MSIX Manifest. You can also achieve this by declaring an execution alias for the application, in the AppxManifest.xml file.

```
<Applications>
   <Application
EntryPoint="Windows.FullTrustApplication"
Executable="AI_STUBS\AiStub.exe"
Id="EnvironmentVariables">
      <Extensions>
         <uap5:Extension
Category="windows.appExecutionAlias"
```

```
EntryPoint="Windows.FullTrustApplication"
Executable="HelloWorld.exe">
            <uap5:AppExecutionAlias>
               <uap5:ExecutionAlias
Alias="HelloWorld.exe" />
            </uap5:AppExecutionAlias>
         </uap5:Extension>
      </Extensions>
   </Application>
</Applications>
```

If you're using the Microsoft MSIX Packaging tool, this would be something that you will need to manually add to the manifest. However, if you also need the PSF and are using PsfTooling then PsfTooling will take care of this for you automatically by different means.

With Advanced Installer Express, this can be done in a few clicks. After adding the executable in your package, you can simply go to Declarations view in Advanced Installer and declare your execution alias.

For full details, check out this article Advanced Installer Blog - MSIX Environment Variables[81]

Context menus

Many desktop apps offer a context menu. A context menu is a pop-up menu that appears, for example, when a user right-clicks on a file. There are two types of context menus:

[81] https://www.advancedinstaller.com/msix-environment-variables.html

- **Shell menus** - which point to the application's exe that you want to run (with or without parameters)
- **Shell extensions** - which point to a dll.

Adding a context menu to your application was easy with MSI. With MSIX, Microsoft added a bit of complexity, and there are still some missing pieces in some scenarios. Hopefully, Microsoft will address these obstacles in the near future.

Read more about it here: Advanced Installer Blog - MSIX Context Menu[82]

allowElevation capability

If your Win32 application needs to ask for user elevation upon launch, then you need to set the allowElevation capability in your AppxManifest.xml.

```
     <Capabilities>
             <rescap:Capability Name="runFullTrust">
             <rescap:Capability Name="allowElevation">
     <Capabilities>
```

Setting this flag alone in the package manifest will not enable your application to request for elevation. As its name implies, this flag only allows your application to request an elevation. How does an application request elevation rights? It does so by setting the execution level to “**requireAdministrator**” from the app’s main EXE manifest.

[82] https://www.advancedinstaller.com/msix-context-menu.html

If your application does not have this value set in its EXE manifest, then the "allowElevation" capability will simply be ignored and the application will not try to automatically elevate when the user launches it. The user can still manually run it as an admin by right clicking the application and selecting "Run as administrator'.

You can find more insights on this article: Advanced Installer Blog - How to set AllowElevation flag for MSIX packages[83]

Note: Starting with Windows 10 20H1 this capability seems to no longer be required for sideloaded applications.

Start Menu Entries

Compared to the MSI shortcuts concept, MSIX shortcuts make no exception when it comes to Microsoft's approach. When an MSIX application is installed, it does not create a .lnk file as we are used to. Instead, it creates only a Start Menu application entry.

Start Menu entries are managed through the AppxManifest.xml and you can find an example of how the Application section should look like below.

```
<Application Id="VLC"
     Executable="VFS\ProgramFiles\VideoLAN\VLC\clv.exe"
     EntryPoint="Windows.FullTrustApplication">
```

[83] https://www.advancedinstaller.com/allow-elevation-msix-packages.html

```
     <uap:VisualElements BackgroundColor="transparent"
                                 DisplayName="GP"
                                  ................
    </uap:VisualElements>
  </Application>
```

Get more details here:
advancedinstaller.com/msix-shortcut[84]

Startup Applications

Registry entries under "Run" key or shortcuts placed in the "Startup" folder are no longer a viable solution for MSIX packaged applications.

To configure an application to launch at startup, you need to define a "StartupTask" in the package AppxManifest.xml, as shown below.

```
  <Applications>
       <Application Id="TheApp" Executable="TheApp.exe"
                EntryPoint="Windows.FullTrustApplication">

          <Extensions>

            <desktop:Extension
              Category="windows.startupTask"
              Executable="VFS\TheApp.exe"
              EntryPoint="Windows.FullTrustApplication"/>
              <desktop:StartupTask TaskId="TheApp"
               Enabled="true" DisplayName="The App"/>

           </Extensions>
```

[84] https://www.advancedinstaller.com/msix-shortcut.html

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

Get more details here: Advanced Installer Blog - The new way of dealing with Startup Application in your MSIX package[85]

Disabling Files and Registry Virtualization

The MSIX container provides default virtualization for any files found under the VFS folder and to any registry entry found under a registry path included in the registry.dat hive from the MSIX package.

To disable the registry and file redirections, the following properties must be added in the AppxManifest.xml:

- **desktop6:RegistryWriteVirtualization**: Indicates whether virtualization for the registry is enabled for the desktop application.
- **desktop6:FileSystemWriteVirtualization**: Indicates whether virtualization for the file system is enabled for the desktop application.

The above properties should be included inside the Properties element of the manifest. For example:

```
<Properties>
      <desktop6:FileSystemWriteVirtualization>disabled</
      desktop6:FileSystemWriteVirtualization>
```

[85] https://www.advancedinstaller.com/startup-programs-msix.html

```
      <desktop6:RegistryWriteVirtualization>disabled</
      desktop6:RegistryWriteVirtualization>
</Properties>
```

These two capabilities appear to be intended only for debugging purposes and are unlikely to be useful for production packages.

Additional details on these settings may be found at Advanced Installer - MSIX Disable Registry File Redirection[86].

[86] https://www.advancedinstaller.com/msix-disable-registry-file-redirection.html

Going Forward

In 1999 Microsoft delivered the first MSI packages. Since then, Windows Installer has become the pillar of application packaging and deployment. The MSI is still the most used packaging technology for Windows applications, but the evolution of operating systems has outpaced the, mostly-ignored, Windows Installer technology. MSIX is well positioned to become the replacement packaging and delivery vehicle going forward.

Within modern OSes containerization is no longer an obscure topic. For more than a decade App-V has been the #1 virtualization solution for enterprise environments, accompanied by ThinApp and others (many discontinued).

OS-es for mobile devices have been running virtualized apps from day one. Starting from a clean slate was a big advantage for iOS and Android, since they didn't have to go through all the trouble of helping developers migrate apps to a new packaging and runtime environment, as MSIX has to do today, with millions of Win32 and .NET applications.

As with every new technology, caution is advised. MSIX is still developing, there are many challenges to tackle and a detailed evaluation and planning process should be executed when shifting to MSIX[87].

[87] https://www.advancedinstaller.com/msi-vs-msix.html

This will be a long journey, but with important benefits for those that choose to adopt MSIX. And each year, this list of benefits keeps growing, for both ISVs and enterprises.

The MSIX story is not complete today. Upending decades of well-established formats and runtimes is not easy. And planning for what we will need for the next decade or two takes careful consideration. Microsoft chose not to design it all up front and give us the complete technology that MSIX will become. Instead, they delivered a part of what we know we need now, and plan to add to it over time.

You have an opportunity to contribute to this plan. Please join your colleagues by participating in the discussions in the MSIX Technical Communities[88] and contribute your ideas.

[88] https://techcommunity.microsoft.com/t5/msix/ct-p/MSIX

MSIX Packaging Fundamentals