Flexera
Inform IT. Transform IT.™

revenera™

A Developer's Guide to MSIX

Timothy Mangan
Hassanul Banna Kazhungil
Varaprasad Reddy Mallam
Kiran Mantagi

Copyright © 2021 by Flexera, Revenera and TMurgent Technologies, LLP

Table of Contents

Foreword 1

About the Authors 2

Chapter 1: Introduction 3

Chapter 2: MSIX and PSF 5

2.1 How MSIX is different for you and your current application 5

2.2 Package Support Framework 7

2.2.1 PSF theory of operation 8

2.2.2 PSF features 9

2.2.3 Common MSIX container issues: how to avoid, or use, the PSF to solve them 10

2.2.4 Adding PSF components to your package 13

2.3 Windows App SDK (aka Project Reunion) 13

Chapter 3: Overview of an MSIX Project 16

3.1 Digital signing 16

3.1.1 Signing an MSIX package 17

3.1.2 How to sign an MSIX package 18

3.1.3 Time stamping 20

3.1.4 Debug signing errors 21

3.1.5 Maintain the package identity across the digital certificate 21

3.2 Package components 22

3.2.1 Ingredient files 22

3.2.2 Files typically generated as part of the build process 26

3.3 MSIX build process 26

3.3.1 Command line tools for MSIX from the SDK 27

3.3.2 Creating resources.pri using MakePRI.EXE tool 27

3.3.3 View resources.pri contents using MakePRI.EXE 28

3.3.4 Building MSIX package using MakeAppx tool 29

3.4 MSIX bundles 30

3.4.1 Building bundle packages 30

3.4.2 Signing the bundle package 31

3.4.3 Flat bundle package 31

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

3.5 Modification packages 32
3.6 Shared package containers 32
3.7 AppInstaller (XML) file 33
3.8 MSIX core 35
3.9 Basic AppXManifest structure 37
Chapter 4: Project Activities 44
4.1 The existing EXE and DLL projects 44
4.2 The question of end-user identity 45
4.3 User settings 46
4.3.1 Assessing your current application 46
4.3.2 Handling application settings 46
4.3.3 Handling data 47
4.3.4 Installation time customization 48
4.3.5 Using the PSF scripting capabilities for customization 48
4.3.6 Coded logic in your app initialization for customization 48
4.3.7 Considerations for customization 48
4.3.8 Modification package 49
4.3.9 New UAP setting APIs 49
4.4 Registry settings and data 50
4.4.1 Functional differences 50
4.4.2 Migrating to the dat 50
4.4.3 Creating the dat file(s) 51
4.4.4 Considerations for what goes into the Registry.Dat 51
4.4.5 File string references in the registry application hive 53
4.5 Start menu 54
4.5.1 Differences in starting modern apps 54
4.5.2 Assessing current start menu needs 54
4.5.3 Visual elements 55
4.5.4 Assets and the start menu 56
4.5.5 Splash screens 57
4.5.6 Rotation preference 57
4.5.7 Migrating existing installed app shortcuts 57
4.5.8 Examples of AppXManifest replacement 59

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.6 File type associations 62
4.7 Protocol handlers 64
4.7.1 Assessing your existing application 64
4.7.2 Changes to support protocol handlers in MSIX 65
4.7.3 Example of integrating the protocol handler 65
4.7.4 com-based protocol handlers 67
4.7.5 Protocol handler references 67
4.8 COM support 68
4.8.1 Traditional vs. Packaged apps and COM 68
4.8.2 Packaged COM in MSIX 69
4.8.3 Assessing your current installer 70
4.8.4 Integrating your COM into MSIX 71
4.8.5 Additional COM limitations 85
4.9 Services 85
4.9.1 Assessing your current installer 86
4.9.2 Integrating into your new package 87
4.10 Firewall configuration 87
4.10.1 Assessing your current installer 88
4.10.2 Integrating into your new package 88
4.10.3 Microsoft Firewall References 89
4.11 Fonts 90
4.11.1 Assessing your current installer 90
4.11.2 Integrating into your new package 90
4.11.3 Runtime differences 91
4.11.4 Font references 92
4.12 Environment variables 93
4.13 Event logging 94
4.13.1 Assessing your current installer/program 94
4.13.2 Options for logging in your new package 94
4.13.3 Option: Initialization Code Check 96
4.14.4 Option: Provider and Log Registration or Update 98
4.14 Other MSIX limitations 101
4.14.1 In-process shell extensions and COM DLL servers 101

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.14.2 Targeting older Windows operating systems using MSIX core 101
4.14.3 Targeting Windows server 2019 102
4.15 Custom install actions 102
4.16 Package Updates and AppInstaller File 104
Chapter 5: Setup Project 107
5.1 Step-by-Step instructions to build MSIX package with InstallShield MSIX project 107
5.2 Step-by-Step instructions to build MSIX package with Windows Application Packaging project 114
5.3 Guidance on Files in your package 118
Chapter 6: Testing and Debugging 120
6.1 Handling installation errors 120
6.2 Digital code signing and validation errors 122
6.3 Runtime Errors 122
6.3.1 Process Monitor 122
6.3.2 Package Support Framework 123
6.3.3 Starting a process inside the container 123
6.4 How do I know if a process is running inside a container? 125
6.5 Debug the app with Visual Studio 127
6.5.1 Using F5 127
6.5.2 Using Installed app Packages 127
6.5.3 Using WaitForDebugger and AttachToExistingProcess 127
Chapter 7: AppxManifest Schema 129
7.1 Schemas supported 130
7.1.1 Foundation schemas 131
7.1.2 UAP schemas 131
7.1.3 Desktop schemas 132
7.1.4 COM schemas 132
7.1.5 WindowsCapabilities schemas 133
7.1.6 RestrictedCapabilities schemas 133
7.1.7 Virtualization schemas 134
7.1.8 CloudFiles schemas 135
7.1.9 Preview schema sets 135
7.1.10 Additional schema sets 135

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.1.11 Unofficial schema sets 135
7.2 Examples of how to read a schema file 136
7.2.1 Types in schema 136
7.2.2 Abstraction, choice, and substitution group 137
7.2.3 Schema conclusion 139
7.3 Identity Element 140
7.3.1 Schema source 140
7.3.2 Microsoft documentation 140
7.3.3 Element attributes 140
7.3.4 Sub-elements 141
7.3.5 Implementation notes 141
7.3.6 Example 141
7.4 Properties element 142
7.4.1 Schema source 142
7.4.2 Microsoft documentation 142
7.4.3 Element attributes 142
7.4.4 Sub-elements 142
7.4.5 Implementation notes 145
7.4.6 Example 146
7.5 Resources element 147
7.5.1 Schema source 147
7.5.2 Microsoft documentation 147
7.5.3 Element attributes 147
7.5.4 Sub-elements 147
7.5.5 Implementation notes 148
7.5.6 Examples 148
7.6 Dependencies element 154
7.6.1 Schema source 154
7.6.2 Microsoft documentation 154
7.6.3 Element attributes 154
7.6.4 Sub-elements 154
7.6.5 Implementation Notes 158
7.6.6 Example 161

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.7 Capabilities element 162
7.7.1 Schema source 162
7.7.2 Microsoft documentation 163
7.7.3 Element attributes 163
7.7.4 Sub-elements 163
7.7.5 Implementation notes 166
Example 168
7.8 Package extensions element 169
7.8.1 Schema source 169
7.8.2 Microsoft documentation 169
7.8.3 Element attributes 170
7.8.4 Sub-elements 170
7.8.5 Implementation notes 171
7.8.6 Examples 172
7.9 Applications element and application element 178
7.9.1 Schema source 178
7.9.2 Microsoft documentation 178
7.9.3 Implementation Notes 178
7.9.4 Examples 179
7.10 Applications: VisualElements element 182
7.10.1 Schema source 182
7.10.2 Microsoft documentation 182
7.10.3 Element attributes 182
7.10.4 Sub-elements 183
7.10.5 Implementation notes 184
7.10.6 Example 186
7.11 Application: ContentURI element 187
7.11.1 Schema source 188
7.11.2 Microsoft documentation 188
7.11.3 Example 188
7.12 Application: Extensions element 189
7.12.1 Schema source 189
7.12.2 Microsoft documentation 190

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.12.3 Element Attributes 190
7.12.4 Sub-elements 190
7.12.5 Implementation notes 195
7.12.6 Examples 195
7.13 BuildMetadata element 205
7.13.1 Schema source 205
7.13.2 Microsoft documentation 205
7.13.3 Element information 205
7.13.4 Example 206
Chapter 8: PSF Json Syntax 207
8.1 About Json 207
8.2 General items 209
8.3 Use of RegEx patterns 212
Chapter 9: Bundle Schema 213
Chapter 10: PowerShell Cmdlets for MSIX 215
10.1 Add-AppxPackage 215
10.2 Remove-AppxPackage 215
10.3 Get-AppxPackage 216
10.4 Get-AppxPackageManifest 216
10.5 Invoke-CommandInDesktopPackage 216
10.6 Shared package container cmdlets 217
10.7 PowerShell references 217

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Foreword

By: Matteo Pagani, Windows App Consult Engineer, Microsoft

How many times did you try to install an application, which was needed to perform a critical task, and it failed? Or how many times, after uninstalling an application, have you discovered that Windows wasn't any more reliable than it was before? Or, again, you installed an application just to find out that other critical applications installed on your machine stopped working. Sounds familiar?

Let's face it, applications' packaging and deployment is one of the biggest challenges to deal within the Windows desktop ecosystem. It doesn't matter if the application is well developed or if it helps users to perform a critical task. If your customers can't install it on their machine in an easy and reliable way, they won't use it or they will put your customer support under heavy pressure to get help.

Delivering a reliable installation experience is as important as creating a good user interface or building a stable application! However, sometimes, with the existing technologies it isn't always easy to come up with a good solution.

These are the reasons that led Microsoft to introduce MSIX, a new packaging technology based on three principles: simple, safe and reliable. "Wait a second," you might say, "I have already heard about MSIX. Isn't it a technology dedicated to IT professionals and system administrators?" Yes, indeed, MSIX helps to solve many of the challenges that IT Pros have faced over the years when it comes to packaging and deploying applications for enterprises: it provides a clean install and uninstall process; it enables to split the application and the enterprise customizations in independent packages; it supports the ability to deploy applications via enterprise tools like Microsoft Endpoint Manager or even to the cloud, thanks to Azure Virtual Desktop and MSIX App Attach.

But what if you are a developer? Is there any value for you in adopting MSIX? The answer is yes, of course! Tim Mangan, one of the biggest MSIX world experts, together with the people who are building one of the most popular tools to build software packages, InstallShield, will guide you through a journey to explore all the opportunities that this technology has to offer to you. From creating MSIX packages directly in Visual Studio to integrate Windows features like file type association directly in the manifest; from optional packages to managing updates with App Installer to solving common developer issues with the help of the Package Support Framework.

I encourage developers who are building Windows desktop applications, no matter the development platform of your choice, to read this book and to start applying all the learnings in your everyday work! Thanks to MSIX, you can provide a more streamlined installer to your users, which is more respectful of their machines and leads to fewer support calls; you can make the life of IT Pro easier, by empowering them to build a modern IT ecosystem and deploy apps and updates faster and more frequently; you can improve the user experience by providing features like automatic updates, differential updates and disk space optimization without having to customize your code or the packaging process.

Happy reading!

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

About the Authors

Timothy Mangan

Runs TMurgent Technologies, LLP a small partnership offering tools, training, and consulting around Microsoft's application virtualization and containerization platforms, App-V and MSIX.

Is the lead author of several books, including MSIX Packaging Fundamentals for the IT Pro Community. Tim is also the primary outside contributor to the Package Support Framework, a GitHub C++ based library to help intercept and fix common issues that older code may have in running inside the MSIX container. Reach out to him at tmangan@tmurgent.com.

Hassanul Banna Kazhungil

Software Architect for InstallShield

Has 15 years of experience in software engineering, application packaging and deployment technologies. Passionate in Microsoft technologies and packaging applications. He is contributing to InstallShield MSIX feature development, authored InstallShield Knowledge Base articles on MSIX, also contributing to the InstallShield community.

Varaprasad Reddy Mallam

Engineering Lead and Software Developer for InstallShield

Has hands-on experience in designing and developing large scale windows desktop products. He has developed both MSI and MSIX features for InstallShield, as well as contributed to InstallShield Community.

Kiran Mantagi

Product Manager for AdminStudio at Flexera

Has 15+ years of experience in the software industry. His expertise in application compatibility, packaging, virtualization and deployments has helped him drive AdminStudio to solve IT Pros real-time problems managing software and updates in their environment. His understanding of traditional software packages and their various issues has made him curious to learn about MSIX and how it solves these legacy issues, since its inception.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 1: Introduction

Here is a question:

> *How can you write a book aimed at developers and include almost no code?*

Here is the answer:

> *You write a book for developers who already have code and don't want to change it much to keep it modern. And you put the code in GitHub!*

This sums up the purpose of this book. To enable you, the developer, to retain as much code as possible as you modernize your application into the new, modern world of containerized packages. **We are going to explain what is going on, provide the missing "why" that is lacking in most of the Microsoft documentation, and provide a framework for you to create a project to modernize your application, along with appropriate reference material to pull it off.**

MSIX is the new, modern replacement for the Microsoft Software Installer (MSI) and setup EXE installer technologies that we have been using for decades. At the same time, it is a new, modern runtime environment intended to protect both your application and end-user systems.

Additionally, when it is properly built, your enterprise customers will prefer this new format as it will reduce the amount of work that their IT departments must do to prepare your application package for delivery to hundreds or thousands of end-users (you know, the ones who really pay the bills).

We wrote this book as a free guide to help software developers and vendors move to this new format. Written by the acknowledged world expert on MSIX, Tim Mangan, along with the co-authors from Flexera and Revenera who build the go-to tool for building software packages, InstallShield, this book provides an up-to-date reference on everything you need to move your existing projects to this new and exciting format no matter what tools you use to build.

The assumption through much of this book is that you have an existing product that is built and delivered as a Win32, Dot Net Framework or Universal Windows Platform (UWP) package today, and you want to modernize to MSIX. But even if you are starting a new project from scratch, the book should serve as an excellent reference.

We will first cover some of the basic concepts about running code in an MSIX container and how the Package Support Framework (PSF) can help solve some common issues faced by developers coming from non-containerized deployments.

Next, we'll introduce you to your new MSIX project to create your new project, and in the chapter following that step, we'll walk you through many of the project tasks that you'll need to take on to convert your existing product to MSIX.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The remaining chapters of the book serve as a reference, especially for the all-important, yet scantily documented, AppXManifest file.

MSIX is a work in progress, and we wrote this book for the community as a snapshot in time to help people get started.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 2: MSIX and PSF

The tradition of Win32 and Dot Net Framework applications is that they install and integrate directly into the operating system and run co-dependently with all other applications installed onto the same system. This tradition is the source of the strength of Windows as a platform and is why it is the dominant OS worldwide.

But while an asset to the OS, this unfettered integration has a dark side too. These applications are allowed to break other applications, and, more important to you, other applications are allowed to break yours! If you're careless, an application can also break the OS itself. While practices and security measures have been implemented over the years to mitigate some of that, the stability and security of our computer systems remain a huge problem.

And then there is the issue of “Windows Rot”a term used to describe when performance on the OS degrades over time because of the repeated installation, execution, and unclean uninstallation of applications.

When Microsoft created UWP and its new application model, one of the goals was to address the dark side issues. This model allows applications to co-exist on the OS side by side, but using an isolation model backed by the lightweight version of Windows Containers keeps applications from interfering with each other and with the OS. By using a prescriptive installation technology to avoid Windows Rot, it also addresses performance.

Of course, that wasn’t the only goal for UWP, and a rewrite of the old Windows API into a small, portable subset came along to make these applications run cross-platform. And while it works well for applications that can live with the restrictions, many developers, especially vendors producing applications for consumption by the business community, found UWP too restrictive for their needs.

2.1 How MSIX is different for you and your current application

MSIX represents a middle ground for application developers. It still runs in a container, but with an improved application model that is far less restrictive than that of UWP. Although Microsoft calls this new application model the “Full Trust” model, don’t be fooled into thinking the cure is to let your application do whatever it wants. It could have been called the “Way more trust than UWP” model if the name wasn’t too long.

But MSIX also lets you code in your old ways, whether it be unmanaged code using Windows Forms and calling the Win32 APIs or managed code using XAML and Dot Net Framework versions of the APIs. Sure, there are a slew of things you’ll need to do differently (or remediate) to be able to run inside the MSIX container, and we’ll help you with a bunch of those, but the bulk of your existing code will be just fine.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

MSIX provides these key benefits to your customers, which is why they want you to migrate to this new form:

- Guaranteed clean installs and uninstalls.
- Isolation between applications (except when we don't want it).
- A higher level of security, both for your customer and to protect your application as well.
- Single-instance storage: essentially an automatic block level deduplication for any file in any installed package.

To make isolation and integration work better hand in hand, the ways that applications integrated into the OS had to change. Your installer can no longer actually write to a lot of common locations that you are used to writing to. Instead, your installation logic is descriptive, contained in an XML file called the AppXManifest file, and the installation technology for MSIX, called AppInstaller, will take care of the details on how to perform the integration.

Often, this is done in different ways than in the past.

Here is a simple example. In the past, your installer may have written to one of the Classes keys of the registry to register a new File Type Association. Instead, now you describe the file type association in the AppXManifest, and the package installation will write that data into a different location in the registry that is specific to your package. They modified the OS code in the shell to look not only at the traditional locations but also at those for these installed packages. The same thing goes for COM, Shell Extensions, Protocol handlers, and even Shortcuts. By not commingling these types of installation activities with those of other applications, this makes conflict resolution and removal cleanup a snap.

Another feature that MSIX uses is file and registry layering. Your package files include a folder called VFS, which stands for Virtual File System. This is the place for any files in your package that are intended to be seen as being in a common folder location. For example, many programs might want to drop some DLLs into the System32 folder. Instead (assuming an x64 program for an x64 OS), you would simply place these files in a folder called "VFS\SystemX64" inside your package. Any process running inside your container that asks to see the C:\Windows\System32 folder will then see this folder layered on top of the current System32 folder.

From the registry perspective, none of your registry entries, either HKLM or HKCU, will actually be written into the registry. Instead, these registry items exist in a separate "application hive" that is organized with a machine and user section that overlays onto the real registry for any process running in your container.

So while the location of files and registry items might actually be implemented differently on the system than you are used to, most of the time your code can continue to reference the items as they used to.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Where you will run into problems is when you try to change things. Your application assets are completely protected, and nothing, not even your code, can actually overwrite them. There are various techniques to allow some things to be written to; some of these are automatic and some require configuration. We will discuss how a bit more later on, but first understand what happens when things change.

When changes are allowed, the changes will not overwrite your package definition, but instead these writes will be redirected to per-user locations associated with your package (in essence, a redirection to store in a new folder in the user's LocalAppData folder). This not only protects your application assets from corruption by the end-user (or other malware), but it supports a package-level repair feature that resets the application to its original installed state simply by clearing out the redirection area. Software, even when running elevated or at the root system level, cannot corrupt your application files. And should someone somehow find a way, the system would still detect the corruption via block-level hash detection.

So what can the app change? By default:

- Files in the package are immutable, meaning they cannot be changed, overwritten, or deleted.
- HKLM is immutable.
- HKCU writes will be redirected.
- HKCU will not allow requests for "Full Access" or any access that includes deletion rights or change owner writes.

These may be extended by one of the following:

- The Package Support Framework (PSF) can be added to your package and configured to allow file redirection and certain registry fixups. Almost everything you need from a file perspective is covered, and currently no support for writing to HKLM is provided.
- There is a new AppXManifest option, called InstalledLocationVirtualization, that provides some file redirection as well. It is intended to allow you to have user-modifiable configuration files in the program folder; however, this feature is only available on some newer OS builds, so you might not want to depend on it yet.
- Change your code to avoid the problem. Always the best option when possible and/or practical!

2.2 Package Support Framework

Running existing software inside the MSIX container can lead to a number of situations where remediation action is required. While modifying your existing code to avoid these situations is generally advisable, this isn't always possible or desirable. The Package Support Framework (PSF) contains a collection of software components that you can use to overcome some of these issues.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Generally, we look to the PSF when source code modifications are not possible. The primary use of the PSF is for IT professionals who are repackaging vendor applications into MSIX, but there are uses for developers as well.

The PSF is an open-source project hosted by Microsoft on GitHub at https://github.com/microsoft/MSIX-PackageSupportFramework. It comes with a number of ready-to-go projects that can be used to overcome common issues. The PSF also has a NuGet package, although currently out of date. An improved version of the PSF is also available from Tim's fork, https://github.com/TimMangan/MSIX-PackageSupportFramework, which has release zip with binary redistributable options.

But the PSF is also a framework from which you can build your own application compatibility fixups, fixups that are written to your exact needs. All it takes is some C++ coding skills to write your own fixup.

Developers should generally look to the PSF only in the following scenarios:

- Your application requires start menu situations not allowed by MSIX. These can include command line arguments, multiple shortcuts to the same EXE and issues that occur because of a change in the process working directory.
- Your application requires a runtime script to be used, especially at first use.
- Your application contains too many common issues to attempt code-level remediations at this time.
- You want your code to be the same for MSI and MSIX installed packages.

In addition to the source code on GitHub, there is also a NuGet package with prebuilt versions. Unlike many NuGet packages, this doesn't add something that your code can call, but just provides the redistributable files for you to include and wire up.

The PSF does not currently have any real versioning. The NuGet packages are tagged, and the latest NuGet package should usually correspond to the master branch source. The master seems to be updated infrequently, and as long as you are willing to build your own copy, the develop branch might be more up to date. Microsoft partners that provide products to create MSIX installers may have their own way to embed their built copy into your setup project, but the Windows Application Project that is built into Visual Studio, for example, does not.

2.2.1 PSF theory of operation

The PSF has a launcher and a series of DLLs that are injected into processes that require the fixup. These fixup DLLs are the modern equivalent of AppCompat shims of the past. These DLLs are based on Detours, a standard DLL injection technique employed by other Microsoft products like App-V and UEV. These Detours-based injections typically target specific API calls made by your code; for example, to open a file or registry key. The code inspects the arguments of the call and possibly modifies the arguments before calling the intended function, or possibly taking completely different actions in order to satisfy the application request.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The PSF uses a configuration file named config.json that will be part of your package. This configuration file provides all the settings to tell the launcher, and the fixups, what to do. The format for the config.json file is described in Chapter 8: PSF Json Syntax.

You will be adding the appropriate components and a config.json file into your project. You will also be modifying any external entry points to EXE files that need the PSF, such as shortcuts and file associations, to point to the appropriate launcher.

As the PSF is written in non-managed C++ code, the appropriate architecture (x86 or x64) PSF components must be used to match the architecture that your application runs as. In the case of applications built as AnyCPU, it is recommended to include both in your package.

2.2.2 PSF features

The PSF consists of a number of components, and you will add a subset of these into your project:

Name	Type	Description
Config	json	Configuration file for the PSF.
PsfLauncher	EXE	This component would be used as a replacement for your application EXE anytime the PSF is needed. For example, you would modify your shortcut to point to this EXE, and the json configuration file will take the necessary steps to start your application with the required PSF components. It addresses common shortcut issues working directory issues and it allows for a pre-launch script (typically used for first-use activities involving a script file that may be customized by the customer in an enterprise deployment).
PsfRuntime	DLL	Required component that performs the injections of fixup DLLs into processes running inside the container. These fixups intercept Windows API calls to alter the calls made to the OS to fix issues.
TraceFixup	DLL	Utility that outputs trace information at API locations that are targets for fixups. Used to determine what fixups might be needed (or where code could be improved).
PsfMonitor	EXE	Used with TraceFixup to provide a ProcessMonitor-like GUI to view TraceFixup output.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

FileRedirectionFixup	DLL	Used for two purposes: • Allows your application to see files that are part of your package but located under the VFS\AppData or VFS\LocalAppData folders. • Allows your application the illusion of being able to write to files inside your package. This leverages copy- on-access functionality to make a copy in the user's LocalAppData profile area and redirects further file I/O for that file to this location.
RegLegacyFixups	DLL	Modifies call to registry keys and items to make the call conform to the restrictions of the container. For example, a call to open an HKCU registry key with FullAccess will fail in the container, and this fixup can modify the call to ask only for Read/Write access.
EnvVarFixup	DLL	Provides for an application container environment variable. May be needed if your current installer adds any new environment variables.
DynamicLibraryFixup	DLL	Solves "DLL not found" issues that can happen because of MSIX differences.
ElectronFixup	DLL	Useful for software written for Electron.
WaitForDebugger	DLL	Because you'll use a launcher to start the application, adding this is useful to add a break point when the PSF-enabled application starts. You then attach your debugger to that process.
Other	DLL	There are a couple of other test fixups; however, these are rarely, if ever, used anymore.

2.2.3 Common MSIX container issues: how to avoid, or use, the PSF to solve them

Ideally, if you can change your application source code to solve these issues, you should do so. Most of the time, these changes are also appropriate for your MSI deployment of the application and should be considered as part of the modernization process.

This section will outline some common issues and suggest possible actions.

2.2.3.1 Shortcut issues

Your existing lnk file and url file-based shortcuts will be migrated to the AppXManifest file as Visual Elements. The details of that process are described in 4.5.3 Visual elements.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- Existing shortcuts that point to non-EXE targets require a Launcher. This could be PsfLauncher or your own.
- Existing shortcuts that have command line arguments require a Launcher. This could be PsfLauncher or your own.
- Applications started by shortcuts that use the process Working Directory. Your code could be modified to determine the folder you need in a different way (registry value to if .Net Framework based, you can use ExecutionAssembly), or PsfLauncher can solve it.

2.2.3.2 Installer custom actions

Your first choice will be to determine whether the equivalent functionality exists in the AppXManifest file. If it does, you should use that.

You may also be able to add PsfLauncher as the end-user entry point and have it perform a first- run script action. Keep in mind that the script can't do many of the things the custom action performed without prompting the user for elevation via User Account Control (UAC) prompt.

Your script will need to be in the form of a PowerShell script; however, that script can call whatever it needs.

2.2.3.3 Packaged files under VFS\AppData and VFS\LocalAppData

If your package includes any files under these folders, your application will not see them. Whether this is an oversight of the VFS layering in MSIX or a purposeful feature is irrelevant if you need them.

It may not be possible or advisable to remove these files from your package. While your application could be altered (in code or by using a PsfLauncher script) to create the files on first run, in many cases we prefer these files to be part of the package so that an enterprise can customize the application by applying a modification package.

Adding PsfLauncher and the FileRedirectionFixup can also solve this by making a copy in a separate area of the user profile and redirecting File I/O there.

2.2.3.4 Packaged files requiring ability to be written to

It is common for the application package to contain files with initial data/settings that can or may be modified by the end-user in the use of the application. Application configuration settings are a great example.

Previously Microsoft encouraged vendors to stop using the registry for these settings and use file-based configuration. As each application has its own application hive you might reconsider going back to the registry for these settings.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Alternatives include keeping the initial file out of the package and writing to an external area, such as the user's profile, or using PsfLauncher and the FileRedirectionFixup to allow for copy-on access.

2.2.3.5 Improper registry access

Your traditional app always had the ability to ask for any and all permissions in the user registry hive, leading to a lot of sloppy coding that will not work under MSIX. There are also some reasonable things your code might have tried to do in the user hive that are just not allowed.

Under MSIX, you can provide for user registry settings to be part of your package. Those keys and settings, like the files in your package, are immutable. If your code tries to add to a key or change a value, it will be allowed to do so, as the MSIX runtime will redirect those changes to a special Registry.Dat file for the package in the user's profile.

But the MSIX runtime refuses registry calls asking for permissions that were allowed in the past. We see this most often with apps that open a registry key including the following permissions not allowed by the MSIX runtime currently:

- Delete Access
- Change Owner Access
- Change Permissions Access

Quite often the code only requires read/write access to the key but requests a full access that includes these capabilities. It might never intend to delete the key, but MSIX isn't going to risk it, and the existing call will fail.

The best approach is to modernize your code to request only the permissions that you need. Otherwise, the PsfLauncher and RegLegacyFixups may be used with a ModifyAccess rule that will change the parameters of your call to something acceptable.

Another form of registry access that can be remediated by RegLegacyFixups is when the application uses a HKCU-based key as a scratch pad creating a subkey at application launch, adding temporary items to it, and then deleting the key at the end. Even though the subkey was created in the redirected application hive in the user's profile, the call to actually delete the key will fail.

In this case you should consider another form of scratch pad; however, RegLegacyFixups does have a FakeDelete rule that will just return success to the application delete call without attempting to delete the key.

If your product uses externally developed components that you do not have source code access for, should those components violate the MSIX runtime rules for the registry you will need to either use the PSF RegLegacyFixup or replace those components.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

2.2.3.6 Environment variables

Your current installer might alter the Path Environment Variable or create new ones for use in your package. We'll treat these separately in section 4.12 Environment variables.

2.2.3.7 Path variable

There are two reasons your current installer might change the Path variable: to help the system find an EXE or to help your application find DLLs. Both of these uses should have been migrated to an AppPath registration previously (i.e. "HKLM\Software\Microsoft\Windows\CurrentVerion\App Paths"), but AppPath registrations need to be migrated to the AppXManifest file anyway.

The equivalent for finding the EXE is called "ExecutionAlias" in the AppXManifest. This entry in the manifest will cause anything in the shell that tries to start a given name to use the path and file listed in the entry. While normally you'd enter the name of the EXE, it is possible to use another name so that the end-user can just type "flowers" in the search box (next to the Windows logo in the Windows taskbar) and have it start your ILoveBlueFlowers.EXE program.

There is no equivalent in the AppXManifest for "find DLLs." You can restructure your program, or if .NetFramework-based you can add SetDLLDirectory() to your program initialization, but to use this technique you need to explicitly load the DLLs in your code using LoadLibrary after adding the directory to the search list used by LoadLibrary. The other alternative solution is to include PsfLauncher and DynamicLibraryFixup with a configuration mapping in the json file from the DLL name to the relative file path in your package.

2.2.3.8 Other variables

You should look at replacing these unless the end-user really needs to modify them. If the end-user needs to modify them, PsfLauncher and EnvVarFixup can be used to make your package setting of the environment variable visible and optionally modifiable. Modification would need to occur from inside the container, so you may need to add code to your app to change the setting.

2.2.4 Adding PSF components to your package

If you find that you need the PSF, the best practice would be to place all PSF files at theroot folder of your package. Some third-party installers that produce multiple package output formats may refer to this as the traditional "Application" folder.

2.3 Windows App SDK (aka Project Reunion)

While there are also many code-level changes that you might make to avoid issues due to running in the containerized MSIX environment while still building traditional Win32 and .Net Framework components, and you can use the PSF to solve others, in the long run you will likely eventually start using the newly announced Windows App SDK.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The Windows App SDK is a set of tools and libraries that provide access to a unified set of APIs and modern UX frameworks like WinUI3 and WebView2. Windows apps built using Windows App SDK can take advantage of the new features and functionalities from the UWP APIs along with the best of Win32/.Net Framework APIs combined with the modern user experience.

Originally announced by Microsoft at Build 2020 as an open-source project under the code name Project Reunion, the Windows App SDK is released on 16th Nov 2021. The goal of the project is to unify Win32 and UWP APIs to provide a single platform for the developers to build apps with modern user experience.

Before starting your project, you should look into the Windows App SDK APIs and determine whether this is also the right time to consider a move to the Windows App SDK. At the time of this writing, version 1.0 of the API had just released, so in this book we will focus on migrating traditional Win32 and .Net Framework apps without it. Currently, we don't recommend using this SDK for migrating existing app projects, but for investigation projects using newly written code.

Windows App SDK is made available as an extension in Visual Studio 2019 that includes project templates configured to use Windows App SDK components and as a NuGet package that can be installed in the existing package build. The extension is available from the following link:

https://marketplace.visualstudio.com/items?itemName=ProjectReunion.MicrosoftProjectReunion

Windows App SDK is decoupled from the operating system, making it possible to break the dependency on the OS version developing certain features in the app that may be dependent on a certain version of OS. Unlike the new Windows APIs and features that are released with a new version of OS, which happens twice a year, Windows app SDK being decoupled from OS will have its own release cycle. That said, apps built with the Windows App SDK are currently only compatible with runtime operating systems of Windows 11 and Windows 10 1809 or above, so consider if your customers may have older systems.

Microsoft says that updates to Windows app SDK will be released on a faster cadence to enable developers to take advantage of the new features much earlier than ever before. This enables fast-paced development, which is one of the goals of Project Reunion. Because of this decoupling, apps built using Windows app SDK can work across various versions of Windows operating system, starting with Windows 11 or Windows 10 1809. Keep in mind that your other calls to the traditional APIs, whether they be UWP or Win32 oriented, may still have OS dependencies and that this SDK is not a silver bullet for all problems.

Developers, while building Windows apps, must choose platforms and frameworks. Each platform and framework provide a certain set of features that may be unique to a specific version of the platform. An app built using such features will have a dependency on that specific version. Now, it makes it a lot easier for the developers to simply choose Windows App SDK as a unified platform for building their apps without having to worry about what version of OS is being used by their app users.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

In using the Windows App SDK you will set your target framework as .Net 5 (or above). This new framework is really the next version of .Net Core 3, so moving to this from .Net 3 or 4-based code may require some additional work, and there are gaps that might only be met by the older frameworks. However, in the new SDK the UI is separated from the APIs, so it is possible to keep your existing Windows forms or .Net 4 XAML UI if necessary. And with features such as XAML islands, there is even a limited ability to mix and match UI components.

When migrating existing applications, we expect that most developers would base the main executable on .Net 5 and continue to use most (if not all) of the DLL and COM components built using the older frameworks. In the end, if you use .Net 5, your package will be built as an MSIX package. Microsoft has made it clear that they intend .Net 5 to replace these other frameworks in the long run, but we expect that there will continue to be some functionality (like com) that will require the older frameworks to be around for a long time.

The SDK also comes with release channels, allowing you to select the experimental or preview channel early in your development cycle to access new features that should be in the stable channel by release time. Currently, Microsoft is frequently updating the SDK, and each new release is filling in some of the gaps as well as adding new UI capabilities.

Reference: https://docs.microsoft.com/en-us/windows/apps/windows-app-sdk/#benefits-of- the-windows-app-sdk-for-windows-developers

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 3: Overview of an MSIX Project

This chapter will provide an introductory overview of the process used to build the package file(s) that you would deliver to the customer. In many cases, we will provide links for details that may be found either in later chapters of this book or in external web resources.

Your project output for an MSIX style of package will be a bit different from what you might be used to. The types of files that your project might generate for delivery will include a subset of these:

File Type	Description
.appx or .msix	Contains a version of a package. The file type extension traditionally indicates if the package is UWP or MSIX; however, the extension is ignored and the contents of the internal AppXManifest file determines what type the package actually is.
.appxbundle or .msixbundle	Optional. Contains a bundle manifest file that references several packages that are also placed inside the bundle. This typically is used to bundle packages built for different processor architectures, such as x86, x64, and/or ARM.
AppInstaller (.XML)	Optional. This file may be used for installation to reference dependency packages and/or provide automatic update information. The package or package bundle is referenced in this XML using a URI path, and the package would be external to this file.
Shared Package Container (.XML)	Optional. This file is more likely to be generated by customers, but a vendor might create it if the package needs to integrate with another and the other package doesn't have a direct plug-in model.

3.1 Digital signing

MSIX packages provide secure installation by installing the application to the app container and enforce the app packages to be digitally signed with a valid code signing certificate to install successfully, hence ensuring only the trusted apps are installed and enforcing the integrity of the package and its container after it is deployed.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Before installing a signed MSIX application, it will require that a code signing certificate used to sign the package be trusted by the end-user system. For certificates issued by a public certificate authority, this should be automatic, but for self-signed certificates it will be necessary to import the certificate into the system store on each device to trust the MSIX packages signed by that certificate.

During installation of the package, and also when the application is used, Windows checks the integrity of the package and its contents after the deployment by parsing the information updated in the AppxBlockMap.XML and AppxSignature.p7x files created during the package creation and digital signing. While the architecture of MSIX should prevent the package from being tampered with after installation, the package integrity can be tampered by modifying the files in the package either by the third-party application or manually by the user's intervention. If the package is found tampered, Windows will block Application launches from this package. If the package is installed from the Microsoft store, it will then initiate a repair of the package automatically from the original store source. If the package was installed from a custom source location, the installed package must be manually repaired to fix the package integrity. In this case, we expect that uninstalling and reinstalling the application package is the easiest choice to resolve the issue.

3.1.1 Signing an MSIX package

It has long been a best practice to sign at least the application EXE DLL files that are part of your package. This approach should continue even when part of an MSIX package, as protection schemes like AppLocker and Windows Defender Application Guard depend on having signatures to match against. Similarly, it has also been a best practice to sign the MSI or EXE installers as part of the installer creation process.

When replacing the existing installer with an MSIX package, you will be required to digitally sign this file before it can be deployed to any machine other than one in Development Mode. But signing the MSIX package is a little different and requires a small amount of additional consideration. During the MSIX package sign, the signing tool reads the publisher information from the package AppXManifest file and compares it with the certificate subject information. If the values are not matching, then the MSIX packaging signing throws an error. So this process ensures that the AppXManifest file publisher name will match the signing certificate Subject.

The Subject of the signing certificate is used to identify the Publisher who is doing the signing. The Publisher string is updated in the form of "CN=Flexera Software, O=Flexera Software, L=Itasca, S=Illinois, C=US" where CN is the common name, O is the organization name, L is locality, S is state and C is country.

MSIX package can be signed using the Microsoft SignTool utility by specifying the package name and the certificate details as command line parameters. Packaging applications provide a graphical interface to

```
signtool.exe sign /f C:\Test\Certifcate.pfx /p password /fd sha256 /tr http://timestamp.digicert.com
"C:\TestPackage\TestMsixSigning.msix"
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

select the certificate and to update the publisher information with the certificate subject during the build.

So it is required to sign all the MSIX packages before install. Windows throws an error if the package is not digitally signed while installing.

Additionally, the SignTool utility will generate a file named “AppxSignature.p7x” and add this file to the package during the signing process. The block map file, AppxBlockMap.XML, is then verified and secured with a digital signature and the overall package file is signed.

3.1.2 How to sign an MSIX package

Signing is one of the critical tasks to perform, especially with MSIX packages because the user will not be able to install the MSIX package if the MSIX package is not signed with a trusted certificate.

To get started, you will need a code-signing certificate. In creating the certificate, you will need to supply certain information, such as:

- EKU, marking the certificate as a code-signing certificate.
- Subject field.
- Certificate expiration date.
- Certificate algorithms, such as SHA256.

We should note that public certificate authorities have moved from using 1024 bit encryption to a minimum of 2048 bits as the standard for code signing. As this encryption is used only to sign the package and to extract out the files during installation of the package, you may want to future-proof your packages and get or create a certificate with 2048 or 4096 bit encryption.

Depending on your intended use of the application package build environment, and methods of delivery used for your users, there are several options available to consider. The four most common sources of certificates for code signing are described here.

3.1.2.1 Self-signed certificate

Self-signed certificates are appropriate only when the organization signing the package is to be the user of the application package, such as internal line of business applications or enterprise repackaging of a vendor installer. Self-signed certificates are also sometimes used to sign the MSIX application during the development and testing of the application, and the package is re-signed using a different certificate prior to publishing.

Self-signed certificates are created in-house by one of three methods. The main benefits of self-signed certificates are that the cost is zero, and you can get started right away.

A self-signed certificate can be generated by:

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- Executing the "New-SelfSignedCertificate" PowerShell command.
- Executing the "makecert" command from the Windows SDK.
- Using an in-house PKI system such as Active Directory Certificate Services Role for your AD Systems.

To install MSIX packages signed with a self-signed certificate, the certificate must be installed into the certificate store of that computer and added to the Trusted Roots folder of the system store. In an enterprise, this can often be performed as part of image building, Group Policy deployment, or in-house systems such as System Center Configuration Manager or Microsoft Endpoint Manager.

3.1.2.2 Trusted certificate

Trusted certificates are acquired from publicly registered certificate authorities (CA), hence the certificate will be accepted by default on the end-user systems for installing MSIX packages.

Microsoft updates the list of publicly registered certificate authorities within the OS. So signing with the trusted certificate from the publicly registered certificate authority guarantees that the Windows OS will see a valid digital signature applied to your package. This applies to both enterprise systems and to consumer systems.

A public CA should also provide a time stamping server and therefore provide the time stamping service. Usually, the certificates provided by the Certificate Authorities (CA) will have a short life span ranging from one to three years. So the certificates may expire but the time stamping helps to extend the life of the package by validating that the package was signed prior to certificate expiration and therefore should still be trusted.

3.1.2.3 Device guard signing

Device Guard signing is a Device Guard feature that is available in Microsoft Store for Business and Education. Device Guard is a feature set that consists of both hardware and software system integrity hardening features.

Device Guard signing requires an Azure Active Directory (AD) or Office 365 account for your organization and needs to be the global administrator for your organization.

To sign up using Device Guard Signing, you would do the following steps:

1. Sign into the Microsoft Store for Business or Microsoft Store for Education portal. This is a free service for customers with an Azure Tenant, including Office 365 customers. Your Azure Tenant Administrator can set up this service.
2. Assign the role with permissions necessary to perform Device Guard signing. You need the Device Guard signer role. This is the least privileged role that has the ability to sign. Other roles include Global Administrator and Billing Account owner.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

3. Register your application in the Azure portal with the proper settings so that you can use Azure AD authentication with the Microsoft Store for Business.
4. Get an Azure AD access token in JSON format.
5. Obtain the Device Guard Signing version 2 (Microsoft.Acs.Dlib.DLL) by downloading the NuGet Package.
6. Run SignTool to sign your MSIX package with Device Guard signing and pass the Azure AD access token obtained in the previous step.

```
signtool sign /fd sha256 /dlib Microsoft.Acs.Dlib.dll /dmdf <Azure AD in .json format> /t
<timestamp-service-url> <.msix package>
```

3.1.2.4 MSIX signing with Azure key vault in a CI/CD pipeline

To sign the MSIX package as part of the CI/CD pipeline with the certificate stored in the Azure Key Vault, we are going to use a tool called Azure SignTool, which is an open-source tool. It works like the standard SignTool utility included in the Windows 10 SDK, but instead of using a local certificate, it connects to Azure Key Vault to use one of the available certificates. To establish the connection, we first need to register an application on Azure, which will give us the credentials we need to enable Azure SignTool to authenticate against our Azure Key Vault service to get the certificate and sign. Azure Key vault needs to be configured with a valid signing certificate, and necessary access permission for the application registered in Azure.

Azure SignTool is available as a .Net global tool. So the tool can be installed by executing the command below in the Azure pipeline. Also be sure to have the latest .Net SDK installed on the host to enable the dotnet command.

```
>dotnet tool install –global AzureSignTool
```

More information on configuring the Azure Key Vault and Azure can be found here: https://docs.microsoft.com/en-us/windows/msix/desktop/cicd-keyvault.

3.1.3 Time stamping

Usually, the certificates provided by the Certification Authorities (CA) will have a short life span ranging from one to three years. Without timestamping, Windows throws the error while installing the signed package after the certificate has expired, considering the certificate to be invalid. Adding Timestamping helps the package continue to be valid and accepted by the Windows OS even after the certificate has

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

expired. Timestamping allows the package signature to be validated with respect to the time it was signed instead of the current time.

Windows SignTool and the packaging applications support adding the time stamping to the signature during the digital signing by specifying the timestamp server URL. So the signing tool communicates with the timestamp server (during the signing process) to update the hash and store the information in the file signature during the digital signing. After applying the timestamp, the package will be valid and can be installed even after the certificate has expired.

3.1.4 Debug signing errors

A few of the more common issues with digital signing are discussed here:

3.1.4.1 Publisher name mismatch with the certificate subject

MSIX package signing throws an error ("Error: SignerSign() Failed" -2147024885/0x8007000b) if the publisher entry in the AppXManifest file is not matched with the signing certificate subject.

To fix this error, be sure to update the publisher entry in the AppXManifest file with the signing certificate subject. To get the certificate subject, either double-click the certificate in File Explorer, and then select the details tab, or execute the "certutil -dump" command to dump the certificate information to the command line console window. The match must be character to character, so spaces are important!

3.1.4.2 Bad PE certificate (0x800700C1)

Bad PE certificate error is thrown while signing the MSIX package if the package contains executable with corrupted certificates.

To identify the executables with corrupted certificates, set an environment variable named APPXSIP_LOG with value "1" in a command prompt, and then sign the MSIX package by executing the SignTool.EXE with the certificate and the MSIX package details, which dumps the file names with the bad PE certificate in the console window.

3.1.5 Maintain the package identity across the digital certificate

The package identity element is a top-level element (under Package) that contains a set of attributes that uniquely define this package. One of the attribute "Publisher" hold the Subject name of the signing certificate. Package identity is used to check whether the package is already installed on the machine, or to identify the main package to install the package update. See section 7.3 Identity Element for more information on Identity Element.

So, the package identity will vary if the package is signed with different digital certificate. And the packages will be considered entirely new package with its own package identity. This kind of situation arises when the digital certificate got expired, and renewed to get a new certificate. One more case is when there is an organization change, like the organization is acquired by some other organization or

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

the organization itself started a new organization. All these cases, it is required to persist the package identity across the package to maintain the App's update experience. Else, the package will be installed as an entirely new package.

To persist the package identity, need to follow the steps mentioned in the Microsoft documentation which create an artifact to show the relationship between the old certificate and the new certificate: https://docs.microsoft.com/en-us/windows/msix/package/persistent-identity

In case of certificate expiry, you need to do the steps and create the artifact before the certificate expiry. This is relatively a new feature, supported from windows 11 onwards while writing this book.

3.2 Package components

The files that end up in your MSIX package will be gathered as a set of “ingredients” that you supply along with some additional files that are created and added as part of the build process.

While you may need to generate some files directly, such as image files for the start menu, most of the files will likely be output by other vendor tooling used in prior steps of the build process or as part of the MSIX package build itself.

3.2.1 Ingredient files

The ingredient files will be a set of files included in a folder that will ultimately be passed to the MakeAppX command to generate the package.

3.2.1.1 Files at the root

Although the folder containing your ingredients may have any name, for convenience and clarity we will refer to this as the "Root folder." Additionally, files placed in this ingredient folder, either directly or under any subfolders other than one named ‘VFS,’ we refer to as files at the root.

When your package is installed, this will be the folder containing your package ingredient files, the equivalent of the installation folder from traditional setup programs. The person installing the package usually no longer has a choice of installation location, which will be C:\ProgramFiles\Windows Apps\[Your Package Family Name]. But be aware that there are situations where the package might be installed into an alternative location, so any package file references made by your code will need to be relative to the actual installed location.

- For managed code, we’d recommend using the application executing assembly folder.
- For unmanaged code, techniques used in the past, such as Current Working Directory or having the installer create a registry entry with the folder, may be problematic, and it may be easier to use VFS pathing so that your code can just reference a native path that is overlaid with your package VFS folders. As an alternative, you can create a registry string in your package using the [{AppVPackageRoot}] variable to reference the installed location.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

3.2.1.2 VFS files

Under the root folder you may have a folder called VFS, which is short for Virtual File System. The VFS folder can consist of a number of specially named subfolders, each of which acts as an overlay to well-known system locations. These same standard locations are also used in registry string references, in a slightly different form.

You will need to use VFS folders in your package if you intend to place files in locations that look like standard Windows folders. These files will actually remain inside your package under this VFS structure, but if your software references the old native path, it will find the folder in your package.

For example, you may be used to adding some 64-bit DLL files into the C:\Windows\System32 folder. By placing them in the VFS\SystemX64 folder of your ingredient list, any software running in your package can enumerate the standard System32 folder and see your files overlaid on top of those present in the system.

The best current reference for these folders is an old article from the Desktop Bridge: https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the- scenes.

The VFS folders we are aware of are shown in the following table, along with the standard system KNOWNFOLDERID reference:

VFS Folder	Registry Reference	Equivalent
System	[{System}]	FOLDERID_System
SystemX86	[{SystemX86}]	FOLDERID_SystemX86
SystemX64	[{SystemX64}]	FOLDERID_SystemX64
ProgramFilesX86	[{ProgramFilesX86}]	FOLDERID_ProgramFilesX86
ProgramFilesX64	[{ProgramFilesX64}]	FOLDERID_ProgramFilesX64
ProgramFilesCommonX86	[{ProgramFilesCommonX86}]	FOLDERID_ProgramFilesCommonX86
ProgramFilesCommonX64	[{ProgramFilesCommonX64}]	FOLDERID_ProgramFilesCommonX64
Windows	[{Windows}]	FOLDERID_Windows
Common AppData	[{Common AppData}]	FOLDERID_ProgramData
AppVSystem32CatRoot	[{AppVSystem32CatRoot}]	FOLDERID_System\catroot
AppVSystem32CatRoot2	[{AppVSystem32CatRoot2}]	FOLDERID_System\catroot2
AppVSystem32DriversEtc	[{AppVSystem32DriversEtc}]	FOLDERID_System\drivers\etc

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

AppVSystem32DriversStore	[{AppVSystem32DriversStore}]	FOLDERID_System\driverstore
AppVSystem32LogFiles	[{AppVSystem32LogFiles}]	FOLDERID_System\logfiles
AppVSystem32Spool	[{AppVSystem32Spool}]	FOLDERID_System\spool
Profile	[{Profile}]	FOLDERID_Profile
AppData	[{AppData}]	FOLDERID_RoamingAppData
LocalAppData	[{LocalAppData}]	FOLDERID_LocalAppData
Desktop	[{Desktop}]	FOLDERID_Desktop
Documents	[{Documents}]	FOLDERID_Documents
Fonts	[{Fonts}]	FOLDERID_Fonts
Pictures	[{Pictures}]	FOLDERID_Pictures
Videos	[{Videos}]	FOLDERID_Videos
Programs	[{Programs}]	FOLDERID_Programs
(not applicable)	[{AppVPackageRoot}]	The root folder of the package
AppVPackageDrive	[{AppVPackageDrive}]	The drive letter for Windows

When multiple ways to reference a file path exist, you should use the most specific choice available. For example, you would reference the VFS\Desktop instead of VFS\Profile\Desktop. The AppVPackageDrive identifier covers anything not covered by other entries. The AppVPackageDrive reference is typically used when there is a need to have a file path look like C:\VendorName. Note that the “AppV” names were borrowed from Microsoft App-V but is not an indication that App-V is involved.

Traditionally, installers place most of their files in the root area. With MSIX Packaging there is a choice of placing them at the root or under the appropriate VFS\ProgramFilesX** folder.

- The root area of the files is private to this package and not subject to overlay behaviors.
- Files placed in VFS locations will appear to processes running in the container as being overlaid on top of the equivalent system folder. Thus, if the app enumerates files under C:\Windows, it will see the VFS\Windows folder overlaid upon the C:\Windows folder.
- Additionally, only files in the package that are in VFS are subject to an overlay of a modification package. A modification package is a separate package that may consist of a plug-in, or a customized configuration file. Files in that package must be VFS referenced, and these will overlay the primary package and local system.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- While (at the time of this writing) we don't know the details of how a future MSIX feature for grouping packages will work, given the equivalent feature in App-V we would anticipate that the VFS folders of all packages will merge while root folders remain separate. This grouping function would allow multiple independently generated packages to all run together in a single container.

There are a few odd VFS restrictions to be aware of:

- Files in the package under the "AppData" and "LocalAppData" VFS folders are not visible to the application unless the PSF FileRedirectionFixup is used. See 2.2.3.3 Packaged files under VFS\AppData and VFS\LocalAppData about use of these folders.

- Files in the package under the "CommonAppData" VFS folder will not be visible either, unless the package is running on Windows 10 2004 (20H1) or above, unless the PSF FileRedirectionFixup is used.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

3.2.2 Files typically generated as part of the build process

There are some package files that you do not include in the ingredients, as they will be generated for you by the MakePri and MakeAppX steps:

Some setup projects will generate the AppXManifest.XML file during the build process as well. In any case, this file must be included in the ingredients.

MakePri is used during the build process to generate the necessary language-specific packages and will appear in the package as a file called resources.pri.

MakeAppX will create a few files:

- The [Content_Types].XML file. This file has an entry containing MIME information for every filetype included in the package.
- AppxSignature.p7x and AppXMetadata\CodeIntegrety.cat files, which are used for security.
- AppxBlockMap.XML, which is used for single instance download/storage.

3.3 MSIX build process

There are several tools from Microsoft and third-party vendors (like Flexera and Revenera) available in the market to either create new packages or convert traditional MSI and EXE-based installers into MSIX packages. Most of the tools do this work automatically.

There are occasions where custom modifications are needed for a package; for example, one might want to use Package Support Framework (PSF) to handle a few unexpected scenarios, or need some custom edits to AppXManifest file, or add localization/scaling support to the existing package. In these kinds of scenarios, we may need to extract the MSIX package contents, add PSF files, make modifications to AppXManifest file, and repackage the extracted contents back into the MSIX package. The last step includes signing the package.

For an overview of the complete process in building a MSIX/APPX package manually without using any packaging tools, please refer to the documentation at this link:

https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-manual- conversion.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

3.3.1 Command line tools for MSIX from the SDK

The following command line tools are helpful in building your package and are available as part of Microsoft Windows 10 SDK (C:\Program Files (x86)\Windows Kits\10\bin\<build number>\x86 or x64 folder):

- MakePRI.EXE
- MakeAppx.EXE
- SignTool.EXE

3.3.2 Creating resources.pri using MakePRI.EXE tool

Your package may contain resource files that may be referenced using an index. The Resources element in the AppXManifest.XML declares the union of all languages and display scales for the resources that the package contains.

A primary use of the resources.pri would be to reference graphic files used; for example, in the start menu, based on the screen size, dpi settings, and text scaling in use on the target system. While your traditional installer may have chosen which files to install at installation time, under MSIX they are all included in the package and installed, along with a Primary Resource Index (PRI) file, which is then used at per-user installation to select the current best resource to configure. MSIX also uses this to index resources such as language-specific files.

The MakePRI.EXE tool can be used to create or dump PRI files. This utility comes with Windows 10 SDK and is usually located under (C:\Program Files (x86)\Windows Kits\10\bin\x86) folder or in the respective dedicated SDK folders.

MakePri typically uses an XML configuration file along with the AppXManifest file Resources element and the Assets folder of your ingredients to control the creation of the pri file.

The XML configuration file syntax is documented here https://docs.microsoft.com/en-us/windows/uwp/app-resources/makepri-EXE-configuration but is usually generated for you by vendor tooling, and you should not be concerned with its contents, but it might be possible to build your own customized version. Below is an example file generated by the Microsoft MSIX Packaging tool:

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
<?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<resources targetOsVersion="10.0.0" majorVersion="1">
     <index root="\" startIndexAt="layout.resfiles">
          <default>
               <qualifier name="Language" value="en-us" />
               <qualifier name="Contrast" value="standard" />
               <qualifier name="Scale" value="400" />
               <qualifier name="HomeRegion" value="001" />
               <qualifier name="TargetSize" value="256" />
               <qualifier name="LayoutDirection" value="LTR" />
               <qualifier name="Theme" value="dark" />
               <qualifier name="AlternateForm" value="" />
               <qualifier name="DXFeatureLevel" value="DX9" />
               <qualifier name="Configuration" value="" />
               <qualifier name="DeviceFamily" value="Universal" />
               <qualifier name="Custom" value="" />
          </default>
          <indexer-config type="folder" foldernameAsQualifier="true"
          filenameAsQualifier="true" qualifierDelimiter="." />
          <indexer-config type="resw" convertDotsToSlashes="true"
          initialPath="" />
          <indexer-config type="resjson" initialPath="" />
          <indexer-config type="PRI" />
          <indexer-config type="RESFILES" qualifierDelimiter="." />
     </index>
</resources>
```

MakePri compiles into a file called resources.pri that is packaged with the app, and then it is called upon by the Windows Runtime to resolve resources when the app runs in a user context. If any of the visual assets of application ingredients are modified after the PRI is created, a new PRI file needs to be generated.

3.3.3 View resources.pri contents using MakePRI.EXE

PRI file is a binary file, and its content can be dumped using either the MakePRI.EXE tool (as below) or PRI APIs.

```
C:\Program Files (x86)\Windows Kits\10\bin\10/0/19631.0\x86>MakePri.exe dump /if
“C:\MsixTest\resources.pri” /of “C:\MsixTest\resources.pri.XML”
Output File at C:\MSIXTest\resources.pri.XMLSuccessfully
Completed
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

For more information on MakePRI.EXE usage and examples, please refer to Microsoft documentation: https://docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-manual-conversion.

3.3.4 Building MSIX package using MakeAppx tool

MakeAppx.EXE creates, optionally encrypts and decrypts, and extracts files from app packages (.msix or .appx) and bundles (.msixbundle or .appxbundle). This tool is included in the Windows 10 SDK and can be used from a command prompt or a script file.

The tool can be used to create a new package or to extract the contents of an existing MSIX package, allow edits, and repackage the contents back into the MSIX package. To create an MSIX app package using the 'pack' option, this tool needs either a content directory or a mapping file for the location of the package, and the AppXManifest file also should be present as part of the content directory root folder. You can also encrypt a package while creating it, although this is not commonly done.

For more information on MakeAppx tool usage and examples, please refer to this document: https://docs.microsoft.com/en-us/windows/msix/package/create-app-package-with-makeappx-tool.

3.3.4.1 Signing the package using SignTool

SignTool.EXE is a command line tool used to digitally sign an app package or bundle with a certificate. Except for developers, MSIX packages cannot be installed without being signed using a code-signing certificate. Additionally, the subject field of the code signing certificate (a string in the form of "CN=xxx, O=yyy, …") must be used in the Publisher element of the AppXManifest.XML file inside the package.

The certificate can either be created by the user (for testing purposes) or be issued by a company (for distribution). Except when distributing through the Microsoft Store, those who plan to distribute the MSIX file outside their own company will likely want to purchase a code-signing certificate from a Certificate Authority.

```
signtool sign /f <TestCertificateCA.pfx" /t http://timestamp.digicert.com /fd SHA256
MyPackage.msix
```

The use of time stamping during the signing is highly recommended. For more information and examples using SignTool, including time stamping, please refer to Microsoft documentation https://docs.microsoft.com/en-us/windows/msix/package/sign-app-package-using-signtool.

SignTool has a debug option available to show certificate errors and filtering. It also logs more information in the event log for a few errors.

Detailed steps to troubleshoot the signing-related errors can be found at this link: https://docs.microsoft.com/en-us/windows/msix/package/signing-known-issues.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

3.4 MSIX bundles

The MSIX bundle package is a bundle of MSIX packages targeting different system architectures, language-specific assets, varying image-scale assets, or resources that apply to specific devices. MSIX Bundle helps you to pack the MSIX packages for different architectures and pack the related asset packages to one entity. During the .msixbundle installation, it installs only the packages that are applicable to a device's architecture.

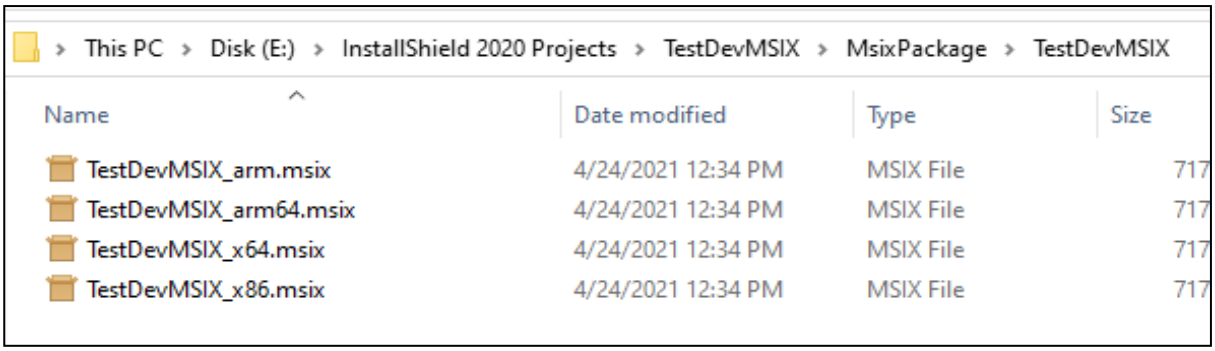
3.4.1 Building bundle packages

Bundle packages can be built using the MakeAppx.EXE tool available in the Windows 10 SDK. In addition, applications like InstallShield and others can provide graphical user interface to build bundle packages by specifying the payload MSIX packages targeting different system architectures.

```
C:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86>MakeAppx bundle /d
"E:\InstallShield 2020Projects\TestDevMSIX\MsixPackage\TestDevMSIX" /p
E:\MSIXBundle\TestDevMSIXBundle.msixbundle
```

To create the bundle package, you need to call the MakeAppx.EXE tool with the “bundle” option, “/d” specify the directory where the bundle payload packages are available, and the “/p” specify the output MSIX bundle package name.

The input directory in the command above contains the MSIX packages targeting different platforms. Here is an example:



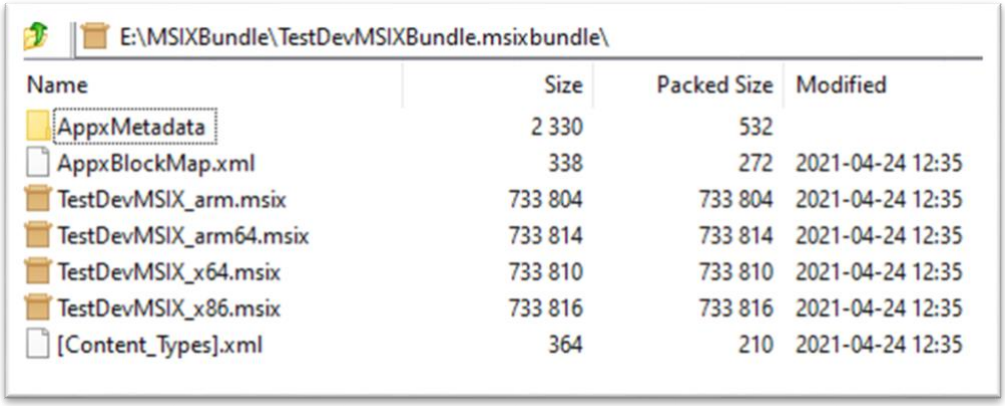
The generated MSIX bundle package contains all the payload packages as well as the footprint files required for the bundle package:

- AppxBlockMap.XML
- [Content_Types].XML
- AppxMetadata\AppxBundleManifest.XML
- AppxSignature.p7x (available only if the bundle package is signed)

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Here is an example:



While building the MSIX bundle package, ensure the payload MSIX packages have same package identity 'Name,' 'Publisher' and 'Version' or else the build will fail. The bundle package is not supported to pack MSIX packages with different package identity information.

Also, MSIX bundle is not supported to contain multiple packages targeting the same system architecture, nor is the "Neutral" architecture package with packages targeting architecture other than "Neutral" architecture.

3.4.2 Signing the bundle package

Like the MSIX package, the MSIX bundle package is also required to be signed before distribution or installation. But it is not necessary to sign the payload MSIX packages inside the MSIX bundle. The MSIX bundle package can be signed using the Windows 10 SDK SignTool.EXE.

The MSIX bundle package can be signed with a trusted certificate, which is acquired from a publicly registered certificate authority (CA). So the certificate will be accepted by default on the end-user systems for installing the MSIX bundle package.

```
signtool.exe sign /f C:\Test\Certifcate.pfx /p password /fd sha256 /tr
http://timestamp.digicert.com E:\MSIXBundle\TestDevMSIXBundle.msixbundle
```

SignTool.EXE generates a file named "AppxSignature.p7x" and adds it to the package during the signing process.

3.4.3 Flat bundle package

Flat Bundle packages are bundle packages, but the payload packages will not be inside the bundle package; instead, the payload packages will be referred from an outside location. Usually, the Flat Bundle will reference the payload packages from the same folder; otherwise, the path needs to be mentioned in the Bundle manifest.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Flat Bundle packages also can be built using the MakeAppx.EXE tool bundle option, with an additional "/fb" switch.

```
C:\Program Files (x86)\Windows
Kits\10\bin\10.0.18362.0\x86>MakeAppx bundle /d "E:\InstallShield 2020
Projects\TestDevMSIX\MsixPackage\TestDevMSIX" /fb /p
E:\MSIXBundle\TestDevMSIXBundle.msixbundle
```

3.5 Modification packages

MSIX also supports the creation of packages intended to modify or extend an existing package, which are referred to as Modification packages. The primary uses of modification packages include:

- To allow the customer to create customized configurations for your package.
- To enclose a plug-in to your package.
- Modification packages are just like regular packages, except:
- They include a reference to the primary package they modify. See MainPackageDependency in 7.6.4.3 MainPackageDependency sub-element.
- They may not include any applications or other forms of entry points. These must all be contained in the primary package.

Note that while it is technically possible to use a modification package as a patch file, we would recommend issuing a new version of the package instead.

The modification package will act as an overlay for the files and registry of the package. While the registry overlay is complete, at this time, only original package files located under VFS paths may be overlaid. If you want to be able to use modification packages for plug-ins or to allow your customers to provide their own file-based configuration, your original package structure must allow this.

As MSIX intentionally lacks the Custom Actions available in MSI installation, your enterprise customers who want to pre-configure your software will be looking to create their own modification packages. They would appreciate your making available documentation and possibly a template modification package that they could quickly edit.

3.6 Shared package containers

As valuable as the Modification package is, there are still times when we want to have two separate packages, each including its own start menu and other entry points, work closely together.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Normally, if both are MSIX packages, they would run independently in separate and isolated containers, unable to share components and/or configuration information. Application data associated with the packages might also be separate.

The Shared Package Container is an additional XML file that may be installed on an end-user's computer to tell the operating system that these packages should share a single container rather than be isolated from each other.

The Shared Package Container is supported only on operating systems with build 21354 and above (currently only Windows 11 supported).

Microsoft documentation on the Shared Package Container is found at the following links:

- About: https://docs.microsoft.com/en-us/windows/msix/manage/shared-package-container
- Schema: None at this time.

An example of the shared package container definition file is given in the about link above; the format is very simple. It is an XML file, and Microsoft has not introduced a separate file type for it.

The file simply contains an ordered list of packages that will be included in the shared container if present when the end-user starts any application contained in any of the packages referenced. To keep it simple, the references are by Package Family Name alone and not by version or architecture.

These packages would be considered to layer file and registry information together, such that if there is a file or registry conflict, the first package listed would win.

We don't anticipate the need for a vendor to distribute these files, but there may be situations where documenting for your customers how to use such a file may be considered. In most cases, the IT staff of an enterprise would be creating Shared Package Container files from bringing together applications from different vendors. If your application needs to work with an application from another vendor, you might want to provide a sample file for your customers.

The Shared Package Container file is installed using special purpose PowerShell commands included as part of the OS in builds supporting container sharing, such as the *Add-AppSharedPackageContainer* cmdlet.

3.7 AppInstaller (XML) file

While MSIX and Bundle files are generally directly installed into the operating system, the OS includes the ability to install and manage updates automatically, without the need for code within your programs to deal with all of that. Applications distributed through the Microsoft Store have used this capability for years, but now Microsoft has made it available for packages delivered externally by using an AppInstaller XML file to perform the installation.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Vendors who distribute the software from their own websites will want to offer both MSIX (or MSIX bundle) files for direct installation AND an AppInstaller file option. Individual consumers and small businesses will appreciate the option to install once and have their software always be up to date. But meanwhile, your larger enterprise customers will want to use their own tooling, especially Microsoft Endpoint Manager (aka Configuration Manager) to control the update process by using the .msix/bundle file directly.

The AppInstaller file is XML based and will tell the OS:

- Where to find the current msix file.
- Where to look for an updated AppInstaller file in the future.
- The configuration of the update settings policy for this package.
- The Microsoft documentation for the AppInstaller file is given in these links:
- About: https://docs.microsoft.com/en-us/windows/msix/app-installer/app-installer-file-overview
- Schema: https://docs.microsoft.com/en-us/uwp/schemas/appinstallerschema/element-appinstaller

To perform an installation with an AppInstaller file, the same PowerShell command is used (Add-AppXPackage), except that rather than provide the path to the msix/bundle file, an additional parameter— AppInstaller is used, followed by the URI path to the AppInstaller XML file.

When an installation is performed using the AppInstaller file, the OS will follow the update policy inside that file to manage the updates. When an update check is triggered, the OS will check the URI reference contained in the installed AppInstaller file to see if it has been updated. Thus, when you create a new version of your product, you can push it to any location on your website and update the AppInstaller file on your website for the new version (replacing the existing AppInstaller file).

The triggers that cause an update check that are available for update settings include a combination of the following:

- Check whenever the user launches the application.
- If the app remains running, a check can be made every X hours.
- The OS can check every eight hours, even if not run by the user.

When triggered, if an update is found, it will be downloaded and installed. If the app is running, by default the new version will become active on next launch, but there is an option to force the new version to be used immediately.

Business customers who don't have the luxury of an enterprise-scale distribution system like MEM can also copy and modify your AppInstaller file and place it on their own internal server along with a copy of the current MSIX file. Using your file as a baseline template, they would then change the URI references

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

to the AppInstaller file and packages to references their own server shares. Then, when they are ready to take your new version, they just copy the new MSIX package version from your website and update their copy of the AppInstaller file.

Below is an example of a simple AppInstaller file. Note that the file extension used is not defined by Microsoft, and we see both ".XML" and ".appinstaller" being used.

```
<?XML version="1.0" encoding="utf-8"?>
<AppInstaller Version="1.0.0.1"
Uri="https://vendorsite/folder/Package.appinstaller">
     <MainPackage Name="PackageNameField"  Publisher="CN=Vendor O=Vendor …"
              ProcessorArchitecture="x64" Version="19.0.0.0"
             Uri=" https://vendorsite/ PackageNameField-1.0.0.1_x64
             xwfzvwzp69w2e.msix" />
     <UpdateSettings>
           <OnLaunch />
           <AutomaticBackgroundTask />
           <ForceUpdateFromAnyVersion>true</ForceUpdateFromAnyVersion>
     </UpdateSettings>
</AppInstaller>
```

The file supports referencing either a MainPackage or MainBundle. Dependencies and other referenced packages may be included in this file, but the OS will not automatically install those additional items (a lack of a dependency will prevent installation of the main package/bundle). The ForceUpdateFromAnyVersion is recommended, as it would allow you to downgrade should a critical issue be detected after the release. If it were originally specified in this file, then you could restore the previous AppInstaller file to your website to revert your customers.

There is a second version of the schema for the AppInstaller file which also supports update policies to hold off launching the application and prompting the user for updating permissions. For more information about the AppInstaller file, see 4.16 Package Updates and AppInstaller File.

Microsoft does not require the AppInstaller file itself to be signed at this time.

3.8 MSIX core

MSIX Core (not to be confused with .Net Core) was designed to enable the installation of MSIX apps on previous versions of Windows, provided that the apps are already built to work on those versions of Windows. MSIX Core is built for the older Windows versions that don't currently natively support MSIX.

NOTE: Before investing resources in MSIX Core, you should conduct a simple test to see how well it works out of the box. The limitations discussed below are likely to be impactful to anything except for

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

the simplest of applications and supporting down-rev OS customers with your current installer might be a better strategy.

To enable MSIX Core support for your app package, the MSIX package AppXManifest must contain a proper TargetDeviceFamily entry with the name “MSIXCore.Desktop” and a “MinVersion” matching the old operating system build number. To enable native MSIX support as well for the app package, you need to include the relevant Windows 10, version 1709 and later entry as shown below:

```
<Dependencies>
  <TargetDeviceFamily Name="MSIXCore.Desktop"
                      MinVersion="6.1.7601.0"
                      MaxVersionTested="10.0.10240.0"/>
  <TargetDeviceFamily Name="Windows.Desktop"
                      MinVersion="10.0.16299.0"
                      MaxVersionTested="10.0.18362.0" />
</Dependencies>
```

For more information on MSIX Core, please refer to the Microsoft documentation below:
https://docs.microsoft.com/en-us/windows/msix/msix-core/msixcore

Many third party vendors available in the market enable bundling of MSIX Core and MSIX packages together with the help of a bootstrapper (or using other related techniques) so that the installation experience is seamless on the older versions of Windows.

To deploy the app package manually on an older operating system, you need to make sure MSIX Core is installed on the target device. MSIX Core is an open-source project on GitHub, and the MSI installer for MSIX Core can be downloaded from the link https://github.com/microsoft/msix-packaging/releases.

For more information on installing MSIX Core, refer to the link below:
https://docs.microsoft.com/en-us/windows/msix/msix-core/deploy-with-msix-core

Considerations of MSIX Core:

- MSIX Core does not provide the container benefits of native MSIX; it simply provides a way to deploy the contents of an MSIX package to an older operating system where native MSIX support is not present.
- MSIX Core cannot enable an app that uses Windows 10-specific features to work on previous Windows versions. The app must be built to run on older version of Windows.
- MSIX Core does not support Microsoft Store integration.
- MSIX Core must be installed on the older operating system prior to the deployment of the app package.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- MSIX Core supports Windows Server with Desktop Experience-based operating systems. However, deployment to Windows Server Core is not supported.

3.9 Basic AppXManifest structure

The most important file of all in an MSIX package is the AppXManifest file. The AppXManifest file is an XML file that defines how the application will integrate with the operating system for access by the end-user, including the application model used affecting runtime container features.

While Chapter 7 provides a detailed reference to the AppXManifest schema and syntax, it is important for you to understand the basic structure of the AppXManifest file early on. At a high level, the typical file looks like this:

```
<Package IgnorableNamespaces="uap uap2 uap3 uap10 rescap com"
xmlns:com=”http://schemas.microsoft.com/appx/manifest/com/windows10”
xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/window
s10/restrictedcapabilities"
xmlns:uap10="http://schemas.microsoft.com/appx/manifest/uap/windows10/10"
xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3"
xmlns:uap2="http://schemas.microsoft.com/appx/manifest/uap/windows10/2"
xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10">
     <Identity ProcessorArchitecture="x64" Version="78.0.0.0"
     Publisher="CN=MyCompany" Name="SampleApp"/>
          +<Properties>
          +<Resources>
          +<Dependencies>
          +<Capabilities>
          +<Applications>
          +<Extensions>
     </Identity>
</Package>
```

The following are the important element tags in the AppXManifest file:

Package element tag

The AppXManifest file will always have one, and only one, Package element as the root XML element node. The attributes shown are some of the typical schemas included in many packages. More detail on the schemas available and syntax for you to add additional schemas is given in the schema chapter.

The ordering of the sub-elements under Package are not important, but most tooling generating an AppXManifest file will place them in an order similar to the example above.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Identity element tag

Every package must have an Identity element, which is used to define the identity of this package.

The attributes in this tag--Name, Publisher, and Processes Architecture--help Windows OS to uniquely identify this application at the Package Family Name. This (along with the Version attribute) is analogous to Product Code in a traditional MSI setup. Windows OS uses the properties in the Identity tag to know if an application is installed on a device. The Version property is used by the OS to identify if the application is on the current and latest version or needs an update. The Publisher name is taken from the Subject Name of the digital certificate that is used to sign the MSIX package.

```
<Package …>
     <Identity ProcessorArchitecture="x64" Version="78.0.0.0"
     Publisher=”CN=MyCompany” Name=”SampleApp”/>
     …
</Package>
```

Properties element tag

This required section of the AppXManifest has attributes that define the way the package appears to the users upon installation; for example, when using the AppInstaller GUI.

The PackageIntegrity sub-element is optional. See 7.4.4 Sub-elements for more information.

```
<Package …>
     …
     <Properties>
          <DisplayName>SampleApp</DisplayName>
          <PublisherDisplayName>My Company</PublisherDisplayName>
          <Description>This is a sample application created for
          testing</Description>
          <Logo>Assets\StoreLogo.png</Logo>
          <uap10:PackageIntegrity>
               <uap10:Content Enforcement="on"/>
          </uap10:PackageIntegrity>
     </Properties>
     …
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Resources element tag

The resources element is where support for languages, display scales, and DirectX feature levels are declared. Often, this is declared here in conjunction with a resources.pri file that is contained in the package. Below is an example of using multiple languages to support changing the text seen when installing the package via AppInstaller on operating systems running different language packs:

```
<Package …>
     ...
     <Properties>
          <DisplayName>ms-resource:Name</DisplayName>
          <PublisherDisplayName>ms-
          resource:Publisher</PublisherDisplayName>
          <Logo>Assets\StoreLogo.png</Logo>
     </Properties>
     <Resources>
     <Resource Language=”en-US”/>
     <Resource Language=”ja-JP”/>
     <Resource Language=”fr-CA”/>
     </Resources>
     …
</Package>
```

This is a required element and must support at least one language.

Dependency element tag

This required section contains a list of dependencies of an MSIX package. The following five types of dependencies are supported by MSIX:

- Target Device Family: Defines the target devices this package is meant to deploy. This could prevent installation if the system does not meet the minimum levels, or limit runtime functionality added in later OS versions.
- Package Dependency: These are separate packages that are required in order to install this package. Generally, these must be separately installed prior to installation of this package; however, certain scenarios and/or tooling may trigger this to occur automatically.
- Main Package Dependency: A separate package for which this package acts as a Modification package. Used in packages that are data/configuration only, or plug-ins. The main package must be installed prior to this package.
- Driver Dependency: The driver must be present or available for automatic deployment triggered by installing this package.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- OS Package Dependency: Allows specification of OS build version requirements. This is a new addition to the schema sets, and it is not yet clear how this interacts with the TargetDeviceFamily dependency.

The most common usage of this element includes only the required TargetDeviceFamily sub-element:

```
<Package … >
     …
     <Dependencies>
          <TargetDeviceFamily Name=”Windows.Desktop”
          MaxVersionTested=”10.0.19041.1”  MinVersion=”10.0.17763.0”/>
     </Dependencies>
     …
</Package>
```

Note that both the version fields specified may impact certain runtime features of your application. Microsoft only documents this with the feature impacted.

Capabilities element tag:

Capabilities are declared in the AppXManifest to gain access to certain Windows OS features, such as picture, music, and internet. The application will get access to the Windows API of these features only when the required capabilities are declared in the AppXManifest. When a user installs a package from the Store, the user is notified about the capabilities that are declared in the AppXManifest file of the app, so the user is aware of what features of this device and OS the app will have access to. Following are the different types of capabilities supported by MSIX:

- General-use capabilities
- Device capabilities
- Restricted capabilities
- Custom capabilities

MSIX packages containing Win32, .Net Framework, or Windows App SDK code require the runFullTrust capability to be able to run non-UAP code:

```
<Package … >
     …
     <Capabilities>
          <rescap:Capability Name=”runFullTrust” />
     </Capabilities>
     …
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Applications/Application element tag

The Application element defines the entry points of your package. These are often analogous to the shortcuts in a traditional MSI or EXE setup types, but they may also be for other programmatic entry points such as file type associations and protocol handlers. The EntryPoint and Executable attributes define the main process that should be invoked when the application is launched. The ID attribute is required to reference this application in the AppXManifest or while using developing fixups using Package Support Framework.

The Applications element is simply a container for one or more Application nodes as sub-elements. Each application node has other sub-nodes like VisualElements and Extensions. VisualElements defines what graphic image should be displayed for the application at different locations like start menu, splash screen, etc.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Here is an example that adds a Start Menu item and also defines several application extensions:

```
<Package … >
     …
     <Applications>
           <Application EntryPoint="Windows.FullTrustApplication"
           Executable="VFS\ProgramFilesX64\SampleApp\sampleapp.exe"
           Id="SAMPLEAPP">
                 <uap:VisualElements Description="SampleApp"
                        Square44x44Logo="Assets\SAM PLEAPP-
                         Square44x44Logo.png"
                        Square150x150Logo= "Assets\SAMPLEAPP-
                         Square150x150Logo.png"
                        DisplayName="Sample app"
                         BackgroundColor="transparent">
                            <uap:DefaultTile
                            Square71x71Logo="Assets\SAMPLEAPP-
                            Square71x71Logo.png"
                               Square310x310Logo="Assets\SAMPLEAPP-
                                Square310x310Logo.png"
                               Wide310x150Logo="Assets\SAMPLEAPP-
                                Wide310x150Logo.png"/>
                 </uap:VisualElements>
                 <Extensions>
                            +<com:Extension Category="windows.comServer">
                            +<uap3:Extension
                            Category="windows.fileTypeAssociation">
                            +<uap3:Extension
                            Category="windows.fileTypeAssociation">
                            +<uap3:Extension Category="windows.protocol">
                            +<uap3:Extension Category="windows.protocol">
                 </Extensions>
           </Application>
     </Applications>
     …
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Extensions element tag

Extensions are of two types: package-level extension and the application-level extension seen previously. Application-level extensions appear under the application node, while package-level extensions appear alongside the application node.

- Package-level extensions generally intend to have an effect on the entire package; however, for some categories the effect may be system-wide.
- Application-level extensions generally intend to affect only the application they are part of, although they may affect the entire package, or even system, on certain extension categories.

Extensions are a way for an application to integrate with the operating system to be able to support this package and the end-user using it. Extension is like an agreement between an app and the operating system to extend the standard features of operating system for use in the app; for example, to define additional application exceptions to the Windows firewall.

```
<Package …>
     …
     <Applications>
          …
     </Applications>
     <Extensions>
          <desktop2:Extension Category=”desktop2:FirewallRules”>
               …
          </desktop2:Extension>
     </Extensions>
</Package>
```

There are several package level extension types supported by MSIX. See 7.8 Package extensions element for details.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Chapter 4: Project Activities

We assume that you, as a developer, are familiar with the processes to build traditional EXE and DLL projects, and possibly with building setup/MSI-based installer packages. In this chapter, we will walk you through many of the activities that you may face in converting an existing application into an MSIX package for delivery to your customers. Your particular package may require additional activities not covered here, but we should easily cover 80% of existing packages. Heck, the Windows APIs have been around for 35 years and not even the most seasoned Microsoft developer knows them all.

With moving to MSIX it will be necessary for you to concern yourself both with changes in installer technology, and with how your application accesses things.

From an installation standpoint, MSIX minimizes and controls the changes made by the installation process. All changes made outside the package installation folder are tightly controlled and will need to be specified in the AppXManifest file that is used by the AppInstaller to integrate your application into the system. So where you may have created an lnk file for a shortcut in the past, you will convert this into an Application VisualElements entry in the AppXManifest. Where you may have called runDLL32 to register a COM component in the past, you add a COM element into the AppXManifest file.

From a runtime standpoint, while MSIX allows for your existing EXE and DLL files to run inside the MSIX container without changing out calls to underlying frameworks and user interfaces, that fact remains that running the software inside an isolated container is not the same as what you are used to.

This chapter provides a walk-through of the most important topics that you will need to evaluate and determine if you need to make code-level changes to make your conversion successful.

Many of the sections in this chapter ask you to make an assessment of your existing software for a specific purpose. We recommend that you take a light read through the chapter first to understand all the assessments, and then perform the assessments and go back to read details on what actions you should take.

4.1 The existing EXE and DLL projects

Generally, your existing project files that build EXE or DLL files as output will be OK as they are. It will be changes to the source code inside those projects that you will need to make to conform with MSIX, and we will handle those changes in other sections of this chapter.

The primary consideration for your build project file for these components will be how you want to handle redistributable dependencies. In a non-containerized situation in the past, we would often want to separate out these components, as they are installed only if not already present in the system.

In a containerized situation, our preference is generally to include these in the package. Under MSIX these will remain inside the package and cannot affect any other software. This increases the stability of the customer's system and reduces risk for you. By keeping these components inside your package, you ensure that your app will work no matter what else gets installed.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The downsides to inclusion are your package size and the requirement for you to release a new version of your application should one of those components require a security update.

- As to the size, that only applies to the size of the package you create. MSIX has an automatic deduplication that prevents multiple MSIX/AppX packages with the same files from taking up additional disk space on the end-user's system.

- Major dependencies, like Frameworks and certain Runtimes (like VCRuntime 140), are available as external packages that may be referenced as dependencies rather than included in your package. Referencing external dependencies adds a bit of complication but eliminates the need for re-releasing your package should the dependency need updating.

As a general rule, you should keep Frameworks out of your package and treat them as dependencies. Older VCRuntimes that are no longer under maintenance can be included in the package, although perhaps it is time to upgrade to supported ones anyway.

Likewise, device drivers of any type and software that would run inside the OS kernel is not a candidate for MSIX packaging (although the native installation may be targeted as an external dependency). While database client connector software may be included, including SQL and Oracle clients and odbc/msodbc, the back end databases such as SQL Express or SQL lite themselves must be external. As of the time of writing this book in mid-2021, we don't believe that retargeting your existing projects to .Net 5+ and/or "Project Reunion" is ready enough for consideration on conversion, but see 2.3 Windows App SDK (aka Project Reunion) for the more on that. You might consider it for new EXE/DLL components that you build, but if you are looking at a conversion project, you probably have too much on your plate to take on this additional burden now.

4.2 The question of end-user identity

Although not an MSIX conversion item, any company preparing for a new version of an application that has awareness of user identity needs to review, and possibly update, how identity is handled.

While we are not going to go into what you need to do, because it really isn't specific to MSIX, we do want you to consider the following as part of your project:

- Many applications tend to rely on user identity from the logged in Windows user session. In the past this might have been a local login or one based on Microsoft Active Directory Domain Services. Today this might also include other identity sources, especially Azure Active Directory. Often this doesn't affect your code; however, if you display or prompt for such credentials, you need to consider all modern scenarios.

- Applications that need end-user identity but don't rely on the Windows session (or machine) identity will need to consider multiple identity sources, including not only those above but other new identity sources as well.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT. revenera

- Applications using private identity sources, should consider the possibility of federating to a well-vetted source for identity.
- Any storage of user credential information should be reviewed, secured and monitored.

4.3 User settings

Two closely related parts of applications are data associated with the application and data associated with settings. Both require storage and treatment. With traditional applications, we see the following:

- Often, but not always, the application provides some of the items as part of the application.
- Sometimes, it is created by the app dynamically as part of the installation.
- Sometimes it is created/modified by an organization as part of how they deploy the application.
- Sometimes it is created/modified by the end-user using the application.
- Sometimes it is used external to the application.
- Rarely, an additional application can set it to affect future use of the app.

4.3.1 Assessing your current application

An early task that should be part of your conversion activity is to assess how your existing application uses and stores configuration and data. This assessment should include:

- Where settings and data are currently stored.
- Whether enterprise customers typically make changes as part of deployment.
- Whether the end-user needs to modify settings and data delivered with the app.

Based on what you find, this section should be consulted so that you can form a plan on how to ensure the application works as smoothly while in the MSIX container.

4.3.2 Handling application settings

Focusing specifically on your application's settings, the Microsoft-recommended methods for storing settings have changed over time. The primary methods used for storing settings include:

- INI files
- Other structured format files (XML, json, etc.)
- Files with structure known only by the application
- Windows Registry (HKLM and HKCU)

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- New UAP interfaces for file-based settings

For years, Microsoft has been providing guidance that developers should stop using the Registry and move back to some sort of file-based method. This is primarily due to registry bloat and flotsam left behind. With modern apps, we have a new type of registry hive, the application hive which directly addresses the reasons Microsoft stated for pushing file-based settings.

Applications that currently use the registry should be able to work without code modification in most cases. The application hive acts as an overlay to the existing system hives, and your code should work unmodified. However, there are some bad practices we see in some existing code that may have worked in the past, but you will need to fix them in your project:

- The HKLM hive is not writable, even with elevation. You will need to migrate these to HKCU.
- Keys in the HKCU part of your application hive are more restricted than the native Windows registry. Specifically, you may only ask for enumerate, read, and write permissions. We do run into code from time to time that opens these keys for full access, and under MSIX these calls will fail. Even worse, the developers might not check the error, as perhaps it never failed in the past, and then that code will now attempt to use a null key handle. In this case, the right thing to do is determine what access the code really needs and request only that. The easy thing to do is to ask for "Maximum_Allowed" access, and you'll get the key with whatever rights the OS is willing to grant (which under MSIX is currently enumerate, read, and write).
- Keys in the HKCU part of your application hive may not be deleted. And this includes new keys that were created by the application at runtime. We sometimes see applications that use the registry as a scratch pad this way and clean up. Instead, you probably want to add a new item under the temporary key that invalidates the content. The item can be a simple Boolean that you create/write to.

For file-based settings, we do not yet have good and simple guidance on what developers should do. Moving settings back into the application hive may be a viable option. We can, however, discuss the file-based settings issues a developer should consider during a conversion along with file-based data.

4.3.3 Handling data

By default, data files inside the package are not writable. If these files need to change, a copy of the file must be made outside the package. This is not done by the install but by runtime: either using your own code or using the Package Support Framework to help automate it for you.

Additionally, if your current installer places data files in the AppData/Local or AppData/Roaming folders, your MSIX project will initially need to place the non-writable file elsewhere in the package, and then you may copy to external locations at runtime.

If you have a small number of data files to work with, handling them in your runtime code yourself is probably the easiest way to go. But if you have a lot of files or are unable to perform a proper

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

assessment, building the PSF and configuring the FileRedirectionFixup is an easy way to make all package files writable using a copy on write method that automates the copying and redirection. In that case, your runtime code simply writes to the file as before.

4.3.4 Installation time customization

The traditional installer (setup EXE or msi) had the ability to include install time logic that could write settings and data based on the environment or parameters specified. The installer also had to consider the choice of per-machine or per-user installation.

The MSIX installation is always per-user. There is a pre-deployment model of installing to all users, but this simply causes per-user installation as soon as that user next logs in. So, you should think of it as always per-user. Additionally, custom install logic, either in the form of MSI custom actions or coded logic in a setup EXE, is not possible.

What is possible under MSIX instead is logic tied to the first use of the application. Here, you have a couple of options.

4.3.5 Using the PSF scripting capabilities for customization

When converting older applications, we tend to use the PSF PsfLauncher scripts for this purpose. This method provides for the inclusion of a PowerShell-based script file that is triggered by the user starting the application via PsfLauncher. In effect, the shortcut that you have today is modified to point to PsfLauncher, and PsfLauncher is configured to run this script. The script can be run at application startup and/or termination and has an optional setting to run only once. The start script with "run once and wait" parameters will ensure that this part of your traditional custom install activities takes place before your application starts up.

4.3.6 Coded logic in your app initialization for customization

But with access to the source code, you can also embed the login inside your initialization routines. You may easily implement your only “run once” logic by having this code look for a marker (file or registry) that is not part of the package to trigger the one-time initialization, and this one-time initialization will add the marker in addition to other customizations.

Coding this logic in your source code is simpler than bringing in the PSF, but locking the customization away in binary code makes it less flexible, both for you and your customers. Implementing the coded logic in your code to trigger the need and calling some sort of customizable script external to the EXE should be considered.

4.3.7 Considerations for customization

A major consideration on all file-based settings and data is where to place them. In the past, writing files to the program directory became taboo because of OS protections, and writing to the AppData Local or Roaming folders (as appropriate more common when the settings/data are primarily intended for this

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

application only. Today, it is possible to place these files inside the program directory and either use the PSF FileRedirectionFixup (see 2.2.2 PSF features) or InstalledLocationVirtualization (see 7.8.6.4 InstalledLocationVirtualization) to allow the files to be modified by the runtime redirecting to a copy made in the user profile application area.

VFS layering in the package does not extend to the end-use AppData local or roaming areas, so you cannot simply include files for those areas in the package file system (meaning in the package as files under VFS/AppData, for example) as you might have in the past. If you want to include files under VFS/AppData or VFS/LocalAppData, then you must use the PSF FileRedirectionFixup to overcome this limitation. One technique to avoid this may be to have such files initially stored elsewhere in the package and copied on first run.

4.3.8 Modification package

In particular, developers who sell to large organizations need to consider that unlike consumers who install their own software, larger business tend to utilize their IT staffs to bring in your application and customize the settings as part of how they deploy the application.

If you provide an MSIX package, these customers will be able to create a modification package that acts as an add-on to your package. This modification package has the ability to add files, replace entire files, and add registry settings. For them to replace files, these files must exist in the package under VFS folders. A common technique to allow for this, also useful for allowing plug-in files, it to structure your package to not place the application files directly at the package root, but to place them in a VFS folder such as VFS\ProgramFilesx64\Vendorname.

Even with a structured setting file, like an INI file, the customer would need to take your package INI file, make appropriate changes, and add the entire replacement file in the modification package. Alternatively, if you revert to registry-based settings, the customer can add only those values he or she cares about in the modification package. This significantly increases the likelihood of that modification package being usable against your upgraded package next release.

4.3.9 New UAP setting APIs

In UAP, Microsoft created some new APIs for apps to store settings. A novel idea, the app can specify the name of the file but not where it is stored. These APIs store the data in a special area in the user application profile (different from where the PSF FileRedirectionFixup or InstalledLocationVirtualization places them). If the user is logged in to Azure Active Directory and the Azure instance has enabled "Enterprise State Roaming," then those settings are automatically migrated to Azure (likely just an area in One Drive). These two limitations should give the developer something to think about before depending on those capabilities.

Enterprise State Roaming has more recently been deprecated and replacements are not well understood at this time. Check Microsoft documentation for the latest developments.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.4 Registry settings and data

Your traditional installer may include registry items to be written as part of the installation. Normally, these items are written to either the HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER hives, although writing to the default user's hive is sometimes done.

As discussed in the previous section, under MSIX your app and installer will no longer write to the registry directly, but instead use a new type of hive referred to as the application hive. By moving the registry usage to the application hive this prevents conflict and improves the overall performance and stability of the OS.

Because some types of these registry changes are not for your application code to use but to help others, especially the Windows File Explorer. It integrates with your application and some of the items that your installer writes to the registry will end up needing to be migrated to the AppXManifest. File Type Associations are great examples of these.

4.4.1 Functional differences

Prior to the OS 2004, your application could not modify any entries defined in the application hive. New items could be added and modified, but the existing items were immutable.

As of OS 2004 and above, HKCU items defined in the package may now be modified (but not HKLM). However, this change comes with certain limitations, namely that MSIX isn't going to allow deletion, ownership changes, or other advanced operations against those items. Even some seemingly benign things your code might do, such as open HKCU\Software\Vendor key for full access, will now be denied (because that includes the right to delete the key once you have the handle with those accesses). Modifying your code remains the best approach to resolve these issues, but there is a PSF fixup called RegLegacyFixups to address certain common issues.

4.4.2 Migrating to the dat

To use the application hive, your code references to registry locations do not change, but the initial registry structure and data normally added by your current installer will be migrated into the package in a new form, which will be a registry "dat" format file (or files). The primary file will be named Registry.Dat and will be placed in the root folder of the package. This file will contain all registry items that you want to create. The two additional files are technically optional, but we recommend building them as a filtered subset of the Registry.Dat file and placing them at the root of your package as User.Dat and UserClasses.Dat.

The Registry.Datfile has two top-level keys: one called MACHINE and the other USER. These keys are equivalent to HKLM and HKCU. So if your existing installer adds a key HKLM\Software\Vendor, this would appear in the file as MACHINE\Software\Vendor.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The User.Dat file should contain all entries from the USER key of the Registry.dat file. The UserClasses.Dat file should contain all entries from the USER\Software\Classes key.

4.4.3 Creating the dat file(s)

The registry dat format has been around for quite some time, and there is the possibility that your setup installer tool might support you with interfaces identical to how you requested them in the past, but instead of adding the entries into the MSI database they will create the registry.dat file for you. The Microsoft WAP and InstallShield are great examples of installer builder programs supporting this.

If your setup installer tool doesn't directly support your needs, you can always make your own dat file by exporting out of the RegEdit tool. To do this, use RegEdit:

- Create a temporary key anywhere convenient.
- Under that key create two subkeys, MACHINE and USER.
- Under MACHINE, add entries you'd normally add under HKLM, such as SOFTWARE.
- Under USER, add entries you'd normally add under HKCU, such as Software.
- Entries you may have previously added for all users should now go under the USER key.
- Entries for things to be migrated to the AppXManifest may either be skipped or included.
- When ready, click on the temporary key, and from the RegEdit menu select File→Export to save the file to one named Registry.Dat.
- Click on the USER key under the temporary key and export this to User.Dat.
- Click on the USER\Software\Classes key and export to UserClasses.Dat.

We have found no documentation on when the User.Dat and UserClasses.Dat files are required, nor how these are used. We have seen that certain features, like some shell extensions, seem to need to have the equivalent registry items present in these files. We have also seen that having the User.Dat file without the UserClasses.Dat file causes AppInstaller to fail; adding a UserClasses.Dat file with just a dummy value seems to overcome this error.

4.4.4 Considerations for what goes into the Registry.Dat

There are many items that get added into the registry during an MSI or setup EXE installation; that you will need to review them and consider on a case-by-case basis whether they belong in the Registry.Dat file.

In cases where your application expects to find app-related settings, items normally placed in your vendor-specific key under HKCU\Software or HKLM\Software, you will want these in the Registry.Dat file of your package.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

But in cases where those items end up under the Classes keys, the situation could be different.

If those entries are placed in the registry for the purpose of integrating with external software (the OS or other application) to use software from your application, that style of registration must migrate to the AppXManifest file. But for internal use, especially when AppXManifest support does not exist, we might need them in the Registry.Dat.

Take, for example, a protocol handler that might be used for something external to start up one of the EXEs inside your package. For that functionality, the protocol handler must now be identified in the AppXManifest so that AppInstaller will create the appropriate new-style registration. Therefore, the registry changes used in the past for that protocol handler will no longer be needed. File Type Association registrations are a second example. So, for these cases, migrating the description of the needed registration from the MSI tables to the AppXManifest is all that is required, and the handler would be available for both internal or external use.

But not all types of integrations for file associations, handlers, and COM that have existed in the 30+ years of Windows APIs are supported by the MSIX AppXManifest, and not all of those integrations are needed external to your package. For those that are not supported, we can include the actual registry settings from the MSI install inside the Registry.Dat of your package, and these will work like the MSI install for any software running inside the container.

For example, certain kinds of Context Menu handlers (but not all kinds) have been problematic under MSIX. While Microsoft is still addressing the challenge, these handlers do not show the menu when right-clicking on the proper file types external file explorer menus but do work correctly in dialogs from within your package, such a file picker dialogs.
Keep in mind that for common areas like the Classes key that exist under both HKLM and HKCU, your old installer probably used HKLM, but the choice potentially affected other applications and users. Under MSIX, these may be written to either the Machine or User section of the application hive, and these will only affect the application running in this container. You may choose either, but here are some things to consider:

- As the installation is per-user, it seems logical that using the User section of the application hive might be best.
- The local system may have a different entry, and you almost always want to ensure your values are used inside your container. The search for these entries uses HKCR, which overlays HKCU\Software\Classes on top of HKLM\Software\Classes. Your hive User and Machine overlay on top of each of those individually. So, to ensure your package entries win over anything locally installed, you should place them into the User section.
- But if you want to ensure that nothing can change the entries, placing them in the Machine section makes them read-only.

Our experience so far is that when in doubt, even for things now handled through the AppXManifest, continuing to add appropriate entries into the Registry.Dat file is useful to get more functionality when

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

you run into things not yet fully supported through the AppXManifest. It helps the application work in some cases and does no harm other than adding package bloat.

4.4.5 File string references in the registry application hive

If your vendor for the MSIX package creates registry entries for you, they will probably take care of making registry items that have strings in them for file locations that use the appropriate variable substitution syntax. But if you create your own Registry.Dat file, you will need to take care of this yourself. This applies to both REG_SZ and Multi. The Expand variety probably should be converted since the variable substitution syntax covers the same need.

An example is probably all you need to understand. Let's say you have an existing registry entry created during a typical installation of your current installer under some key as follows:

```
WhereToFindStuff   REG_SZ C:\Program Files\Vendor\app\SpecialFolder
```

The variable substitution form in the registry.dat file will replace the starting file path of the registry item value using the same VFS variables mentioned in 3.2.1.2 VFS files:

```
WhereToFindStuff   REG_SZ [{AppVPackageRoot}]\SpecialFolder
```

If you need better examples, you may want to run your current installer through an MSIX repackaging tool, such as the Microsoft MSIX Packaging Tool or Admin Studio. These tools will produce a registry.dat file that you can examine:

- Extract the dat file from the created package.
- Open RegEdit and create a temporary key (such as HKCU\Software\DELETEME).
- Click on the temporary key, then from the RegEdit menu pick File->Import.
- In the file picker, change the types to all registry hives and pick the file.

You could even then use this as a basis for your own registry.dat file.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.5 Start menu

The primary access for your end-user to start your application is typically through the start menu. MSI/EXE-based installers typically create these entries by placing files into specific locations, either the system or current user Start Menu Folder.

These files most often are .lnk files, which have specific properties to describe the ultimate target program. But the Windows Explorer that creates the start menu will also look at any file type placed in these areas and make a shortcut that uses the local file type association to access the file. The most common types of files used include:

File Type Extension	Purpose
.url	The properties describe a website, accessed using the default browser.
.txt	Text documentation, such as “ReadMe” or License files.
.pdf	Documentation.
.chm	Very old form of documentation that you should replace.

In addition to the start menu, there are other locations where these files might be placed for access. These includes the Desktop folder and Quick Launch bar, as well as Task-bar pinning.

4.5.1 Differences in starting modern apps

In modern applications, Microsoft would like to see software vendor installers produce a cleaner experience, both for the end-user and the system.

This starts with the installation experience itself, which does not allow for dialogs and choices. Practices that your current installer uses to install to either all users or the current user are no longer available. Duplication of shortcuts, such as placement in multiple locations in the start menu, are no longer available. Indeed, currently your shortcuts must point to an EXE file and may not even have command line arguments. You will be able to work around these limitations if you need to, but please consider what your application truly needs and how end-users will perceive old practices carried over.

In modern apps, the package no longer creates and adds these files into the Start Menu folder and other locations. Instead, entries describing the desired result are included in the AppXManifest file. You will need to change these methods to modern ones as described in the next sub-sections.

4.5.2 Assessing current start menu needs

In assessing your current application, please consider the following:

- The Quick Launch Bar is obsolete, and the AppXManifest does not contain this capability.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- It is considered a bad practice to place items on the desktop. The installation experience does not give the person performing the installation any choices. The AppXManifest does not contain this capability; however, the end-user can choose to create a desktop shortcut on their own from any start menu item that is displayed.

- Currently, the AppXManifest does not contain the ability to pin to the taskbar as part of the installation, as Microsoft seems to want this to happen only when the end-user requests it. There is an option in the AppXManifest called “windows.desktopAppMigration” (see 4.5.7 Migrating existing installed app shortcuts) for when migrating a traditional installed application that is currently pinned to an MSIX package.

- The modern experience uses a cleaner start menu, and you should consider removing unnecessary shortcuts and replacing them with features directly in your main application. Although Folders have returned to the start menu, they feel out of place, and we are not sure that they will be useful in future iterations of the OS. The modern start menu experience is one of the items sorted alphabetically or searched for by typing. At a minimum, you may want to rename your start menu items to all start with the same letters so they stay together and may be found together.

Although it may seem odd to include this topic here, the use of splash screens must also be addressed. Typically, these are embedded in the program itself, and while you may continue to have that logic built into the app as-is, Microsoft has created an external splash screen that may be defined in the AppXManifest, so you will need to consider if you want to migrate this as well.

The following sub-sections will walk you through additional details that you need to consider updating your end-user entry points and launching attributes. Not all of them are treated the same way.

4.5.3 Visual elements

Application entries in the AppXManifest may include the VisualElements XML element that is used to define a number of things that affect how your application first looks to the end-user.

By replacing the traditional shortcuts into the AppXManifest file, the task of integrating your entry points into the system, and removing them during uninstallation, is removed from tasks performed by the vendor installer logic and left to the system. Additionally, Microsoft has enhanced the visual aspects of how your application appears in various ways in the end-user's start menu experience, and on different systems with a variety of dpi, text scaling and high-contrast settings.

Application entries in the AppXManifest may include the VisualElements XML element to define various graphics that will be used to represent the application entry. This, essentially, is the replacement for start menu shortcuts used in traditional applications. MSIX has no replacement for Desktop or Quick Launch Bar shortcuts. Similarly, there is no equivalent to the MSI “advertised shortcut.”

While traditionally Windows has used icon files (which support multiple sizes inside a single icon file), under MSIX all these graphic files are individually sized png or jpg files.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

You can find documentation on the VisualElements element of the AppXManifest at the following links, but the text that follows will offer some practical advice.

- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-visualelements
- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap3-visualelements-manual

Inside the AppXManifest file will be Application elements that are used to describe all externally available executable components. Each application may have a VisualElements sub-element.

The VisualElements attributes are primarily associated with the start menu. Although it is part of the UAP schema extensions, any MSIX application will also use the UAP schema extensions; these are not restricted to UAP-based applications.

Application element entries in the AppXManifest file that do not require a start menu entry are also possible. These Application elements should include a UAP:VisualElements entry but set the AppListEntry attribute to “none” and not add the DefaultTile sub-element.

4.5.4 Assets and the start menu

As discussed in Assets, the start menu uses different sizes of graphics, for the start menu itself and for pinned items (which may be pinned as small, medium, large or wide). And for any of those we may have different files for systems using different text scaling for high-count DPI monitors. Basically, in the start menu Microsoft treats your graphics as graphics including text, whether or not you have any. So, if the end-user has requested text to be displayed at 400% (a user session-wide setting), the 4x-sized version of your graphic is used (e.g, 310x310.scall-400.png).

It is likely that your existing project uses icon files that have varying sizes of graphics embedded in a single file. You will need to create new graphic files for a variety of sizes, most of which will be larger than you would currently have.

- Someone should first make a high-quality, large-sized, square graphic file in png or jpg format. Without a splash screen, the base image your designer creates is 1240x1240 (310x310 for systems with text scaling set to x400).
- From that image, make scaled-down sizes from that for each of the sizes needed. Generally, a resizing of the largest will be acceptable, but it may be possible that the smaller sizes will need a less complex version of the original image. These will be used for the package logo, all the start menu graphics (except for the “wide” format ones), and badges (if used).
- A second base image of 1240x600 is needed for start menu pinned “wide” format. While you can paste a scaled-down copy of the original image, it will look odd, and a fuller image should be developed.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- From the second image, scaled-down copies are also needed.
- If using the new splash screen, a separate image (2480x1200) for that would also be produced and scaled down.

The background of these graphics may also be set. While a transparent background might have been interesting in the past, user customization to enable “dark mode” or otherwise set the background color of the start menu itself means that you should probably set a background color for your graphic if the file uses transparency.

Microsoft uses the term “unplated” to describe a situation where the graphic is used as is, and “plated” to describe one in which a background gets applied. In some situations, the system will consider the unplated asset but look at the colors to determine if the plated should be used instead. See documentation https://docs.microsoft.com/en-us/windows/uwp/design/style/app-icons-and-logos for a fuller discussion on asset sizes, uses, and recommendations.

Your original project likely included instructions to build .lnk or url files when the installer ran. When building your MSIX-based installer, these would probably go away automatically, but if your existing installer specified lnk or url files in the file list as premade files, you should consider removing them. It is not necessary to remove lnk for url files from your package, but they will serve no purpose.

4.5.5 Splash screens

Adding the SplashScreen element under VisualAssets and the image is only effective if your application contains UWP code to implement the splash screen. If your software doesn’t call these UWP APIs, any entries here are ignored. This setting is independent of any splash screen code that might be in traditional Win32- or .Net Framework-based programs, so it is not necessary to convert to the newer form if your current project works well. Once “Project Reunion” becomes a reality, it might be possible to convert your code over if desired.

4.5.6 Rotation preference

The InitialRotationPreference element that may be added under VisualAssets is only effective at this time if your application is UWP based. The purpose is to list supported rotations and set a default (for devices without sensors) on a per-application basis (potentially overriding the package level). It will be ignored otherwise.

4.5.7 Migrating existing installed app shortcuts

It is common for software installers (msi/EXE) to have multiple shortcuts that appear in Start Menu, Program Menu and/or Desktop. In many cases, users would pin some of their favorite apps to Task Bar for ease of access. These shortcuts may call either the same program or different programs, within the installer, depending on how the shortcuts are configured in the installer by the developer.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Under MSIX, an application is only allowed to add a single entry to the start menu, even though the feature of pinning shortcuts still exists in the OS. Microsoft has indicated that they want to leave pinning up to the end-user. Microsoft did, however, give us a feature that maintains the pinned experience if the end-user is upgrading from a traditional msi/EXE version of the app to an MSIX one.

For your existing users with pinned shortcuts to continue to have the same experience of launching and using their applications, as a developer, you have to leverage the windows.desktopAppMigration application extension in the AppXManifest file of your MSIX package. This item will point to the disk locations of the existing shortcuts and migrate them to point to the new application.

Windows uses Package Identity to determine if an MSIX package is already installed on a machine. In a traditional installer (msi/EXE) this Package Identity does not exist, so Windows will not be able to avoid having a traditional installer and the MSIX package for a given application of the same version co-exist on a machine. A scenario like this will result in confusing the users when they see two applications of the same version on the machine. To avoid such confusions and any runtime issues, you would want the user to always use the application installed by MSIX package without changing user behavior of launching and using the application. The windows.desktopAppMigration extension comes in handy in such scenarios, where the shortcuts installed by the traditional installer will start to point to the MSIX package installed application.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.5.8 Examples of AppXManifest replacement

Here are two examples of AppXManifest replacements.

4.5.8.1 Visual Element Example

Below is a sample of an AppXManifest for an application that converted from a .Net. Framework-based application. The tooling used automatically added the unnecessary SplashScreen and InitialRotationPreference sub-elements; they are shown here for completeness.

```
<Package … >
     <Application EntryPoint="Windows.FullTrustApplication"
      Executable="ProgramApp.exe" Id="Program">
          <uap:VisualElements AppListEntry="default"
                Description="Application Description"
                Square44x44Logo="Assets\YourApplicationSquare44x44Logo.png"
                Square150x150Logo="Assets\YourApplicationSquare150x150Logo.
                png"
                DisplayName="TheProgram" BackgroundColor="transparent">
                      <uap:DefaultTile
                         Square71x71Logo="Assets\YourApplicationSquare71x71L
                         ogo.png"
                         Square310x310Logo="Assets\YourApplicationSquare310x
                         310Logo.png"
                         Wide310x150Logo="Assets\YourApplicationWide310x150L
                         ogo.png"/>
                      <uap:SplashScreen Image="Assets\YourApplication
                         SplashScreen.png"/>
                      <uap:InitialRotationPreference>
                            <uap:Rotation Preference="portrait"/>
                            <uap:Rotation Preference="landscape"/>
                      </uap:InitialRotationPreference>
          </uap:VisualElements>
     </Application>
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.5.8.2 DesktopAppMigration example

Here is an example of how to add the windows.desktopAppMigration extension to the AppXManifest.XML file. This is an application extension; therefore, it must be added under the <Application> element:

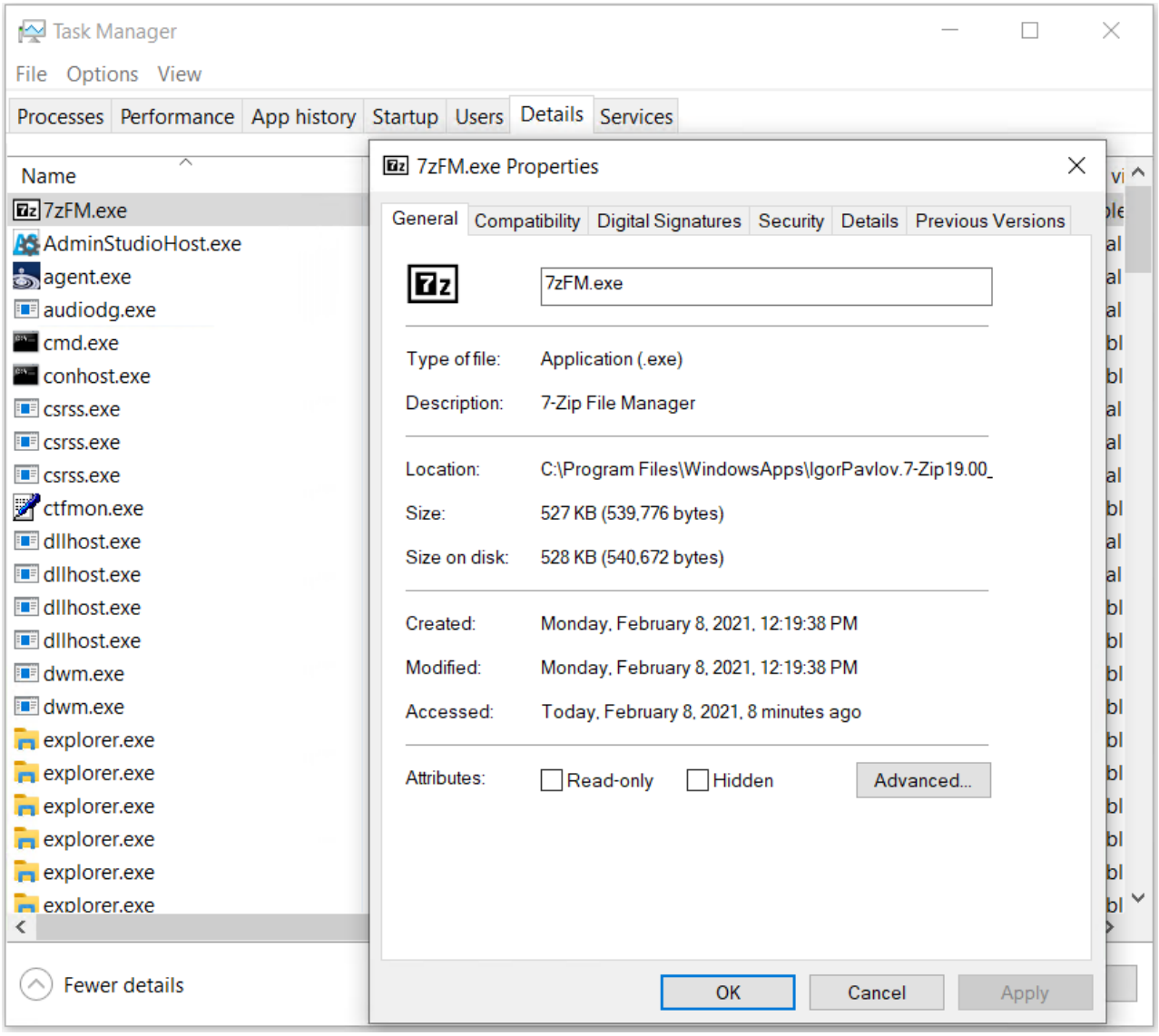
```
<Package …>
     …
     <Applications>
          <Application>
               <Extensions>
                    <rescap3:Extension
                    Category=”windows.desktopAppMigration”>
                         <rescap3:DesktopApp
                         ShortcutPath=”%USERPROFILE%\Desktop\7-
                         Zip File Manager.lnk”/>
                         <rescap3:DesktopApp
                         ShortcutPath=”%APPDATA%\Microsoft\Windows
                         \Start Menu\Programs\7-Zip\7-Zip File
                         Manager.lnk”/>
                         <rescap3:DesktopApp
                         ShortcutPath=”%PROGRAMDATA%\Microsoft\Windows
                         \Sta rt Menu\Programs\7-Zip\7-Zip File
                         Manager.lnk”/>
                         <rescap3:DesktopApp
                         ShortcutPath=”%APPDATA%\Microsoft\Interne
                         t Explorer\Quick Launch\User
                         Pinned\TaskBar\\7-Zip File Manager.lnk”/>
                    </rescap3:Extension>
               </Extensions>
          <Application>
     </Applications>
</Package>
```

In this example, 7-Zip.msi was packaged to MSIX format. While 7-Zip installs a shortcut in the Start Menu, there was an additional shortcut created on the Desktop and pinned shortcuts on the Task Bar and Start Menu. The AppXManifest file of 7-Zip MSIX package was updated to add the Windows.desktopAppMigration extension with all the shortcuts that were installed and pinned as shown above.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

After the updated MSIX package was installed on the machine, all the shortcuts started pointing to the application installed by the MSIX package. The original installer (msi/EXE) could be uninstalled later to avoid having multiple applications of the same version running on a machine. The pinned shortcuts continue to exist and launch the MSIX package even after uninstalling the original installer. The screenshot below shows the 7-Zip File Manager program being launched from the WindowsApp folder where the MSIX package is installed.



Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.6 File type associations

File associations define how the shell treats a file type on the system; for example, which application launches when a file is double-clicked, which icon appears for a file by default, etc. Applications use these file associations to control how the Shell handles file types.

When you right-click on a file, you will see a context menu. This menu shows a list of commands that you can use to perform various actions on a file. Each command on this menu is identified in the registry by its verb. A verb is a simple text that is used by the Shell to identify its associated command, which defines how a file should be handled and by which application. A verb may have a display name that is displayed in the context menu instead of the verb string itself; for example, 'openas' is displayed in the context menu as 'Open With.'

If you wish to associate one or more file types with your application, you must define a programmatic identifier (ProgID) for each file type. The ProgID entries used for file association are stored in HKLM\SOFTWARE\Classes and HKCU\SOFTWARE\Classes.

When a feature containing file associations is installed on a machine, the file associations are registered on the machine and entries are made into the registry. These registry keys link the file types with your application through a ProgID. A ProgID uniquely identifies an application and helps the operating system recognize file associations.

Microsoft has introduced an extension at the application level to add file type associations to MSIX packages. To handle a file type in your MSIX package, you will have to declare a file type association extension in the package AppXManifest file under the application node, which will handle the said file type. For details on syntax and attributes of file type association extension, please refer to:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-filetypeassociation

Here is an example of a sample application PDF Editor for editing PDF files. The MSIX package has a file type association for the file type extension .pdf, and here is how it is declared in the package AppXManifest:

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

```
<Package …>
     …
     <Applications>
           <Application EntryPoint="Windows.FullTrustApplication"
            Executable="VFS\ProgramFil esX64\PDF Editor\pdfeditor.exe"
            Id="PDFEditor">
                 <uap:VisualElements Description="PdfEditor"
                    Square44x44Logo="Assets\PDFEditor-
                    Square44x44Logo.png"
                    Square150x150Logo="Assets\PDFEditor-
                    Square150x150Logo.png"
                    DisplayName="PDF Editor"
                    BackgroundColor="transparent">
                       <uap:DefaultTile Square71x71Logo="Assets\PDFEditor-
                         Square71x71Logo.png"
                         Square310x310Logo="Assets\PDFEditor-
                         Square310x310Logo.png"
                         Wide310x150Logo="Assets\PDFEditor-
                         Wide310x150Logo.png"/>
                 </uap:VisualElements>
                 <Extensions>
                       <uap3:Extension Category="windows.fileTypeAssociation">
                             <uap3:FileTypeAssociation Name="pdf">
                                   <uap:SupportedFileTypes>
                                         <uap:FileType>.pdf</uap:FileType>
                                   </uap:SupportedFileTypes>
                                   <uap:Logo>Assets\htm.png</uap:Logo>
                                   <uap2:SupportedVerbs>
                                         <uap3:Verb Id="open"
                                            Parameters="–single argument %1" >
                                               open
                                         </uap3:Verb>
                                   </uap2:SupportedVerbs>
                             </uap3:FileTypeAssociation>
                       </uap3:Extension>
                 </Extensions>
           <Application>
     </Applications>
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.7 Protocol handlers

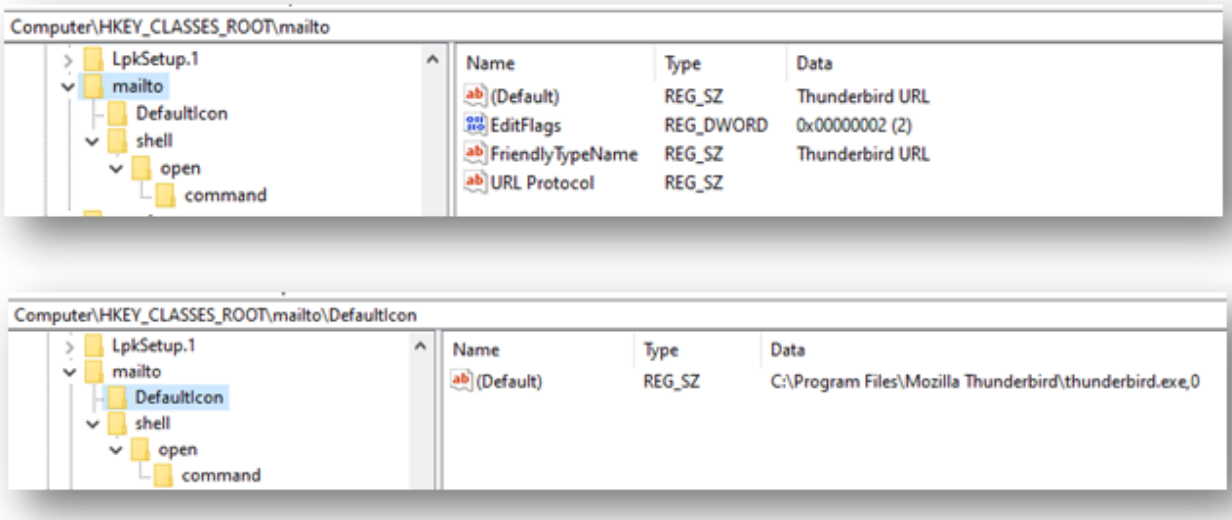
In addition to an end-user starting an application, sometimes applications provide entry points for use by themselves or other software in the form of a Windows Protocol Handler. Once defined and registered in the system, this allows the handler to be invoked through the use of a URI scheme. Examples of common URI schemes you may already be familiar with include HTTP:\\, FTP:\\ and MAILTO:\\ .

If your existing application includes a protocol handler itself, the code and module that you currently have are probably OK (other than the note about in-process COM handlers below), but how this component becomes registered will be different under MSIX.

4.7.1 Assessing your existing application

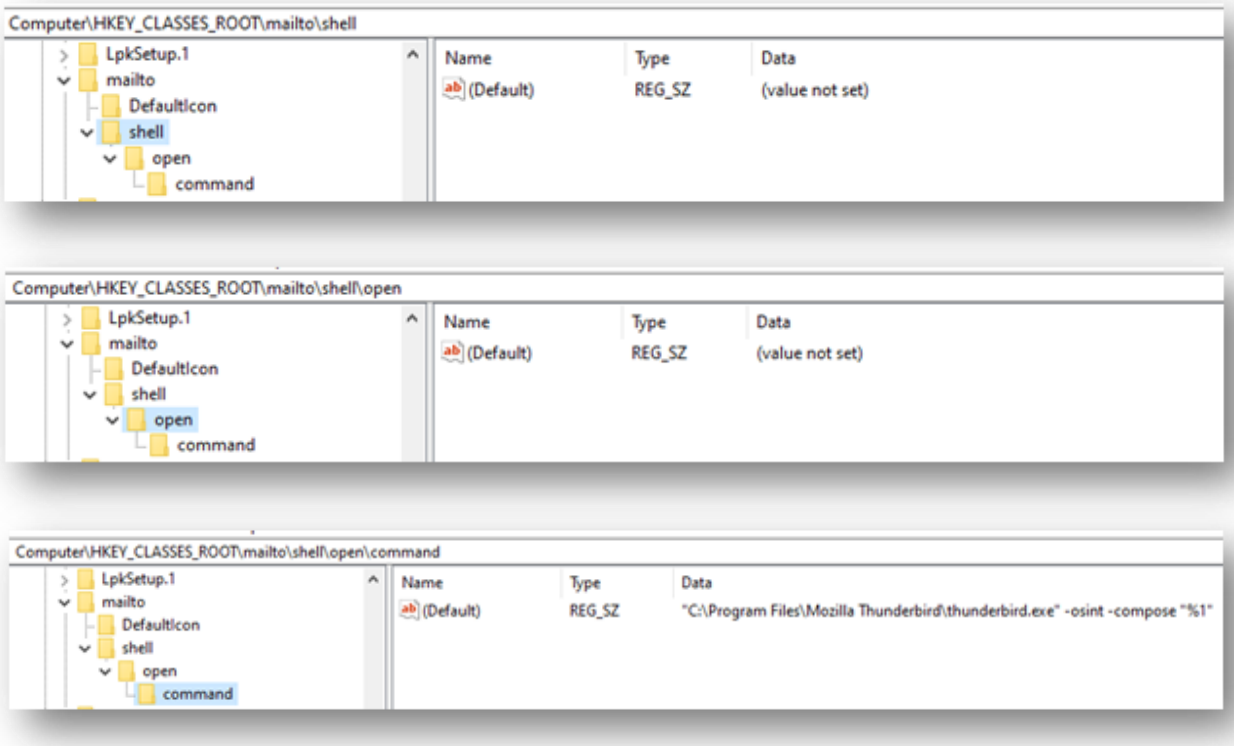
Asynchronous Pluggable Protocols is a way for an application to handle Uniform Resource Identifier (URI) schemes. For an application to handle a particular URI scheme, it needs to be registered as a URI protocol handler and associated with the URI scheme it is meant to handle. In a traditional installer like MSI or EXE, to register an application to handle a particular URI scheme, a certain number of registry keys and subkeys should be added to the HKEY_CLASSES_ROOT hive of Windows Registry. The root key name under HKEY_CLASSES_ROOT must match the name of the URI scheme; for example, to add a mailto scheme, add a "mailto" key under HKEY_CLASSES_ROOT. Under this new key, a string value with the name url Protocol must be added to declare a protocol handler. The value should be an empty string. Without this key, the handler application will not launch.

There are other subkeys required to register an application as a protocol handler. Here is an example of how Thunderbird registers itself to handle a mailto URI scheme.



Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera



4.7.2 Changes to support protocol handlers in MSIX

Registration activities handled by your existing installer are not applicable, but you should not need to take action to remove them. You will need to add a registration request into the AppXManifest file, which will appropriately take care of registration and deregistration on your behalf.

To handle URI schemes in MSIX, a windows.protocol extension must be declared in the package AppXManifest file associated with the application. This (windows.protocol)is an application extension, so it must be declared under the application element that is meant to handle the declared URI scheme.

4.7.3 Example of integrating the protocol handler

For an example of how windows.protocol is added to the AppXManifest file, we will show you how you might handle the mailto URI scheme for Mozilla Thunderbird when the existing installer is converted to MSIX. We can start with the package AppXManifest that contains an EXE with a shortcut (converted to an application with visual elements entry) as shown here:

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

```
<Package …>
     …
     <Applications>
          <Application EntryPoint=”Windows.FullTrustApplication”
          Executable=”VFS\ProgramFilesX64\vendorName\appExeName.e
          xe”
          Id=”APPEXENAME”>
               <uap:VisualElements …>
                    …
               </uap:VisualElements>
          </Application>
     </Applications>
</Package>
```

And then add to it as shown here:

```
<Package …>
     …
     <Applications>
          <Application EntryPoint=”Windows.FullTrustApplication”
          Executable=”VFS\ProgramFilesX64\Mozilla Thunderbird\
          thunderbird.exe”
          Id=”THUNDERBIRD”>
               <uap:VisualElements …>
               </uap:VisualElements>
               <Extensions>
                    <uap3 :Extension Category=”windows.protocol”>
                         <uap3:Protocol Name=”mailto” Parameters=”
                           Parameters=”-osint -compose “%1””>
                             <uap:DisplayName>mailto</uap:DisplayName>
                             <uap:Logo>Assets\eml.png</uap:Logo>
                        </uap3:Protocol>
                    </uap3:Extension>
               </Extensions>
          </Application>
     </Applications>
</Package>
```

This adds the protocol handler as an application extension after the VisualElements entry using the Windows.Protocol category extension element. Although this category exists under the UAP3 schema, this does not appear to be UWP code.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.7.4 com-based protocol handlers

Some protocol handlers are com-based components that require additional registration for the COM components.

Protocol handlers that are implemented as COM objects will require being attached to an Application element of the AppXManifest. This attachment is simply to make the schema syntax valid and does not actually associate the protocol handler DLL with the executable listed in the Application. The executable parameter for the Application entry must end in EXE, so you have a couple of choices on how to resolve this:

- Attach the windows.protocol extension to any application element already present in your AppXManifest. This does not actually tie the handler to that executable but simply solves the problem.
- Create a new application entry for it with the application element executable parameter listing a non-existent EXE file name. This also resolves the schema needs and works, with the potential of confusing the next developer who can't find that referenced EXE file (so use an obvious name).

Note that protocol handlers that use in-process COM (as opposed to out-of-process or DCOM/ remote-com) execute in the caller's process rather than inside your package container. Depending on the design, this could lead to issues as things like the virtual registry will not be available to your handler code. If you use in-process COM for your handler, you will need to thoroughly test and may need to re-architect your code.

See Chapter 7: AppxManifest Schema for handling the COM registration needs.

4.7.5 Protocol handler references

https://docs.microsoft.com/en-us/previous-versions/windows/apps/hh452686(v=win.10)
https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/appxmanifestschema/element-protocol

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.8 COM support

Historically, Win32 applications would create Common Object Model (COM) based components that other Win32- and Dot Net Framework-based applications could use. COM programming has been around a long time as a way for components to expose interface functions to other software in a manageable way. The software component that wants to use COM is referred to as the COM Client, and the part that implements the functionality is the COM Server.

The COM extensions in the MSIX Package AppXManifest file provide a new way to register and install the COM components and also help expose COM registrations to clients both inside and outside the MSIX package as necessary.

4.8.1 Traditional vs. Packaged apps and COM

When it comes to distributing COM components, traditional Windows installer-based setups would use either COM extraction or self-registration methods to install and register the in-process and Out-of-process COM server components so that they may be found by the calling code.

In both approaches, the Windows registry maintains information about all the COM objects installed in the system at two levels: Interfaces and Servers. Depending on how it is installed, this registration would add entries in either the HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER hive under the following two keys:

HK??\Software\Classes\Interfaces
HK??\Software\Classes\CLSID

The COM Client only knows about interfaces, which are roughly the equivalent to function names in traditional programming and passes an interface GUID to an API to call through that interface. The COM Service is responsible for resolving that interface GUID into the CLSID that implements that interface. The CLSID GUID then defines the component location and how it should be instantiated. After determining the component's server, Windows either loads the server into the process space of the client application (in-process components) or starts the server in its own process space (local and remote servers). The server creates an instance of the component and returns to the client a reference to one of the component's interfaces.

When it comes to an MSIX package that contains a COM Server component, if that component is only required for its own personal use within the container, then you can rely on the traditional COM registry entries if you add them into the package's application hive (Registry.Dat and User.Dat files) to support your app. All binaries in the same package have access to that registry. But keep in mind that self-registration at the end-user system may be an issue because of MSIX, so you would want to add the registration entries into the package's application hive of your package for this type of usage.

If the COM Server component in your MSIX package needs to be exposed to software outside your package, then it is necessary to include the COM extension style of registration that is available in the

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

AppXManifest file. Including these extensions in your AppXManifest will also support use by COM clients inside your package whether or not you add the entries into the package's application hive.

Vendor tooling will often take care of creating the entries that you need in the AppXManifest file, but there are a few issues to be aware of before starting:

- This AppXManifest registration may require additional knowledge about the nature of the COM component that was not required in the traditional registration.
- There may be restrictions imposed that negatively impact traditional Win32 in-proc and out-of-process COM components.
- Certain types of special COM Components, such as shell extensions and COM-based protocol handlers, require both the COM extension registration plus other special- purpose registration in the AppXManifest file.
- In-Process COM servers accessed by a COM client outside the container (for example Shell Context Menus) do not currently have access to the applicationhive of the registry. We have seen this cause issues for some shell context menu handlers, for example, that need to read entries in HKLM\Software\VendorName that the application normally adds to the system and are now not visible to the explorer process running outside the container.
- Often COM components are used by multiple applications within the same package. Uniqueness restrictions in the schemas require that a COM Interface or Server only be registered once. Automatic conversions may run afoul of this restriction and may require pruning in your AppXManifest file to remove duplications.

4.8.2 Packaged COM in MSIX

Support for AppXManifest registration has improved over time, and while you may specify all types in your package currently, there may be situations where support to implement the registration did not exist on older versions of the OS. In those cases, the installer does not fail or warn, but the functionality might not be present. When possible, we will identify such limitations through your testing process for any OS versions you will support.

The Windows 10 1703 (Creators Update) adds out-of-process (OOP) COM and OLE support for apps built using the Desktop Bridge (predecessor to MSIX) – aka Packaged COM. While the support level for COM registration in the AppXManifest file has improved since then, this blog by Microsoft is still the best introductory description of this support, which is also applicable for MSIX:

https://blogs.windows.com/windowsdeveloper/2017/04/13/com-server-ole-document-support-desktop-bridge/

Packaged COM (support scope is for out-of-proc COM servers) provides a mechanism for COM and OLE entries to be declared in the AppXManifest file using COM extensions so they can be used by external applications. The underlying system handles the activation of the objects so they can be consumed by

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

COM clients – all while still delivering on the Universal Windows Platform (UWP)/MSIX promise of having a no-impact install and uninstall behavior.

There are two categories of COM extensions which can be declared in the AppXManifest to represent the two parts of the COM object registration information:

- windows.comInterface
- windows.comServer

4.8.3 Assessing your current installer

The COM data within Windows Installer setup can be accessed by inspecting a set of COM specific tables. The data in these tables gets populated based on the registration method was chosen. Below are the different methods of registering the COM components in the windows installer (MSI) database:

- **Self-Registration**:
  This approach populates the MSI installer SelfReg table with the files marked for self-registration. Installation package authors are strongly advised against using self-registration.

  At run time, the installer calls the DLLRegisterServer and DLLUnregisterServer functions for the files populated in this table. Each function contains custom code that registers and unregisters the file. The developer of the file is responsible for writing the code to register and unregister the file in each of the functions.

  Note that **SelfReg** table registers only .DLL and .ocx files and also does not support choosing the order of registering the files. To overcome this, 3rd party tools (like InstallShield) use custom tables (like ISSelfReg table) to enable registering of EXE and tlb files in addition to what SelfReg table offers.

- **Extract/populate COM data to Windows Installer Tables**
  This method enables Windows Installer to take care of your application and the file's registration data is stored in the Windows Installer COM tables. When using this method, the file's DLLRegisterServer and DLLUnregisterServer functions are not called at run time, instead the Windows Installer takes the data specified in the COM tables and registers the file itself.

  The Windows Installer tables used to store the file's registration data are as follows:

  - The TypeLib table (http://msdn2.microsoft.com/en-us/library/aa372092.aspx) provides information that the installer places in the registry for the registration of type libraries. As Microsoft strongly advice not to use TypeLib table (due to rollback issues), many 3rd party tools populate this data to Registry table to register type libraries.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- The Class table (http://msdn2.microsoft.com/en-us/library/aa367861.aspx) is used to register Class IDs and other information for COM objects. This table contains COM server-related information that must be generated as a part of the product advertisement.
- The ProgId table (http://msdn2.microsoft.com/en-us/library/aa370879.aspx) associates program IDs with class IDs.
- The Registry table (https://docs.microsoft.com/en-in/windows/win32/msi/registry-table) contains interface and type library definitions that needs to be registered.

- **Registry-free COM**
  With Reg-Free COM approach, COM data is written to an application manifest file that is stored in the application folder. The manifest file is an XML file that contains information about an application and the libraries that are associated with it. The 3rd party tools like InstallShiled extract the COM data at design time and creates Win32-style manifest files that contains information about binding and activation of COM modules (by referring comClass and typelib tags).

By inspecting the above tables and other files, you can find out if your installer has COM components and this would be also helpful to validate the repackaged MSIX package has correct entries in the manifest.XML or registry.dat files.

One additional thing that you can try is to use the MSIX Packaging Tool and the other MSIX third party tools available in the market, like Flexera Admin Studio, to repackage your current installation into an MSIX Package. While this isn't the package you probably want to release, it will create most of the supportable COM Extension and Registry.Dat entries that you'd want in your package. However, a detailed explanation is given below on mapping traditional registry entries to COM extension nodes in AppXManifest that take place during the conversion process. But this might help in troubleshooting any COM issues that occur during your conversion process.

4.8.4 Integrating your COM into MSIX

In an ideal world, you would just add a new setup project for MSIX and it would automatically detect your COM components and do the right things for you. But if that doesn't work for you, this section might be of benefit.

Ultimately, you should have two kinds of COM extensions in your AppXManifest file: one of type "windows.comInterface" and another of category type "windows.comServer." There may also be additional entries for special types of COM (such as the Context Menus), but we will cover those elsewhere in this book.

NOTE: Examples in this section are based on traditional sample COM programs that may be found on this GitHub page:

https://github.com/flexera-public/MSIXDevGuide/tree/main/Chapter4_ProjectActivities/4.8_COMSupport

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The information below should be helpful to you in mapping the COM definitions in your IDL files to MSI registry entries and from MSI registry entries to MSIX manifest file.

The IDL files demonstrate the fundamental constructs of interface definitions for a COM component. Here is the sample IDL file:

```
// Typelib information
[ uuid(97E3C8AC-67BE-45E5-91EE-E563B68D6B12), version(1.0) ]
library MFCComServerBGColorTest
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	//  Primary dispatch interface and dual interfaces
	[ uuid(DCF859E1-5E2B-4E49-9C5C-9DED49E6C1DA) ]
	dispinterface IMFCComServerBGColorTest
	{
		...
	};

	[uuid(46EC5D13-1362-46FE-A2D3-EC5A39F64A17),
		oleautomation,
		dual
	]
	interface IDualMFCComServerBGColorTest : IDispatch
	{
		...
	};

	//  Class information for CMFCComServerBGColorTestDoc
	[ uuid(FA796308-1FB5-4289-B5DD-504D8499088E) ]
	coclass MFCComServerBGColorTest
	{
		dispinterface IMFCComServerBGColorTest;
		[default] interface IDualMFCComServerBGColorTest;
	};

};
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Below are the corresponding COM registry entries extracted from COM executable.

CLSID - Class information

CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E}
CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E}\InprocHandler32

Interfaces

Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}
Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\ProxyStubClsid32
Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\TypeLib

Interface\{DCF859E1-5E2B-4E49-9C5C-9DED49E6C1DA}
Interface\{DCF859E1-5E2B-4E49-9C5C-9DED49E6C1DA}\ProxyStubClsid32
Interface\{DCF859E1-5E2B-4E49-9C5C-9DED49E6C1DA}\TypeLib

TypeLib

TypeLib\{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}\1.0\0\win32
TypeLib\{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}\1.0\FLAGS
TypeLib\{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}\1.0\HELPDIR

Below are the COM specific MSI tables that stores the extracted information:

Class Table

CLSID	Context	Component_	ProgId_Default	Description	AppId_
{FA796308-1FB5-4289-B5DD-504D8499088E}	LocalServer32	MFCComServerBGColorTest.EXE	MFCComServerBGColorTest.Application	MFCComServerBGColorTest.Application	

ProgId Table

ProgId	ProgId_Parent	Class_	Description	Icon_	IconIndex _
MFCComServerBGColorTest.Application		{FA796308-1FB5-4289-B5DD-504D8499088E}	MFCComServerBGColorTest.Application		

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Note that the Registry table contains typelib specific data in most of the Windows Installer setups, as using TypeLib table is not recommended by Microsoft. Few of these entries are shown below:

Registry Table				
Registry	**Root**	**Key**	**Name**	**Value**
_C5C56ED5185DDF3F9E809A0AD900DC00	0	Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\TypeLib		{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}
_9028ACD8B899E19D66469D193F5F3495	0	Interface\{DCF859E1-5E2B-4E49-9C5C-9DED49E6C1DA}\ProxyStubClsid32		{00020420-0000-0000-C000-000000000046}
_A1497AB58C26EABFB0FEFC14010C412D	0	TypeLib\{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}\1.0\0\win32		[$MFCComServerBGColorTest.EXE]\MFCComServerBGColorTest.TLB
_1FE2EC28D7B965564ECBE0D0F182001F	0	Interface\{DCF859E1-5E2B-4E49-9C5C-9DED49E6C1DA}\TypeLib	Version	1.0

The traditional MSI setup takes care of registering these entries on the target system at the end of the installation.

The typical mappings required in the AppXManifest.XML file to expose the COM registrations to clients outside of the app package are explained below in the **windows.comInterface** and **windows.comServer** sections.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.8.4.1 windows.comInterface extension category

This category type of extension describes the interface GUIDs exposed by the COM component and ties it to a COM Server implementation. The com:Interface entry in the AppXManifest allows the AppInstaller to register new COM Interfaces. It represents typical registration of the traditional installer that are visible under HKCR \Interface\{MyInterfaceID}.

Microsoft schema documentation for this AppXManifest element may be found at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-com-cominterface

Originally, this extension category had to be placed under the Application/Extensions/ Extension AppXManifest element, but newer operating systems now also support placement under the Package/Extensions/Extension AppXManifest element. Both act the same way. If placed as an Application Extension, the interface is visible to any application inside or outside your package and is not tied explicitly to the application it is under. If you place the extension under Package/Extensions/Extension, you won't need to pick an Application to use for each interface, but there may be down-rev OS limitations.

Do keep in mind, however, that any registrations in comInterface that depend on another registration (e.g, an interface references a ProxyStub and/or a TypeLib) must be in the same comInterface extension.

Here is an example of a windows.comInterface registration for a known COM interface:

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
<Package …>
     …
     <Applications>
          …
     </Applications>
     <Extensions>
          <com:Extension Category="windows.comInterface">
               <com:ComInterface>
                         <!--  Interfaces -->
                         <!--  Interface\{46EC5D13-1362-46FE-A2D3-
                               EC5A39F64A17}\ProxyStubClsid32
                               Interface\{46EC5D13-1362-46FE-A2D3-
                               EC5A39F64A17}\TypeLib
                         →
                         <com:Interface ”d="46EC5D13-1362-46FE-A2D3-
                          EC5A39F64A17"
                             UseUniversalMarshaler="true">
                                <com:TypeLib Id="97E3C8AC-67BE-45E5-
                                 91EE-E563B68D6B12"
                                 VersionNumber="1.0" />
                         </com:Interface>
                         <!-- TypeLib  -->
                         <!--  TypeLib\{97E3C8AC-67BE-45E5-91EE-
                              E563B68D6B12}\... -->
                         <com:TypeLib Id="97E3C8AC-67BE-45E5-91EE-
                          E563B68D6B12">
                               <com:Version  DisplayName =
                                 "MFCComServerBGColorTest"
                                 VersionNumber="1.0"
                                 LocaleId=”0”
                                 LibraryFlag="0">
                                      <com:Win32Path
                                       Path="MFCComServerBGColorTest.
                                       tlb" />
                               </com:Version>
                         </com:TypeLib>
               </com:ComInterface>
          </com:Extension>
     </Extensions>
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The AppXManifest entries above correspond to the typical registration entries present under both the HKCR \Interface\{MyInterfaceID} and HKCR\Typelib\{MyTypelibID} keys. The windows.comInterface extension supports Interface, ProxyStubs, and Typelibs in order to map existing registry entries present to those keys. As discussed previously, the extension can be placed under either the Application/Extensions/Extension manifest element, or under the Package/Extensions/Extension manifest element.

The comInterface extension may include the following three types of registrations, based on entries found during the conversion process:

- Interface
- Type Lib
- Proxy Stub

4.8.4.1.1 com:Interface

The com:Interface entry registers new COM Interfaces. It represents typical registration entries under HKCR \Interface\{MyInterfaceID} in traditional installer.

The com:Interface element is documented at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-com-interface

Below is an example showing typical mapping between traditional HKEY_CLASSES_ROOT\Interface\{00EF48D3-321B-4A12-90A1-3DA7D6D8D9E8} registry entries to corresponding AppXManifest com:interface entry:

Exported entries from registry:

```
[HKEY_CLASSES_ROOT\Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}]
@="IDualMFCComServerBGColorTest"

<!-- ProxyStubClsid32 value {00020424-0000-0000-C000-000000000046} indicates standard marshaling.-->
[HKEY_CLASSES_ROOT\Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\ProxyStubClsid32]
@="{00020424-0000-0000-C000-000000000046}"

[HKEY_CLASSES_ROOT\Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\TypeLib]
@="{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}"
"Version"="1.0"
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The equivalent com:Interface entry in the manifest.XML file:

```
<com:Interface Id="46EC5D13-1362-46FE-A2D3-EC5A39F64A17"
UseUniversalMarshaler="true">
  <com:TypeLib Id="97E3C8AC-67BE-45E5-91EE-E563B68D6B12"
  VersionNumber=”1.0" />
</com:Interface>
```

Few points to note:

- **UseUniversalMarshaler** value in AppXManifest is set to true if HKCR\Interfaces\{iid}\ProxyStub Clsid32 value is pointing to standard marshal {00020424-0000-0000-C000-000000000046}.
- AppXManifest **entry UseUniversalMarshaler**=true represents the use of OLE Universal Marshaler (typelib oleautomation) marshalling as the proxy stub by COM runtime.
- If **UseUniversalMarshaler** is true, the Interface registration must include **TypeLib element.**
- ProxyStubClsid and UseUniversalMarshaler are mutually exclusive.
- If the custom proxy/stub needs to be used (as generated by the MIDL compiler from IDL definition file), then the corresponding interface registry entry needs to be changed to the CLSID of that proxy object instead.

4.8.4.1.2 com:Typelib

The com:TypeLib represents type library for an interface. A type library contains a binary description of an interface exposed by a COM-component. In this sense, they contain the same information that is contained in an IDL (Interface Definition Language) file of win32 application.

The com:TypeLib element is documented at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-com-typelib

Registering Type library for a COM component in MSI installer would create two entries in the registry, one under HKEY_CLASSES_ROOT\Interface and the other one refers to HKEY_CLASSES_ROOT\TypeLib.

Below are the TypeLib entries exported from windows registry or extracted from the MSI Registry table:

```
<!-- Referring TypeLib from HKEY_CLASSES_ROOT\Interface -->
[HKEY_CLASSES_ROOT\Interface\{46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\TypeLib]
@="{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}"
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
"Version"="1.0"

<!-- TypeLib definition HKEY_CLASSES_ROOT\TypeLib -->
[HKEY_CLASSES_ROOT\TypeLib\{97E3C8AC-67BE-45E5-91EE-E563B68D6B12}\1.0\0\win32]
@="C:\\Projects\\MFCComServerBGColorTest.TLB"
```

com:Interface entry in the manifest.XML file:

```
<com:TypeLib Id="97E3C8AC-67BE-45E5-91EE-E563B68D6B12">
     <com:Version DisplayName="MFCComServerBGColorTest"
              VersionNumber="1.0"
              LocaleId="0"
              LibraryFlag="0">
          <com:Win32Path Path="MFCComServerBGColorTest.tlb" />
      </com:Version>
</com:TypeLib>
```

4.8.4.1.3 com:ProxyStub

The com:ProxyStub element registers a proxy stub. This element will not be present if com:Interface has the attribute **UseUniversalMarshaler**=true.

The com:ProxyStub element is documented at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-com-proxystub

The extension supports both standard/universal marshalling and custom marshalling.

Standard marshaling or Universal Marshaler

Standard marshalling will be considered for the interfaces with [dual] or the [oleautomation] attributes tagged to them in idl files.

Corresponding traditional registration entry for COM interface will be as below:

```
[HKEY_CLASSES_ROOT\Interface\{ 46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\ProxyStubClsid32]
@="{00020424-0000-0000-C000-000000000046}"
```

From the above registry entry,

The value pointed by ProxyStubClsid32 is {00020424-0000-0000-C000-000000000046}. This is the CLSID of the standard and universal marshaler.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

There is no separate Proxy Stub entry created in the AppXManifest for this scenario as flag UseUniversalMarshaler=”true” indicates the universal\standard marshalling will be used as proxy stub.

Custom marshalling

Dedicated ProxyStub entries in the AppXManifest are needed to register your custom marshalling interfaces if the interfaces defined in the .idl file uses:

- Not a standard (Non OLE-automation-compatible) compatible types referred as parameters
- Not marked with the [dual] or [oleautomation] attributes.

[HKEY_CLASSES_ROOT\Interface\{ 46EC5D13-1362-46FE-A2D3-EC5A39F64A17}\ProxyStubClsid32] entries mapped to corresponding com:ProxyStub entries in the AppXManifest.

4.8.4.2 windows.comServer extension category

This category type of extension describes the CLSID GUIDs exposed by the COM component consisting of the COM Server implementation.

It is important to place the COM server registration under the correct Applications/Application AppXManifest element, because COM server processes will run with the identity of the ancestor Applications/Application element.

The entries used in the AppXManifest file for this element correspond to the typical registration entries found under the CLSID key of the Windows Registry (i.e. HKEY_CLASSES_ROOT\CLSID\{MyClsId}) for an application supporting executable servers and their COM classes (including their OLE registration entries), surrogate servers, ProgIDs and TreatAs classes.

Below is an example that shows a typical windows.comserver extension in the AppXManifest:

```
<com:ComServer>
      <com:ExeServer Executable="MFCComServerBGColorTest.exe"
                     DisplayName="MFCComServerBGColorTest">
            <!-- HKCU\CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E} =
                   MFCComServerBGColorTest Application
                 HKCU\CLSID\{FA796308-1FB5-4289-B5DD-
                   504D8499088E}\LocalServer32
                   = MFCComServerBGColorTest.exe
                 HKCU\CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E}\ProgId
                   = MFCComServerBGColorTest.Application -->
            <com:Class Id ="FA796308-1FB5-4289-B5DD-504D8499088E"
                  DisplayName="MFCComServerBGColorTest Application"
                  ProgId="MFCComServerBGColorTest.Application.1"
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
                VersionIndependentProgId="MFCComServerBGColorTest
                .Application">
            </com:Class>
        </com:ExeServer>
        <!-- ProgId -->
        <!--  HKEY_CLASSES_ROOT\MFCComServerBGColorTest.Application =
                "MFCComServerBGColorTest.Application"
              HKEY_CLASSES_ROOT\MFCComServerBGColorTest.Application\CLSID =
                {FA796308-1FB5-4289-B5DD-504D8499088E} -->
        <com:ProgId Id="MFCComServerBGColorTest.Application"
                CurrentVersion="MFCComServerBGColorTest.Application
                .1" />
        <com:ProgId Id="MFCComServerBGColorTest.Application.1"
                Clsid="FA796308-1FB5-4289-B5DD-504D8499088E" />
</com:ComServer>
```

The **windows.comServer** extension may include the following four types of registrations based on entries found during conversion process:

- EXE Server
- ProgId
- Surrogate Server
- TreatAsClass

4.8.5.2.1 com:ExeServer

The com:ExeServer Registers an ExeServer with one or many class registrations. The ExeServer registrations correspond to LocalServer32 keys and their associated AppID key. The Executable and Arguments attributes refers to the default value of the traditional LocalServer32 registry key. In Win32, LocalServer32 registry key specifies the location of the COM server application to launch.

The com:ExeServer element is documented at this link:

https://docs.microsoft.com/en-in/uwp/schemas/appxpackage/uapmanifestschema/element-com-exeserver

Below are examples exported LocalServer32 entries from the Windows registry of a standard installed component:

```
<!-- Referring TypeLib from HKEY_CLASSES_ROOT\Interface -->
[HKEY_CLASSES_ROOT\CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E}]
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
@="MFCComServerBGColorTest.Application"

[HKEY_CLASSES_ROOT\CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E}\LocalServer32]
@="C:\\ Projects\\MFCComServerBGColorTest.exe"

[HKEY_CLASSES_ROOT\CLSID\{FA796308-1FB5-4289-B5DD-504D8499088E}\ProgID]
@="MFCComServerBGColorTest.Application"
```

The equivalent com: ExeServer entry in the ApXManifest.XML is shown here:

```
<com:ExeServer Executable="MFCComServerBGColorTest.exe”
               DisplayName="MFCComServerBGColorTest">
       <com:Class Id ="FA796308-1FB5-4289-B5DD-504D8499088E"
                  DisplayName="MFCComServerBGColorTest Application"
                  ProgId="MFCComServerBGColorTest.Application.1"
                  VersionIndependentProgId="MFCComServerBGColorTest.Applica
                  tion">
       </com:Class>
</com:ExeServer>
```

4.8.4.2.2 com:ProgId

A programmatic identifier (ProgID) that can be associated with a CLSID. The ProgID identifies a class but with less precision than a CLSID because it is not guaranteed to be globally unique. The ProgID can be used in programming situations where it is not possible to use a CLSID.

The com:ProgId element is documented at this link:

https://docs.microsoft.com/en-in/uwp/schemas/appxpackage/uapmanifestschema/element-com-progid

Below are examples exported LocalServer32 entries from the registry of a standard installed component:

```
<!--  Associating ProgId with CLS ID -->
[HKEY_CLASSES_ROOT\MFCComServerBGColorTest.Application]
@="MFCComServerBGColorTest.Application"

[HKEY_CLASSES_ROOT\MFCComServerBGColorTest.Application\CLSID]
@="{FA796308-1FB5-4289-B5DD-504D8499088E}"
```

A ProgID has the form *name1.name2.version*. If registry entry does not contain a *version* for a ProgID,

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

then the default version is 1. Below is the equivalent entry for the com:ProgId in the manifest.XML file:

```
<com:ProgId Id="MFCComServerBGColorTest.Application"
            CurrentVersion="MFCComServerBGColorTest.Application.1" />
<com:ProgId Id="MFCComServerBGColorTest.Application.1"
            Clsid="FA796308-1FB5-4289-B5DD-504D8499088E" />
```

4.8.4.2.3 com:SurrogateServer

The com:SurrogateServer element registers an SurrogateServer with one or many class registrations. COM makes it possible to create COM servers in DLL form that can be loaded into a surrogate EXE process. Running the DLL server in a surrogate process offers several possible benefits like fault isolation and the ability to service multiple clients simultaneously etc. The COM Surrogate process runs a COM object outside the original process that requested it. If the COM object crashes, it will only take down the COM Surrogate process and the original host process will not be impacted.

The COM framework provides a default surrogate process (DLLhost.EXE), or one can write a custom surrogate if there are special needs. The system-provided surrogate will work very well for most developers, as well as being very easy to use. However, those developers with special considerations may decide that a custom surrogate is necessary.

The com:SurrogateServer element is documented at this link:

https://docs.microsoft.com/hi-in/uwp/schemas/appxpackage/uapmanifestschema/element-com-surrogateserver

The com:SurrogateServer attributes 'CustomSurrogateExecutable' and 'SystemSurrogate' helps in defining the system provided and custom surrogates. If the registry entry value for the DLLSurrogate is NULL or an empty string, choose the system-supplied surrogate for the com:SurrogateServer in the manifest file. The child element com:Class in SurrogateServer helps in defining a SurrogateServer class registration.

For example, A **desktop4:FileExplorerContextMenus** category application extension registers the context menu handler implementation, by defining a COM surrogate server with the class ID of the context menu handler implementation. The COM surrogate process takes care of loading the DLL that has the implementation for the IExplorerCommand or IExplorerCommandState interfaces.

```
<desktop4:Extension Category="windows.fileExplorerContextMenus">
      <desktop4:FileExplorerContextMenus>
            <desktop4:ItemType Type=".contoso">
                  <desktop4:Verb Id="Command1"
                     Clsid="d0c8bceb-28eb-49ae-bc68-454ae84d6264" />
            </desktop4:ItemType>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
      </desktop4:FileExplorerContextMenus>
</desktop4:Extension>

<com:Extension Category="windows.comServer">
      <com:ComServer>
            <com:SurrogateServer AppId="d0c8bceb-28eb-49ae-bc68-
                                 454ae84d6264"
                                 DisplayName="ContosoHandler">
                  <com:Class Id="d0c8bceb-28eb-49ae-bc68-454ae84d6264"
                             Path="ExplorerCommandVerb.DLL"
                             ThreadingModel="STA"/>
            </com:SurrogateServer>
            </com:ComServer>
</com:Extension>
```

Please refer to 7.12.6.5 Example context menu handler extension in this book for more details.

4.8.4.2.4 com:TreatAsClass

A registration that corresponds to a CLSID registration with the TreatAs subkey. The com:TreatAsClass element provides the ability for one server associated with one CLSID to emulate a server of another CLSID. A server that can emulate another is responsible for providing compatible behavior for a different class through a different implementation. This forms the basis for allowing end-users the choice in which servers are used for which objects, as long as the behavior is compatible between those servers.
The traditional 'TreatAs' registry entry under HKEY_CLASSES_ROOT\CLSID class node specifies the CLSID of another class that can emulate the current class.

The com:TreatAsClass attributes Id and 'TreatAs' helps in specifying CLSID's of old and new COM class objects in appmanifest.XML file.

The com:TreatAsClass element is documented in this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-com-treatasclass

The example below shows how 'TreatAs' registry entry helps in handling emulation between CLSID objects 'Microsoft Excel 95 Worksheet' and 'Microsoft Excel 97-2003 Worksheet':

```
<!-- CLS ID Microsoft Excel 95 Worksheet  -->
[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{00020810-0000-0000-C000-000000000046}]
@="Microsoft Excel 95 Worksheet"

<!-- TreatAs attribute pointing to New CLS ID Microsoft Excel 97-2003 Worksheet  -->
[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{00020810-0000-0000-C000-
000000000046}\TreatAs]
@="{00020820-0000-0000-C000-000000000046}"
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

After this entry is set, when CoGetClassObject is called with the object's original CLSID the (00020810-0000-0000-C000-000000000046), it is transparently forwarded to the new CLSID (00020820-0000-0000-C000-000000000046), thus launching the application associated with the **TreatAs** CLSID 'Microsoft Excel 97-2003 Worksheet'.

```
<!-- CLSID entry that handles the
[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{00020820-0000-0000-C000-000000000046}]
@="Microsoft Excel 97-2003 Worksheet"

[HKEY_CLASSES_ROOT\WOW6432Node\CLSID\{00020820-0000-0000-C000-
000000000046}\LocalServer32]
@="C:\\Program Files (x86)\\Microsoft Office\\Root\\Office16\\EXCEL.EXE"
```

And here is the equivalent entry for the com:TreatAsClass in the manifest.XML file:

```
<com:TreatAsClass Id="00020810-0000-0000-C000-000000000046"
                  DisplayName="Microsoft Excel 95 Worksheet"
                  TreatAs="00020820-0000-0000-C000-000000000046"/>
```

4.8.5 Additional COM limitations

- As per Microsoft documentation, COM extensions are not currently supported on the Windows Server 2019 operating system at this time. For the latest updates, please refer to https://docs.microsoft.com/en-us/windows/msix/msix-server-2019. However, COM extensions seems to be working fine on Windows Server 2022 Beta version.
- Client applications that would like to take advantage of COM extensions should use COM API calls. Applications that directly read the registry using registry APIs like RegOpenKeyEx() or RegEnumKeyEx()) will not see any COM extension entries (unless these are manually added to the application hive) and will fail.

4.9 Services

Windows Services are supported as part of your MSIX package, but in a distinct way that warrants discussion.

- MSIX packages install on a per-user basis, which allows for package installation without the need for elevation (UAC Prompt). This is not the case with packages that contain Windows Services.
- Services are associated exclusively with the Windows operating system. Including a service registration in the AppXManifest requires use of the Desktop schemas, which will limit the package to those operating systems. Services require a Windows Client with a minimum version of 2004 (20H1).

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- Additionally, there is a Capabilities designation that is required in the AppXManifest. Packaged services may run only under the Local Service, Network Service, or Local System accounts. Any package with a service must declare the Packaged Services capability. In addition, if the service utilizes the Local System account, the Local System Services capability must also be declared. See 7.7 Capabilities element for more details on Capabilities registration in the AppXManifest.
- The service is natively installed into the OS, making it work like the service you are used to in traditional setup programs, even if you specify the service differently as part of the AppXManifest.
- The service does not run inside the container and might not have access to the package application hive. Services that require package components could have issues, but our experiences to date have been that most services are fine due to how they work.
- Although service dependencies are supported, if the dependent service is not part of the standard OS or your package, in addition to registering the dependency with the service in the AppXManifest, it may be necessary to declare a package dependency to ensure that the dependency is met.

Windows Services have traditionally been used for the following reasons:

- To have a component that runs in the background whether the application is running or not. To support multi-user systems where this component needs to be separated by user, more modern versions of background tasks might be considered as the capabilities of Windows App SDK [see 2.3 Windows App SDK (aka Project Reunion)] firm up.
- To have a component running under the system context. This might be to perform acts on behalf of the end-user who does not have administrative privileges.
- To update the application. These services, and the existing update process, should be removed from the software. MSIX packages cannot be updated this way. MSIX does have an auto/update process that is optional and outside the scope of the package but triggered as part of how the package is installed. This makes it easy for you to have a single auto update process that works whether you leverage the Microsoft Store for distribution or host downloads on your own site.
- To act as a conduit for receiving data over the internet.

4.9.1 Assessing your current installer

Locate existing Windows services in your current installer package.

You should also review the following to ensure that your service registration will be successful under MSIX:

- **Service Name** is limited to 256 characters. The Service Name cannot start or end with white space. Most services should be OK, but you may need to make a minor change in name if not.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT. revenera

- **Startup Type** supports types of disabled, manual, and auto only. If your service uses autostart-delayed today, you will want to change that to auto.
- **Startup Account** is limited to localSystem, localService, and networkService. If your service uses any other account, you will need to determine if you can change it and what impact this might have on operations.

One additional question to answer is if you need the service. If the purpose of the service is to support application updates, MSIX offers additional options that you should consider. This can remove the logic for updates out of your software and simplify your package. Not only is Microsoft Store distribution an option, but the AppInstaller file (see 4.16 Package Updates and AppInstaller File) should be considered for non-store deployments.

4.9.2 Integrating into your new package

You will need to remove any current service installation logic in your installer or application and migrate to the AppXManifest declaration method of getting your service installed. An example of this is given in 7.12.6.2 Example Windows service extension.

Because the services may not have access to the registry hive, additional care should be taken to fully test your MSIX package if it contains services.

Microsoft has documentation aimed at helping you migrate your service at this link:

https://docs.microsoft.com/en-us/windows/msix/packaging-tool/convert-an-installer-with-services

NOTE: Many developers ask why they can't use a script to install the service by using sc.EXE, more similarly to what the MSI installer does.

- The effect of how the service operates today is the same no matter how it is installed. This could change in the future making package declared services more capable.
- There is no capability of your installer running the script, but such scripts must be done at runtime when the user first runs the program (or an outside wrapper such as a PowerShell script is used to install your package and the service). Performing the service installation at user runtime is problematic as elevation is required to install the service via script and often enterprise end-users do not have permissions needed.
- If you install the service, you are also responsible to uninstall it. Again, the package uninstall would not be aware of the service. It is best to leave it to Microsoft to manage the service install/uninstall for you by making the proper references in the manifest.

4.10 Firewall configuration

In both traditional and MSIX applications, an application that needs to communicate with the outside world through a port will be required to add a Windows Firewall Exception with all the required details,

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

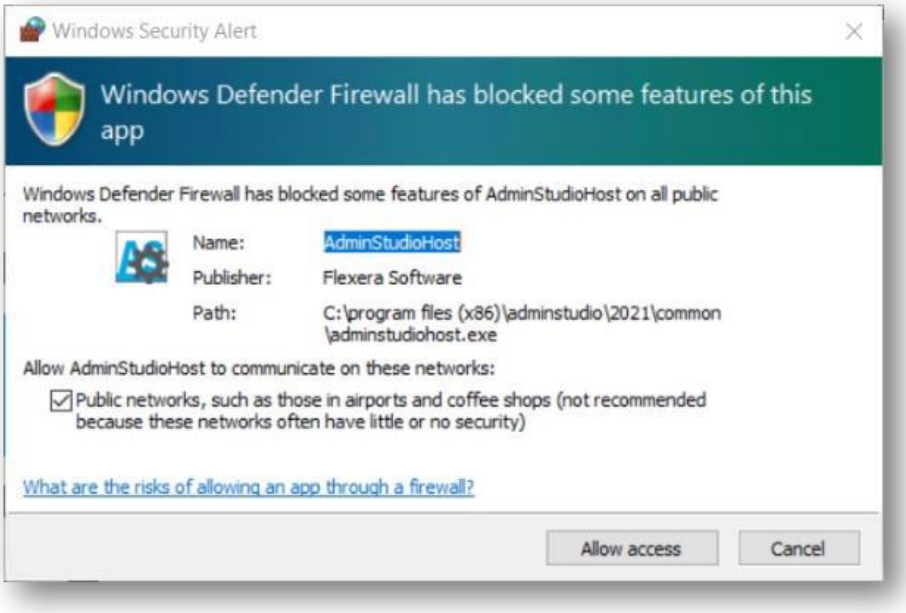
Flexera
Inform IT. Transform IT.
revenera

so the operating system does not block the communication. This will ensure the application functions with no unexpected behavior. As you might guess, this is to be done differently for MSIX.

4.10.1 Assessing your current installer

If the current version of the application installer makes use of additional ports for communication, it is likely that you can discover this in your current application installer. In a traditional MSI installer, there is no native support for adding Windows Firewall Exceptions for a program during installation, so the most common way of adding a firewall exception in an MSI setup today is by creating a custom action using netsh.EXE. There are also a few setup authoring tools that provide out-of-the-box support for adding firewall exceptions to an MSI setup, which is easier than creating your own custom action.

When a program in your application needs to communicate and if there is no firewall exception added to the machine during the installation, the end-user will be prompted with an alert like the one below from the operating system at application runtime, when such a program is triggered by your application:



This should always be avoided, as the end-user may not be aware of what selection to make on this prompt. If an incorrect selection is made, your application will not function as desired. So it becomes prudent to register firewall exceptions to be added silently during the installation itself so no such prompts are thrown at the user while launching or using the application.

4.10.2 Integrating into your new package

Microsoft has made it a lot simpler to add firewall exceptions to an MSIX package by utilizing a special Package Extensions specifically for adding firewall exception rules to an MSIX package: windows. firewallRules. If your application requires a firewall exception, you have to add a windows.firewallRules package extension with all the required attributes to the AppxManifest.XML file of your application.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Example:

Google Chrome adds a firewall exception for its program: chrome.EXE. This exception is silently added during the installation of its traditional MSI installer. Although Google doesn't have an MSIX package released today, here is an example of how the same firewall exception could be added to Google Chrome's MSIX package during its migration from MSI to MSIX format:

```
<Package …>
     …
     <Applications>
          …
     </Applications>
     <Extensions>
          <desktop2:Extension Category="windows.FirewallRules"
           Executable="Chrome\Application\chrome.exe">>
               <desktop2:FirewallRules
                Executable="Chrome\Application\chrome.exe">
                    <desktop2:Rule Profile="all" LocalPortMin="5353"
                     LocalPortMax="5353"
                        IPProtocol="UDP" Direction="in"/>
               </desktop2:FirewallRules>
          </desktop2:Extension>
     </Extensions>
</Package>
```

Although the Microsoft documentation may not be clear, the Executable field on the desktop2:extension is required, and like all file paths in the AppXManifest package is the relative path to the EXE. This extension is placed as a Package-level extension (not application), and although current Microsoft documentation refers to defaulting, just put the relative path to the EXE that needs it in this parameter.

4.10.3 Microsoft Firewall References

For details on the syntax and attributes for the windows.firewallRules extension, please refer to: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-desktop2-firewallrules

And for the specifics of the rules: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-desktop2-rule

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.11 Fonts

Fonts are supported to install and register as part of the MSIX package so that the fonts are available to other applications installed for the user and potentially other users on the OS.

4.11.1 Assessing your current installer

Here is what you might have in your traditional Windows installer to install the Fonts:

Font files included in your existing package. These files are recognizable by file type extension. Here is a list of the most common extensions:

- .ttf and .ttc
- .fon
- .fnt
- .otf
- .woff and .woff2

Numerous proprietary and other attempts at open standards also exist with different file type extensions. You may need to look at the registration activity of your installer to detect them.

- Windows installers copy the fonts to a directory, normally the %windir%\fonts directory, and register the fonts in the target system based on the information available in the MSI Font table by the RegisterFonts custom action.
- You should be on the lookout for any named font files that contain unusual characters as part of the file name, such as certain Google font files that contain “%20” in the name. These will require special attention.

4.11.2 Integrating into your new package

For the most part, you will still include the files somewhere in your package and register them by modifying the AppXManifest. There should not be the need to modify any code unless your code took the unusual step of accessing fonts by path (very rare).

Here is what you do to install Font as part of the MSIX package:

- Add the font files to the package file ingredients in whatever folder you like. It might be necessary to rename some of the font files. Some font file names, such as those with special characters might not be supported by the AppXManifest schema, and if you attempt to validate the schema file or build the app, the error cause will be clear. In this case, simplifying the file name as stored in your package is all that is needed, and the app will be able to access the font using the font name kept internal to the font file.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- These fonts must be declared for registration in the AppXManifest file as an extension of the category “windows.sharedFonts” under an Application element. It is not important what application element of the file is used.
- The path you specify for a font file in the windows.sharedFonts extension should be the path relative to your package. Multiple font files should be included in a single windows.Sharedfonts extension.

Here is an example of how it might look in your AppXManifest:

```
<Package…>
     …
     <Applications>
           <Application …>
                 <VisualElements …>
                       …
                 </VisualElements>
                 <Extensions>
                       <uap4:Extension Category=”windows.sharedFonts”>
                             <uap4:SharedFonts>
                                   <uap4:Font File=”Fonts\abcde.fon”/>
                                   <uap4:Font File=”Fonts\fghij.ttf”/>
                             </uap4:SharedFonts>
                       </uap4:Extension>
                 </Extensions>
           </Application>
     </Applications>
</Package>
```

4.11.3 Runtime differences

For the most part, the fonts will work as they did before, but there are some small differences that might affect the end-user experience:

- MSIX packages install on a per-user basis. Hence the fonts are also installed and registered per-user and are available to the applications installed for the user through the Application extension. Your existing installer may have registered the font globally, making the fonts available to additional users on a multi-user operating system. Under MSIX, only those users with your application installed will have the new fonts available.
- After MSIX installation, the fonts file is not stored in the %windir%\fonts directory. That's because the MSIX package will not copy the file to the %windir%\fonts directory during installation. Instead,

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

it will be registered as available (even to programs outside the container) using the installed file paths for your package.

4.11.4 Font references

There are two choices for registering fonts in the package: one using uap4 schema and also the uap7 schema.

The uap4 version registers as an Application-level extension, although it still makes the fonts available to the entire package. The uap7 schema version registers at the package level to help make this clearer, but it does not work on older OS versions and requires special permission for store submission, so you should just stick with the uap4 version for now.

The Schema references for these in Microsoft documentation are:

- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap4-sharedfonts
- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap7-sharedfonts

Both elements use uap4:Font child elements from this schema:

- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap4-font

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.12 Environment variables

In your existing application installer, Environment Variables might be used. If so, you will need to investigate their purpose and what needs to be done. The MSIX installation does not add or alter any environment variables when installed, nor are any available to the running application without a fixup (such as one from the Package Support Framework).

Some uses of environment variables are new variables for the application, while others are for helping system APIs while the app is running or for external purposes.

- **New variables used only by the application.** By default, MSIX packages have no environment variables defined in the package, and your installer cannot add them to the system on installation. If you need them, you will likely want to use the EnvVarFixup of the Package Support Framework for these variables. Whether your existing environment variables were system or user variables, this will turn your environment variables into "container" environment variables not seen outside the container.

  If the original variable was a System variable, the end-user with standard writes would not have been able to change the value, so you probably want the definition in the EnvVarFixup config for this variable to have the useRegistry value set to false. Setting that value to true allows the end-user to change the value at runtime (assuming that your app has this capability).

  More information on this fixup may be found at this link: https://github.com/microsoft/MSIX-PackageSupportFramework/blob/develop/fixups/EnvVarFixup/readme.md

  The alternative is to create the variables yourself on the first run of the package. This can be done in your code or by using the Package Support Framework PsfLauncher scripting. But once you bring in the PSF, you should probably use the EnvVarFixup for this purpose.

- **Variables** to help your app when calling certain Windows APIs. This covers the situation where your existing installer modifies the Path variable to add additional directories to the DLL loading searches. Collapsing your installation into a single directory might be a solution and using the Package Support Framework DynLibraryFixup. This fixup allows you to name DLLs in your package and ensure they are found inside the package when requested.

  More information on using DynLibraryFixup may be found at this link: https://github.com/microsoft/MSIX-PackageSupportFramework/tree/develop/fixups/DynamicLibraryFixup

- **Variables to help things outside the package.** Another purpose for changing the Path variable is to help the user (or other apps) find your executable by name alone. In MSIX, this is handled by adding an AppExecutionAlias application extension for the EXE. More information on using AppExecutionAlias may be found at this link: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap5-appexecutionalias

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

4.13 Event logging

Logging of events is sometimes necessary in applications, whether for your own diagnostic purposes or because your software needs to feed into management software that looks to the event logs for events.

4.13.1 Assessing your current installer/program

Your traditional application might use one of the following approaches, not that this is a complete list:

- Dumping messages out to the debug console port.
- Writing out to a private log file.
- Writing events to the “classical” Windows Logs, typically Applications or System.
- Writing events to the Applications and Service Logs, probably with your own private log.

4.13.2 Options for logging in your new package

After you identify the type of logging that is currently used, the information in this section will clarify what migration efforts are required. In addition to simple migration, consideration might be given to moving to the latest “current best practice” for event logging.

The migration steps for these four existing types follow.

4.13.2.1 Using the Debug Console Port

Your existing code does not require any modification. Techniques used in the past to view this output remain unchanged.

4.13.2.2 Writing out to a private log file

Generally, your existing code is fine here as well. Consideration should be given as to where this log is written, and you may need to adjust the location since package files may not be overwritten by default. Writing these logs to files outside the package is recommended.

4.13.2.3 Writing events to the “classical” Windows logs

Your existing code to send events should not require any modification if you keep sending text strings that don’t need an event formatter.

4.13.2.4 Writing events to the application and services logs

The issue to be dealt with is how to register the new log and the provider that is used for the event formatter.

In your current software, this might be accomplished by using an installer action or runtime (initialization) code to register your ETW provider utilizing the wevtutil.EXE system application. As that

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

has system-wide implications, the registration of the provider requires elevation. Your existing installer was already elevated, so you never had to worry about that.

Without this registration in place, your code writing the events will not notice any problem, but the events will be silently dropped by the event thread of the kernel since nothing is registered to receive the events. If a previous provider was registered and the application was removed without removing this registration, your current code will now write entries into the existing log, but the formatter will be missing.

MSIX installs do not have any kind of external actions available similar to the MSI Custom Action you probably use today. MSIX AppXManifest-based elements to register your provider and log have been defined, but they are not yet implemented and are ignored if present in the AppXManifest. It is unclear as to when that will be resolved in future operating systems.

Because of this, moving to a runtime code-based registration seems warranted. To implement the installation of a formatter, an application code runtime-based registration will be needed. Because the installation of the provider and log requires elevation, it is unclear what the best practice should be.

You probably would want to create a new executable for this, probably triggered by your primary application detecting that the correct formatter is not already installed. If your app already elevates, adding all this into the startup logic of the app directly can suffice, but we would recommend that you extract the needed code into a separate executable with an internal app.manifest file containing:

```
<requestedExecutionLevel level=”requireAdministrator” uiAccess=”false”/>
```

The reasons for this to be moved to a separate executable from your main software and options to consider include:

- If your exposed entry points have this manifest, you will be required to add the RequireElevation Capability to your manifest, and they are unlikely to pass through the Windows Store Certification.
- It is a bad practice to run the entire app elevated if you only need this.
- If your application includes a Windows service, it could be appropriate to move the manifest installation to the startup code in your service (as it likely already runs under an account with needed permissions).
- It would be a questionable practice to create a service only for this purpose.
- It might be possible to create an out-of-process com-based EXE and have your initialization call trigger it to perform the installation. These exes run as a child of the TaskHost service and should have needed rights.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- It can be a separate EXE triggered to run once after detection. The end-user will only be prompted for admin credentials once. The trigger may be in your main application startup code through the use of PsfLauncher scripting or Custom Install Activities.

The bottom line is that we don't have a great solution for all projects, but hopefully one of these will meet your needs and technical background. You should also look for any updated guidance from Microsoft, as they may attempt to fill this gap.

4.13.3 Option: Initialization Code Check

Your initialization code should check to see if registration is required and then only call to install if needed.

This initialization check needs to determine if the handler is already installed and if you are now running the version of your application that is installed to a different folder. This would occur when your package version number changes or when the certificate to sign the package changes.

Running the PowerShell command "Get-WinEvent" with parameter–ListProvider and argument of your provider name will return a provider object (System.Diagnostics.Eventing.Reader.ProviderMetadata) only if currently installed. You can then look at the ResourceFilePath property of that object for a string containing the path to the current resource file.

- If the provider object is not returned, the event provider has not been installed and you simply need to make the call to install it.
- If the provider object is returned but the path contained in the ResourceFilePath is now incorrect, you'll want to make the call to uninstall the current one and then install the new one.
- If the provider object is returned but the paths match, you are done.
- The following C# code could be used for this installation check without elevation:

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

```
using System.Management.Automation;
using System.Management.Automation.Runspaces;
   /// <summary>
   ///  Checks to see if the event log and provider are registered correctly
   /// </summary>
   /// <param name="ProviderName">Name of the provider in the ETW Manifest,
        typically
   /// in "Vendor-Product" form. </param>
   /// <param name="PathToNewResourceFileName">File path to where
/// the dll file containing the formatter resource is now installed. </param>
   /// <returns>Null, if no provider is registered.  False if registered
        incorrectly,
   /// True otherwise. </returns>
   private bool? IsCorrectEventLogRegistered(string ProviderName,
                                  string PathToNewResourceFileName)
   {
    try
    {
     RunspaceConfiguration config = RunspaceConfiguration.Create();
     Runspace myRs =  RunspaceFactory.CreateRunspace(config);
     myRs.Open();
     Pipeline pipeline = myRs.CreatePipeline();
     Command cmd1 = new Command("Get-WinEvent");
     cmd1.Parameters.Add("ListProvider", ProviderName);
     pipeline.Commands.Add(cmd1);
     System.Collections.ObjectModel.Collection<PSObject> results =
     pipeline.Invoke();
     if (pipeline.HadErrors)
      return false;
     foreach (PSObject result in results)
     {
      //System.Diagnostics.Eventing.Reader.ProviderMetadata
      foreach (PSPropertyInfo psPropertyInfo in
      result.Properties)
      {
       if (psPropertyInfo.Name.Equals("ResourceFilePath"))
       {
        string ExtestingResourceFileName = (string)psPropertyInfo.Value;
        if (PathToNewResourceFileName.Equals(ExtestingResourceFileName))
          return true;
        else
          return false;
       }
      }
     }
    }
    catch
    {
     ;  // Log or take appropriate action here
    }
    return null;
   }
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Because PowerShell can take a while, this initialization check should be performed in a background thread so as not to slow down the startup of the user's application each time.

4.13.4 Option: Provider and Log Registration or Update

These activities require elevation to use the wevtutil system utility for registration activities. Assuming your application does not already elevate, this will result in a UAC prompt for the end-user the first time the application is run.

NOTE: Packages including components that elevate should mark the capabilities requirement "requiresElevation" in the package AppXManifest file. To release through the Windows Store the capability marking is required, and you would need to pass through a manual exception process for authorization by Microsoft. It is unclear what Microsoft would do, but if you explain the purpose and how you have limited the usage, they might find it compelling. In packages delivered via sideloading this was originally required also, but Microsoft removed that restriction in OS 1903; however, we still consider it a best practice to mark the capability in your AppXManifest file.

You also want to consider the situation for enterprise deployment where the end-user does not have elevation rights and logging can never succeed. Your code should be robust enough to survive the user canceling the UAC request and not bother them again.

Removal of the existing provider requires using wevtutil with the ETW Manifest file removed. Fortunately, only the provider name and GUID are used in this process, so the file you provide doesn't need to be the one registered, just one that matches the currently registered provider name and GUID. Thus, the ETW Manifest file in your package will do for this removal operation.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The following C# code demonstrates this removal:

```
private void RemoveExistingProvider(string PathToEtwManifestFile)
{
  // NOTE: This operation requires a manifest file with the correct
     manifest name
  // and guid fields only,
  //  it need not be the original file.
  System.Diagnostics.Process p = new
  System.Diagnostics.Process();
  p.StartInfo.UseShellExecute = false;
  p.StartInfo.FileName = Environment.GetFolderPath
  (Environment.SpecialFolder.System) + "\\wevtutil.exe";
  p.StartInfo.Arguments = "um " + "\"" +
  PathToEtwManifestFile + "\"";
  p.StartInfo.CreateNoWindow = true;
  p.Start();
  p.WaitForExit();
 }
```

The registration involves using the ETW Manifest file that is built into one of the components of your application, but you will need to modify the contents of this file before registration. That is because this file contains path information for the full path of the EXE/DLL file containing the message and resources. Your code does not care about that path, but the registration does. Under MSIX, that path will contain version information as well as a hash of the signature, and it possibly could be installed to an alternative volume, so you cannot even assume “C:\Program Files\WindowsApps” for all cases.

As files in your package cannot be written to, you can load the ETW Manifest from your package (using relative pathing from the ExecutionAssembly folder, for example) into an XML document and rewrite the path to these parameters, saving into the user’s temp folder. Then you can run wevtutil.EXE against this copy.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The following C# code is an example of rewriting the file into a copy and registering the provider and log:

```
    private void RegisterTMEditXEventLog( string ProviderName,
                                          string PackageDir,
                                          string RelativePathToManFile,
                                          string RelativePathToResourceFile,
                                         string RelativePathToMessageFile)
  {
    string UseFileName = PackageDir + "\\" + RelativePathToManFile;
    FileInfo fMan = new  FileInfo(UseFileName);
    if (fMan.Exists == true)
    {
      XmlDocument xMan = new XmlDocument();
      xMan.Load(UseFileName);
      XmlNodeList xNodeList = xMan.GetElementsByTagName("provider");
      string sresourceFileName = null;
      string smessageFileName = null;
      // NOTE: This code assumes dlls are at package root.
      Foreach (XmlNode xNode in xNodeList)
      {
        foreach (XmlAttribute xAtt in xNode.Attributes)
        {
          if (xAtt.Name.Equals("resourceFileName"))
          {
           sresourceFileName = PackageDir + "\\" + RelativePathToRseourceFile;
           FileInfo fResourceFile = new FileInfo(sresourceFileName);
            if (fResourceFile.Exists == true)
            {
              xAtt.Value = sresourceFileName;
            }
          }
          else if (xAtt.Name.Equals("messageFileName"))
          {
            smessageFileName = PackageDir + "\\" + RelativePathToMessageFile;
            FileInfo fMessageFile = new FileInfo(smessageFileName);
            if (fMessageFile.Exists == true)
            {
              xAtt.Value = sresourceFileName;
            }
          }
        }
      }
      UseFileName = Environment.GetFolderPath(Environment.SpecialFolder.LocalAppli
                    cationData) + "\\Temp\\" +
                    System.IO.Path.GetRandomFileName() + ".man";
      xMan.Save(UseFileName);

      System.Diagnostics.Process p = new System.Diagnostics.Process();
      p.StartInfo.UseShellExecute = false;
      p.StartInfo.FileName = "c:\\windows\\system32\\wevtutil.exe";
      p.StartInfo.Arguments = "im " + "\"" + UseFileName + "\"";
      p.StartInfo.CreateNoWindow = true;
      p.Start(); p.WaitForExit();

      File.Delete(UseFileName);
    }
  }
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.14 Other MSIX limitations

There are some known limitations and possible workarounds for packaging existing Win32 desktop applications to MSIX and to run the software in the MSIX container. While we cannot cover everything that might be an application compatibility issue for your software, it is recommended to review the information in this section to enable a smoother transition to this exciting new format.

The most common of these items include:

- In-process shell extensions and COM DLL servers.
- Kernel-mode drivers.
- Issues with writing to AppData folder and current installation directory.
- .Net applications targeting older version of .Net framework.

The Microsoft documentation link below provides a detailed explanation about a long list of issues seen at this link:

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

Additionally, we suggest reviewing the following, some of which is covered in the reference above with additional advice here, and some of which may be new.

4.14.1 In-process shell extensions and COM DLL servers

Since the posting of the document referenced above, some support for shell extensions and COM DLL servers has been added to MSIX, and we cover those in this book (see 4.8 COM support). Microsoft has not yet updated the reference above.

When it comes to shell extensions, what we have found is that as long as your shell extension does not need access to the virtual registry of your package the AppXManifest entries you must add to expose the shell extensions seem to work OK. We anticipate Microsoft supporting more shell extensions in the future as part of Windows 11, so be sure to check the latest Microsoft documentation as well.

4.14.2 Targeting older Windows operating systems using MSIX core

MSIX Core enables the installation of MSIX apps on previous versions of Windows, such as Windows 7. It only provides a subset of features of native MSIX and does not provide the full benefits that come with MSIX (for example, container benefits of native MSIX and store deployment support).

Basically, this is intended for a situation where your code requires no incompatible code/structure changes to run under MSIX and allows you to have a single installer that works on both old and new platforms. This attempt at providing vendors a transitional capability is a great idea in theory but will

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

undoubtedly create more problems. You might be better off supporting the older platforms with your existing installer technology.

The Microsoft link below talks about the most recent considerations using MSIX Core from Microsoft's perspective:

https://docs.microsoft.com/en-us/windows/msix/msix-core/msixcore

4.14.3 Targeting Windows server 2019

MSIX is now supported on Windows Server 2019 (LTSC) with the Desktop Experience. Because Windows Server 2019 with the Desktop Experience does not support all features of Windows 10 client SKUs, Microsoft recommends testing converted MSIX packages on Windows Server 2019 before deploying in production. A few packaged (com and firewall) extensions are not supported, and store options are not available on Windows 2019 Server. For the most recent updates on this, please refer to the Microsoft documentation below:

https://docs.microsoft.com/en-us/windows/msix/msix-server-2019

4.15 Custom install actions

There are many instances where you will need to add certain prerequisites to a software package for the application to be fully functional upon installation. Such prerequisites may include checking for certain conditions on a machine or installing another software package along with the main package. The software packages that are added as a prerequisite to a main package are referred to as redistributable components or redistributable packages or just redistributables. These redistributables can be third-party packages too. For example, if your application is developed in .Net, you might want to add a .Net framework redistributable as a requisite to your package. Upon installation of your application, .Net framework too will get installed, so your application will function as desired.

The CustomInstall extension provides a way to add additional installers, like a redistributable, to an MSIX package. The CustomInstall extension has sub-nodes to define what parameters should be passed to the additional installer for installation, repair, and uninstallation actions. It is recommended to use silent command line switches to be passed as arguments for these actions. CustomInstall extension will requirethe customInstallActions restricted capability to be added under Capabilities in the AppXManifest file.

Microsoft had made it clear in its documentation that CustomInstall extension is currently intended to be used only by certain types of desktop PC games that are published by Microsoft and its partners.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Here is an example of how the CustomInstall extension is used in the AppXManifest file:

```
<Package …>
     …
     <Applications>
          …
     </Applications>
     <Extensions>
          <desktop6:Extension Category="windows.customInstall">
               <desktop6:CustomInstal
                Folder="Redistributables">
                    <desktop6:InstallActons>
                         <desktop6:InstallAction
                          File="SamplePRQ.exe"
                          Name="Sample Prerequisite"
                          Arguments="/i /silent" />
                    </desktop6:InstallActions>
                    <desktop6:RepairActions>
                         <desktop6:RepairAction File="SamplePRQ.exe"
                          Name="Sample Prerequisite"
                          Arguments="/r /silent" />
                    </desktop6:RepairActions>
                    <desktop6:UninstallActions>
                         <desktop6:UninstallAction File="SamplePRQ.exe"
                          Name="Sample Prerequisite"
                          Arguments="/x /silent" />
                    </desktop6:UninstallActions>
               </desktop6:CustomInstall>
          </desktop6:Extension>
     </Extensions>
     <Capabilities>
          <rescap:Capability Name="customInstallActions"/>
     </Capabilities>
</Package>
```

Reference:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-desktop6-custominstall

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4.16 Package Updates and AppInstaller File

It is fairly common for application vendors to include some sort of software update mechanisms as part of their product. These will require attention as part of your project.

Your current software may have a mechanism whereby it makes a query to your website to determine if an updated version is available. Upon detecting the new version, it might notify the user or initiate an update automatically. Sometimes this detection logic is placed in an external process, such as the Windows Schedule Tasks, and sometimes it is in the application itself. Usually, the app itself must be shut down to perform the update (to prevent the need to reboot). Some vendors will add an update service to perform the update.

It is a very good practice that if you have an updater, you make it easy for your commercial customers to turn off the updater. While automating updates for consumers and SMB customers is probably a good thing, your enterprise customers often want their IT department to be in charge of when updates are made to end-user desktops. MSIX includes an optional support for updating your packages that meets these needs and that you should leverage.

Doing this is simple. Ninety-nine percent of the effort will be to remove your existing update mechanisms.

In addition to building an MSIX package as your “installer,” you can also create a companion AppInstaller file. This XML file acts as a handler that manages the updates for you. If you distribute your package through the Microsoft Store, Microsoft creates and manages the AppInstaller for you so you don’t have to worry about it.

For delivery from your own website, you would want to make two links available to your customers to get the software:

- One link provides the MSIX/Bundle file that they may download and manage themselves; this would be used by those commercial customers who want to manage installation and delivery by their IT staff. Your website (especially IIS-based ones) might need a small modification to allow download of the .msix file type.
- The second link provides the AppInstaller file. This XML-based file provides a url link to itself and another url link to the MSIX/bundle package. When the package is installed using the AppInstaller file, the entries in this file will direct the OS to periodically check the first of those links to see if the AppInstaller file has been updated (there is a version field), and if so, to trigger the update. The OS will handle the update implementation for you. This link provides customers wanting the updates to be automated to get the same style of updating that the in-box and Microsoft Store-sourced packages receive.

The AppInstaller framework that is built into the Windows operating system comes with ms-appinstaller protocol support, and this may be used to deploy MSIX package\bundle from a web page or a file share

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

as stated above. Depending on the configuration inside the AppInstaller file, this also can handle automatic updates based on scenarios configured.

Your website content to deploy MSIX packages\bundles using an AppInstaller file and the AppInstaller framework would look something like the sample below:

```
<html>
  <body>
    <h1> Web Page for My app Deployment</h1>
    <a href="ms-appinstaller:?source=http://mywebsite.test.com/myapp.appinstaller"> Install My
    app </a>
   </body>
</html>
```

The Microsoft document “Installing Windows 10 apps from a web page” provides a good introduction and is found at this link:

https://docs.microsoft.com/en-us/windows/msix/app-installer/installing-windows10-apps-web

The AppInstaller file mentioned in the href in the sample above is an XML-formatted file, traditionally using a .appinstaller file extension. This file would also be placed on the website and the website configured to support downloads of this new file extension.

Microsoft documentation on the schema reference for the AppInstaller file itself is found at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appinstallerschema/schema-root

Although this documentation was written for .AppX (UWP) packages, support for MSIX and MSIX bundles has been added, even if this documentation hasn’t caught up.

The most basic AppInstaller file would look something like this:

```
<?XML version="1.0" encoding="utf-8"?>
<AppInstaller Version="1.4.0.0"
   Uri="https://server/folder/subfolder/Product.appinstaller"
  xmlns="http://schemas.microsoft.com/appx/appinstaller/2017/2">
     <MainPackage
        Name="PackageName"
        Publisher="CN=Company"
        Version="1.4.0.0"
        ProcessorArchitecture="x64"
       Uri=" https://server/folder/subfolder/
         PackageName_1.4.0.0_x64_____abcdefghijklm.msix" />
</AppInstaller>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Or if you provide an MSIX bundle file instead of an MSIX file, it would look like this:

```
<?XML version="1.0" encoding="utf-8"?>
 <AppInstaller Version="1.4.0.0" Uri=https://server/folder/
  subfolder/Product.appinstaller
     xmlns="http://schemas.microsoft.com/appx/appinstaller/
      2017/2">
           <MainBundle Name=”PackageName”
           Publisher="CN=Company"Version="1.4.0.0"
           Uri="https://server/folder/subfolder/PackageN
           ame_1.4.0.0___abcdefghijklm.msixbundle" />
</AppInstaller>
```

The file must contain either a MainPackage or MainBundle element (but not both). You may also optionally add additional package references using one of these elements as additional child elements of the AppInstaller root element:

- Dependencies
- OptionalPackages
- RelatedPackages

The final optional child element of the root AppInstaller element is the UpdateSettings element, which is used to control the updates. Without this element, automatic updates would not occur. This element supports three optional child elements:

- OnLaunch – Causes the OS to check for updates whenever the application is launched by the end-user. There is an optional parameter (HoursBetweenUpdateChecks) that may be used to limit how often this check is made. This update would happen in the background and take effect whenever the user shuts down the application.
- AutomaticBackgroundTask – The OS will check for updates every eight hours whether or not the app is running. This is only available if the end-user is on OS 1803 or above (silently ignored otherwise). The element supports no parameters of child elements.
- ForceUpdateFromAnyVersion–Used as a flag to indicate the ability to upgrade or downgrade. When not present or unsupported (also requires OS 1803 or above), only upgrades are possible. This element expects to contain a value of true when present.

The Microsoft documentation link below offers guidance on troubleshooting installation issues with the AppInstaller file: https://docs.microsoft.com/en-us/windows/msix/app-installer/troubleshoot-appinstaller-issues

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 5: Setup Project

There are a lot of MSIX packaging tools available in the market, which automate the packaging process from the payload files configured in the tool and validate the MSIX AppXManifest file generated during the build.

MSIX packaging tools generate the MSIX AppXManifest file during the build from the Applications, Visual Assets, Capabilities, and Extensions configured in the Tool's integrated development environment (IDE). They also validate the MSIX AppXManifest file against schema and report the error. MSIX packaging tools provide the graphical user interface to configure the Capabilities, Visual Assets, and Declarations to make the developer's life easy, instead of directly updating the MSIX AppXManifest file.

This article provides step-by-step instructions to build an MSIX package with InstallShield **MSIX project** and Microsoft Visual Studio **Windows Application Packaging Project.** In both projects, we are going to add a **File Type Associations** extension, an Application icon in the Visual Assets, enable package Capabilities, and finally the Application entry itself.

File Type Associations extension registers the file type with the operating system to open the file with your application when you double-click its icon. An extension is like an agreement between an application and the operating system. Extensions let application developers extend or customize standard features primarily for use in their applications and potentially for use in other applications.

NOTE: Setup Projects used in this chapter are available on this GitHub page:

https://github.com/flexera-public/MSIXDevGuide/tree/main/Chapter_5_Setup_Project

5.1 Step-by-Step instructions to build MSIX package with InstallShield MSIX project

Revenera InstallShield (formerly Flexera InstallShield) is the fastest, easiest way to build Windows installers and MSIX packages and create installations directly within Microsoft Visual Studio.

InstallShield supports:

- Creating MSIX package from scratch.
- Building Modification package.
- Adding Package Support Framework to detour the package runtime errors.
- Validating the package and signing the package during the build.

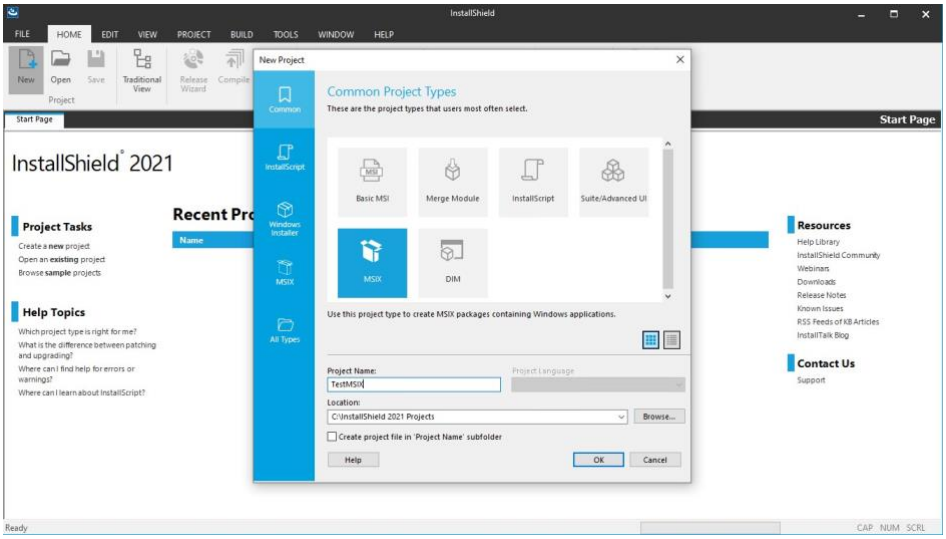
The instructions in this section will allow you to create an MSIX project in InstallShield to package a native Win32 application, a simple Win32 application to open a window when it executes, and the File Type Associations extension to register the Win32 application to handle files of the specified types

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

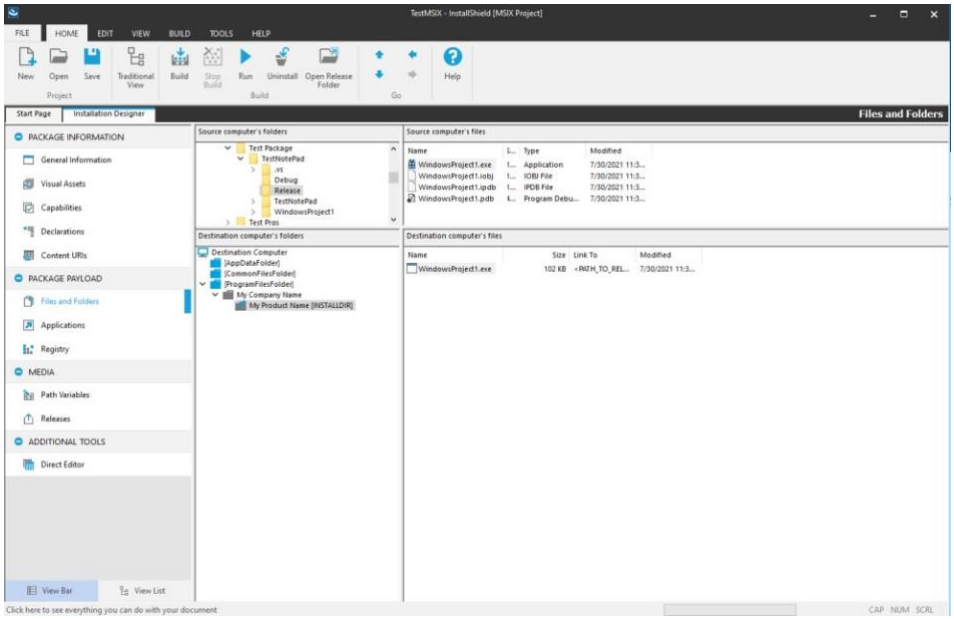
Flexera Inform IT. Transform IT.
revenera

(.rev). After creating and installing the created MSIX package, double-clicking a file of the .rev type will cause the Win32 application to be opened with the clicked file.

1. Launch InstallShield.
2. Click on the New button. Select MSIX Project type. Provide a project name and click OK.



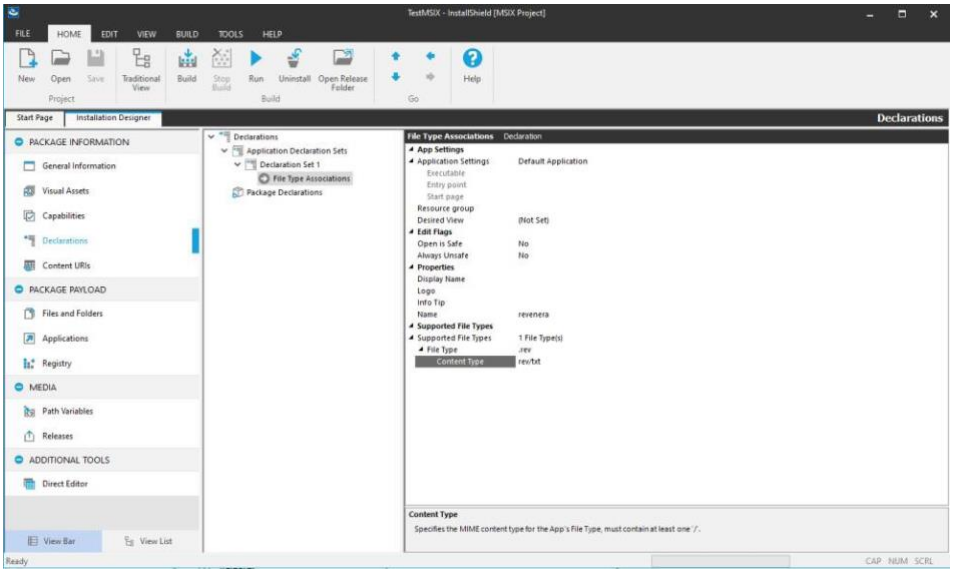
3. Navigate to the Files and Folders view to add the Application files. So for this MSIX project, add the Win32 application built to show a window when it executes.



Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

4. Navigate to "Declarations view," where you can declare the extensions required for your Application. To add an application extension, right-click on ‘Application Declaration Sets’ and then add a "New Declaration Set." Then right-click on the ‘Declaration Set’ created, which lists all the application declarations supported by InstallShield. Add File Type Association declaration from the list. One Declaration Set can contain multiple declarations, and likewise Declaration Sets can contain a multiple Declaration Set. This will allow you to map unique and separate declarations to each application entry in the MSIX AppXManifest.

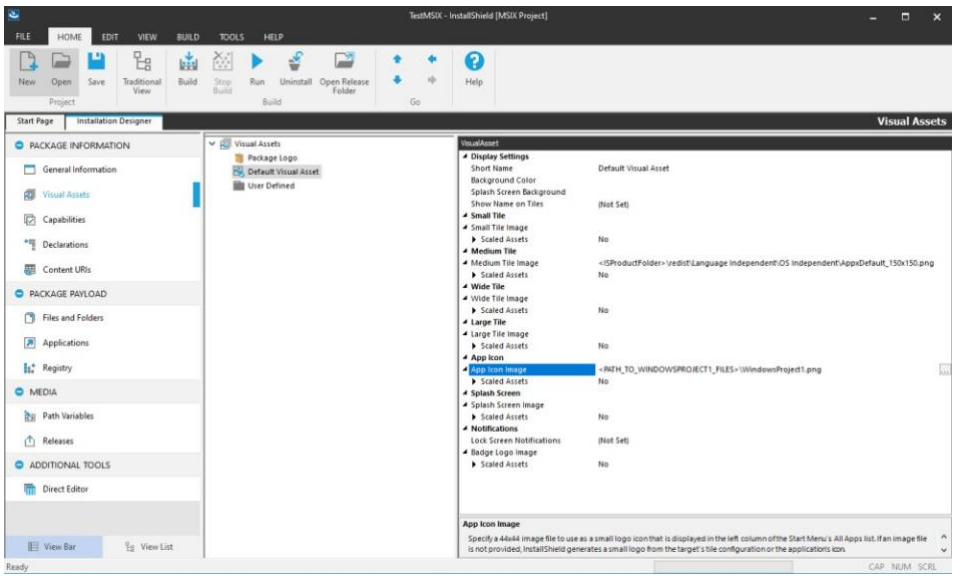


5. Enter the File Type Association declaration details, such as the file type (. rev here), to be registered and Name. Likewise, you need to provide the application-specific details for all the declarations added under the Declaration Set.

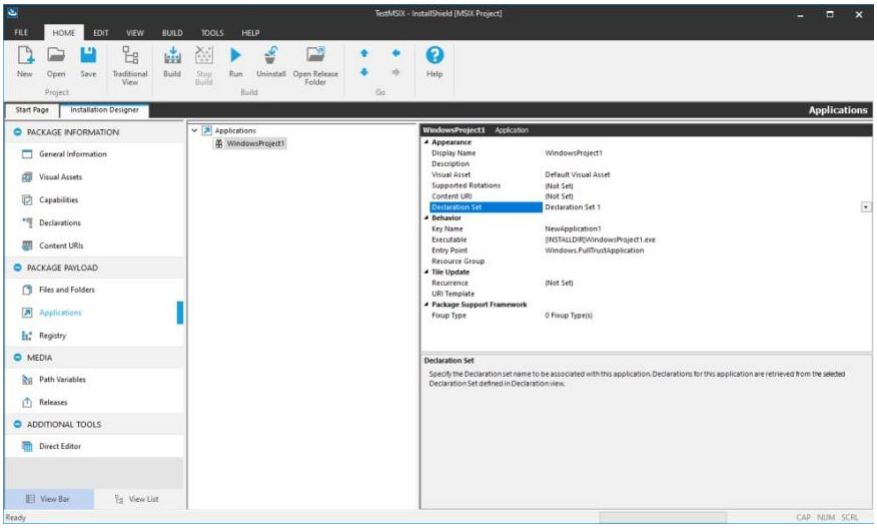
Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

6. Navigate to Visual Assets view, where you can configure the visual aspects of the MSIX application. Here, you can configure the application icon image, various tile images, background color, and the package logo. For the sample package, Application icon image is configured to a new image as shown in the screen shot below. Likewise, Visual Assets view supports creating user-defined visual assets to configure against each application entry.



7. Navigate to "Applications" view, right-click on "Applications" in the center pane, select 'New Application, and browse the application added in the Files and Folders view (WindowsProject1.EXE in step 3). And Select the ‘Declaration Set’ configured in the Declaration View for the ‘Declaration Set’ entry as in the image below to map to the application. Also, select the ‘Visual Asset’ to the configured visual asset for this application entry in the Visual Asset view. Application Entry Point

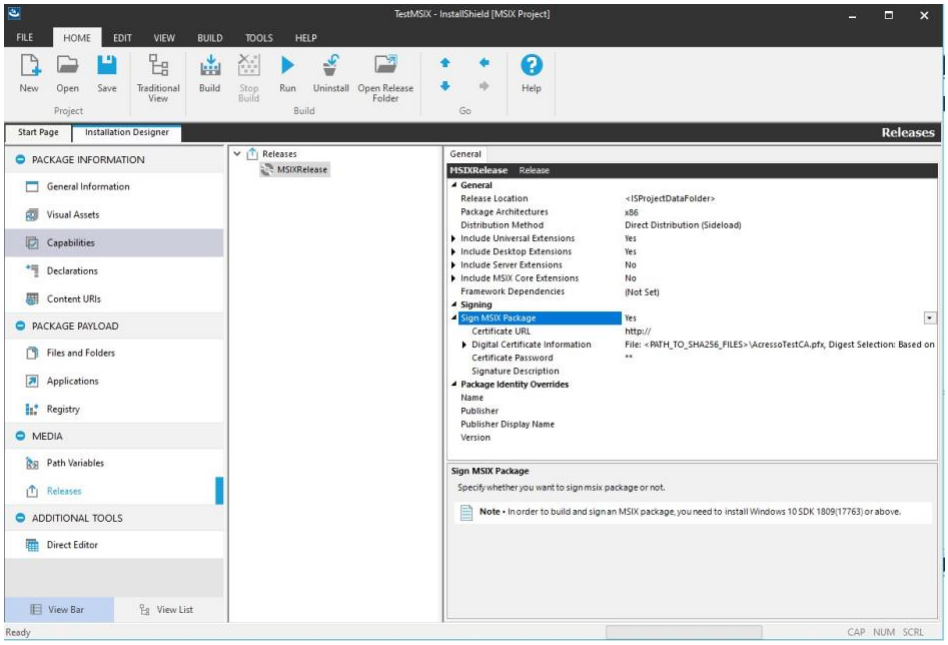


Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

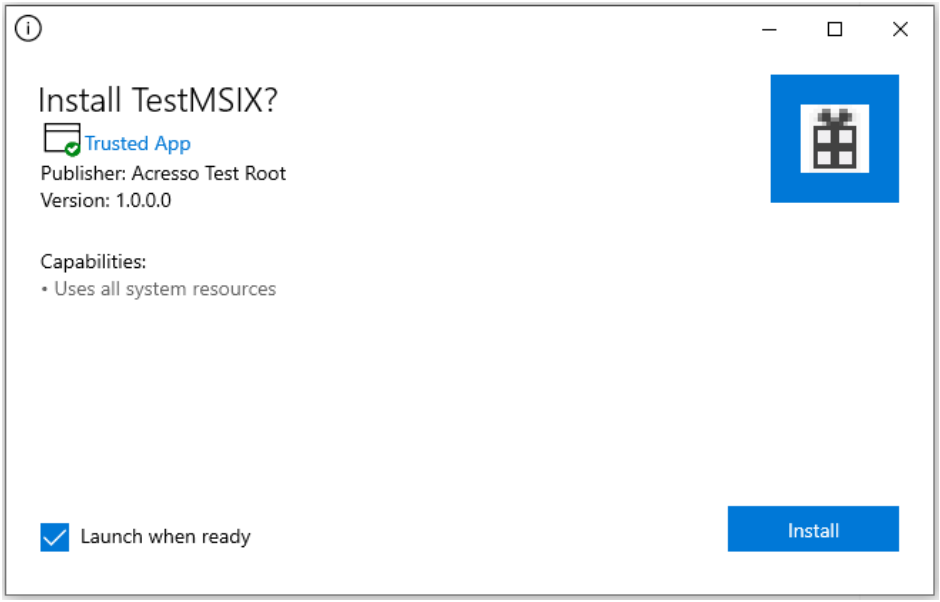
Flexera
Inform IT. Transform IT.
revenera

needs to be “Windows.FullTrustApplication” for the Win32 desktop application, whereas for a regular UWP it is the fully qualified namespace name of your app class.

8. MSIX Packages need to be digitally signed. So now head over to 'Releases' view, click on the ellipses for "Digital Certificate Information," and provide your digital signing information.



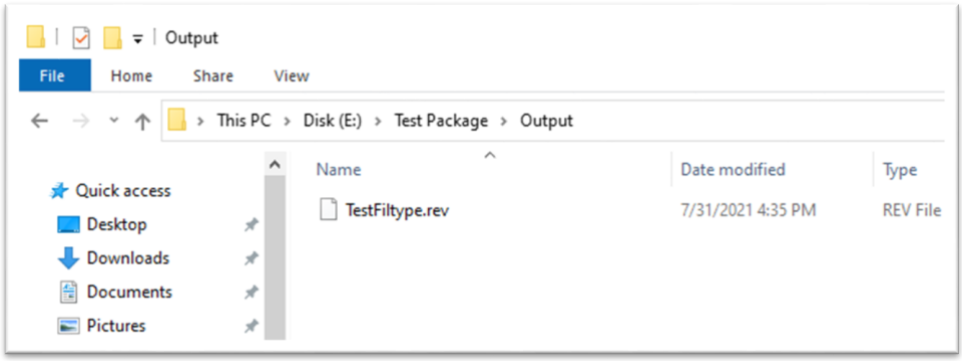
9. Click on Build in the ribbon bar.
10. Click on Run in the ribbon bar.



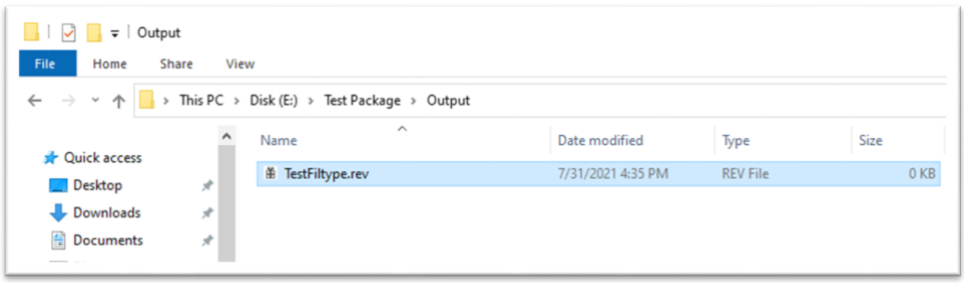
Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

11. Click on Install, and the installation should be successful. The WindowsProject1 app should launch after installation is done and the checkbox 'Launch when ready' is checked.
12. Check the file association before and after installing the MSIX package. Before the installation, the .rev file extension was not associated with any application:



After the installation, the .rev file extension is registered with the application (WindowsProject1.EXE, in this case) installed in the MSIX package; the file icon changed to the application icon configured in the 'Visual Assets' view, and double-clicking on the 'TestFilType.rev' file opened the registered application.



Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

13. Auto-generate MSIX AppXManifest file during the build by InstallShield based on the configuration.

```
<Identity Version="1.0.0.0" Publisher="CN=Acresso Test Root, O=Organization, OU=Org Unit,
    L=Schaumburg, S=IL, C=US" ProcessorArchitecture="x86" Name="TestMSIX"/>
- <Properties>
    <DisplayName>TestMSIX</DisplayName>
    <PublisherDisplayName>Acresso Test Root</PublisherDisplayName>
    <Logo>Assets\StoreLogo.png</Logo>
  </Properties>
- <Resources>
    <Resource Language="en-us"/>
  </Resources>
- <Dependencies>
    <TargetDeviceFamily Name="Windows.Universal" MaxVersionTested="10.0.18362.0"
      MinVersion="10.0.16299.0"/>
    <TargetDeviceFamily Name="Windows.Desktop" MaxVersionTested="10.0.18362.0"
      MinVersion="10.0.16299.0"/>
  </Dependencies>
- <Capabilities>
    <rescap:Capability Name="runFullTrust"/>
  </Capabilities>
- <Applications>
  - <Application EntryPoint="Windows.FullTrustApplication" Executable="VFS\ProgramFilesX86\My
    Company Name\My Product Name\windowsproject1.exe" Id="NewApplication1">
    - <uap:VisualElements Square44x44Logo="Assets\NewApplication1-AppList.png"
      Square150x150Logo="Assets\NewApplication1-MedTile.png" BackgroundColor="transparent"
      Description="WindowsProject1" DisplayName="WindowsProject1">
        <uap:DefaultTile ShortName="Default Visual Asset"/>
      </uap:VisualElements>
    - <Extensions>
      - <uap:Extension Category="windows.fileTypeAssociation">
        - <uap:FileTypeAssociation Name="revenera">
          - <uap:SupportedFileTypes>
              <uap:FileType ContentType="rev/txt">.rev</uap:FileType>
            </uap:SupportedFileTypes>
          </uap:FileTypeAssociation>
        </uap:Extension>
      </Extensions>
    </Application>
  </Applications>
  <Extensions/>
- <build:Metadata>
    <build:Item Version="2021 R1" Name="InstallShield"/>
    <build:Item Version="10.0.19041.1023" Name="OperatingSystem"/>
  </build:Metadata>
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

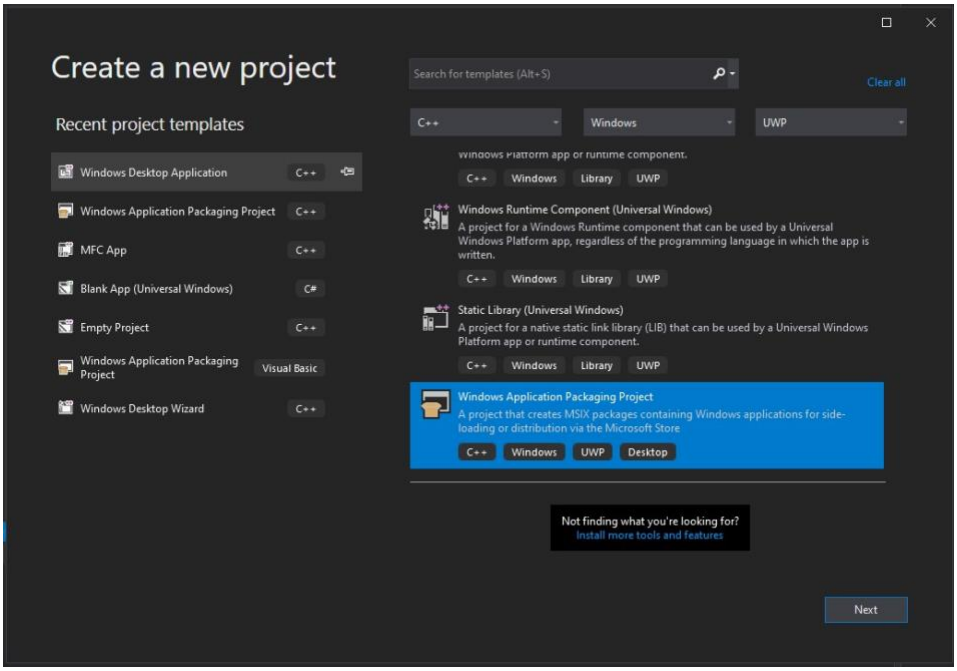
Flexera
Inform IT. Transform IT.
revenera

5.2 Step-by-Step instructions to build MSIX package with Windows Application Packaging project

Microsoft Visual Studio **Windows Application Packaging Project** (WAP) can build the MSIX package from the referenced Application project, and the **Universal Windows Platform (UWP) projects** also can build the MSIX package for the UWP Application. To generate an MSIX package for a desktop application, you'll need to add the Windows Application Packaging Project to the solution.

Windows Application Packaging Project is available in Visual Studio 2017 15.5 version and later. To get the Windows Application Packaging Project template in the New Project Dialog, you'll need to install the “Universal Windows Platform development” workload or the “MSIX Packaging Tools” workload while installing Visual Studio.

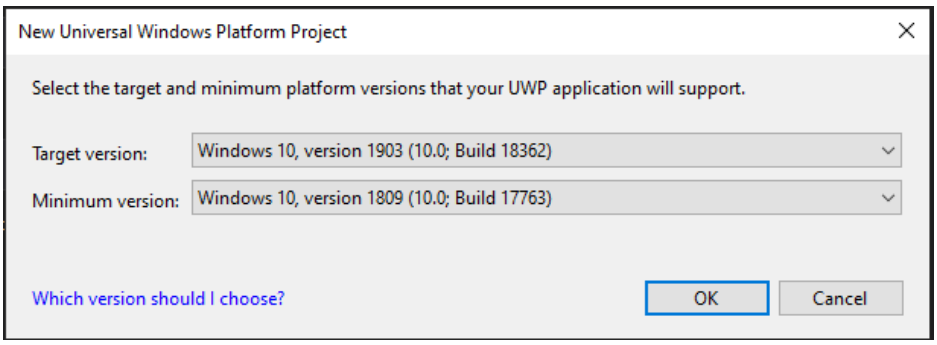
1. Launch Visual Studio.
2. Create new **Windows Application Packaging Project** or add **Windows Application Packaging Project** to your existing Visual Studio solution.



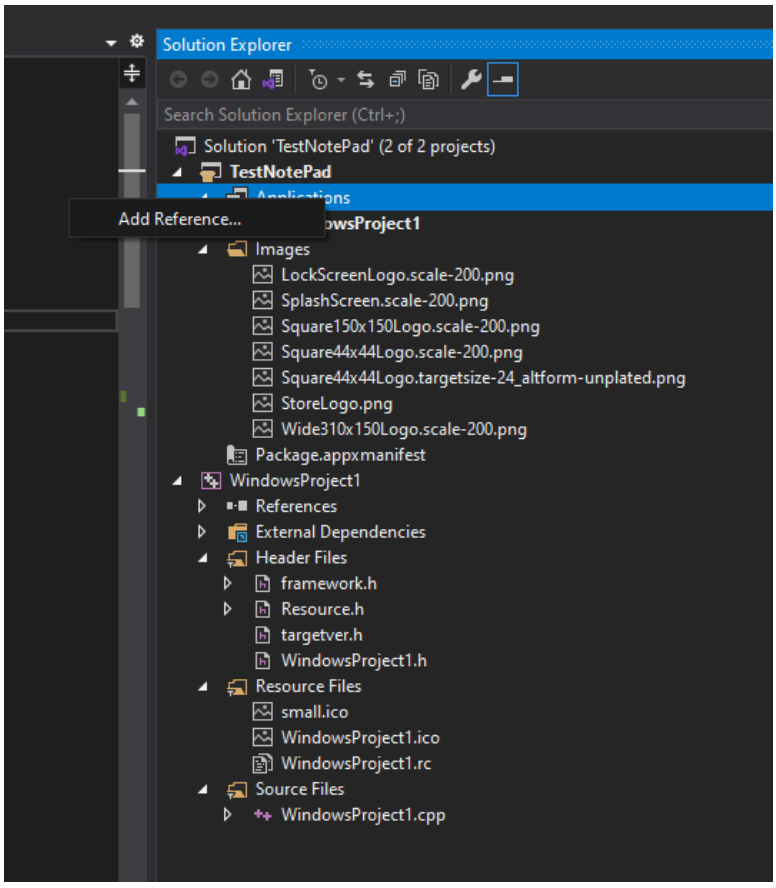
Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

3. Set the target version of the package. To build the MSIX package, select the version according to the supported platforms (see https://docs.microsoft.com/en-us/windows/msix/supported-platforms#package-format-support).

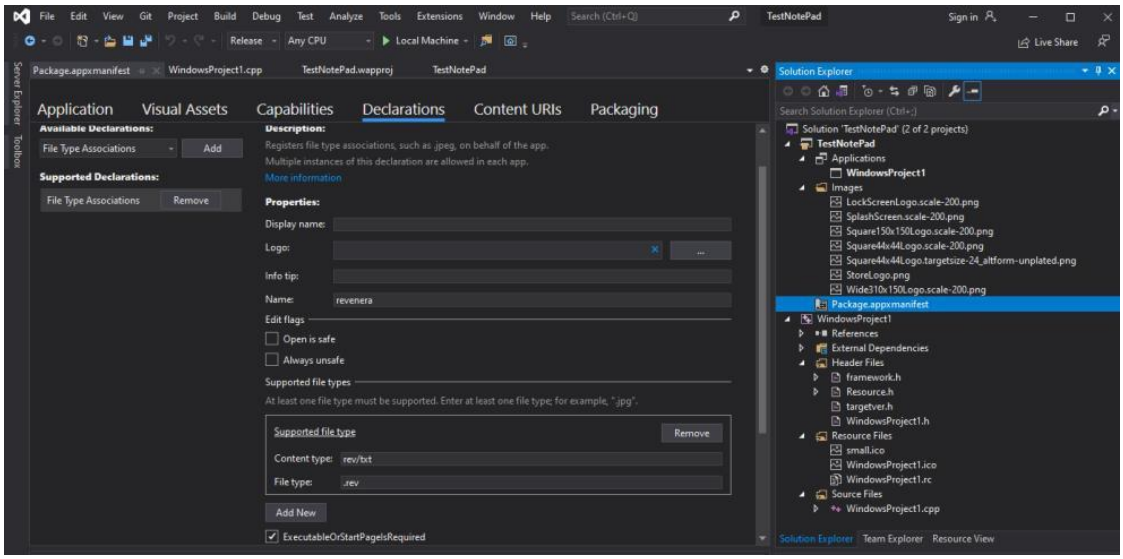


4. Add Applications to the packaging project by right-clicking on the Applications folder and selecting "Add Reference" to select the application project. Select the WindowsProject1 project here. A simple Win32 application will open a window when it executes.



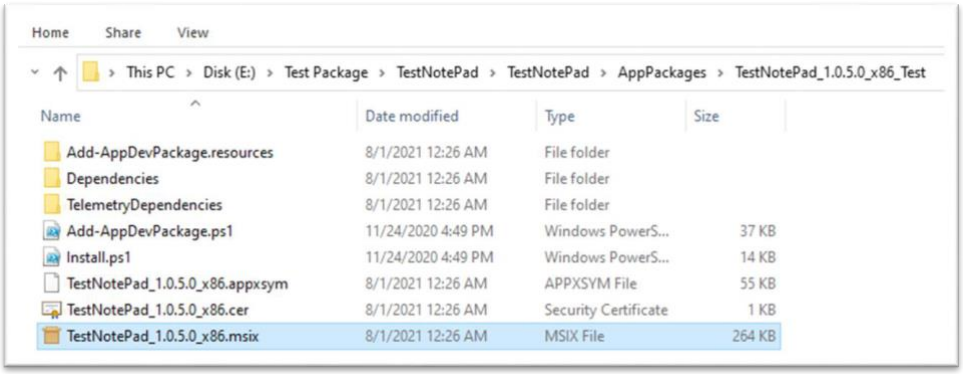
Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

5. Configure the Capabilities, Declarations, and Visual Assets required for the Application. Add File Type Association declaration from the Supported Declarations list and configure the file type information. Configure the Visual Assets for the Application icon and the package logo. Visual Studio provides the option to configure the package either through the package AppXManifest XML editor or through the graphical user interface.



6. Build the packaging projects to ensure there are no errors and select the correct platform that you want to create the package from in the configuration manager.

7. To build the MSIX package, Select Publish->Create app Packages by right-clicking on the packaging project (before Visual Studio 2019 version 16.3, the Publish menu is named Store). It takes you to the “Create app Packages” wizard, where you can configure the packaging sideloading option, package signing option, package architecture, version, and package bundle options, and finally to create a package to build the MSIX Package.



Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

8. MSIX AppXManifest file is generated by Visual Studio Windows Application Packaging Project.

```
<Identity ProcessorArchitecture="x86" Version="1.0.5.0" Publisher="CN=TestMSIX" Name="32612acc-983b-
    4491-98a3-54f496819c09"/>
- <Properties>
    <DisplayName>TestNotePad</DisplayName>
    <PublisherDisplayName>HKazhungil</PublisherDisplayName>
    <Logo>Images\StoreLogo.png</Logo>
  </Properties>
- <Dependencies>
    <TargetDeviceFamily Name="Windows.Universal" MaxVersionTested="10.0.18362.0"
      MinVersion="10.0.17763.0"/>
    <TargetDeviceFamily Name="Windows.Desktop" MaxVersionTested="10.0.18362.0"
      MinVersion="10.0.17763.0"/>
    <PackageDependency Publisher="CN=Microsoft Corporation, O=Microsoft Corporation, L=Redmond,
      S=Washington, C=US" Name="Microsoft.VCLibs.140.00.UWPDesktop" MinVersion="14.0.29016.0"/>
  </Dependencies>
- <Resources>
    <Resource Language="EN-US"/>
  </Resources>
- <Applications>
  - <Application EntryPoint="Windows.FullTrustApplication" Executable="WindowsProject1
    \WindowsProject1.exe" Id="App">
    - <uap:VisualElements Square44x44Logo="Images\Square44x44Logo.png"
      Square150x150Logo="Images\Square150x150Logo.png" BackgroundColor="transparent"
      Description="TestNotePad" DisplayName="TestNotePad">
        <uap:DefaultTile Wide310x150Logo="Images\Wide310x150Logo.png"/>
        <uap:SplashScreen Image="Images\SplashScreen.png"/>
      </uap:VisualElements>
    - <Extensions>
      - <uap:Extension Category="windows.fileTypeAssociation">
        - <uap:FileTypeAssociation Name="revenera">
          - <uap:SupportedFileTypes>
              <uap:FileType ContentType="rev/txt">.rev</uap:FileType>
            </uap:SupportedFileTypes>
          </uap:FileTypeAssociation>
        </uap:Extension>
      </Extensions>
    </Application>
  </Applications>
- <Capabilities>
    <Capability Name="internetClient"/>
    <rescap:Capability Name="runFullTrust"/>
  </Capabilities>
- <build:Metadata>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

5.3 Guidance on Files in your package

The structure of the files within your package is up to you, however there are some requirements and some practices that we recommend. In this section we also make suggestions in regard to the content of certain files.

Generally speaking, a simple flat structure is preferred, placing all of the files in a single folder, except for those that would have traditionally been placed in other system locations, in which case the VFS folder structure within your package might be appropriate. The use of other sub-folders under your package is optional, and if your current MSI or Setup EXE package uses sub-folders it is OK to retain that structure. You may need to keep in mind that changes in working directory and such may make DLL locating more difficult and as previously mentioned you might need to take actions to make the app work under MSIX.

The package XML files, such as the AppXManifest, pri, blockmap etc must be placed in the root folder.

Inside the AppXManifest file, any EXE that could be considered an entry-point of your existing package will need an Application entry. This includes not only entry-points like shortcuts, but also file type associations, protocol handlers, services, and other "api" like executables that might be present. Any EXE based entry-point that is a Win32 or .Net Framework based EXE will need to be marked for full trust (EntryPoint="WindowsFullTrustApplication") in that Application entry.

If the Package Support Framework (PSF) is included:

- The PSF components should all be placed at the root folder of the package. While placement in subfolders (especially the subfolder containing an EXE that needs them) is possible, for simplicity these should all be at the root.

- Except for PsfLauncherXX.EXE, the PSF files should not be renamed. It is best not to rename PsfLauncher, unless you have multiple Application entries in the AppXManifest. Each Application Executable in the AppXManifest requires a uniquely named executable, so when multiple Applications with PsfLauncher are required, you will need to place renamed copies in the package.

- Other PSF DLL (and EXE) files are provided in both 32 and 64 bit form as well. These should never be renamed. These files are provided in both 32 and 64 bit form, and the best practice is that you should include both the 32 and 64-bit forms whenever you need one of these files. Even if you know that your EXE is only 32-bit, if any of your code makes calls to open a system process, that process might run 64-bit and you'll need the 64-bit versions of the DLLs.

- If PsfLauncher is included in the package, you must include the PsfRuntimeXX.DLLs and PsfRunDLLXX.EXE files, along with the config.json file. Other Psf files are added only if you need them.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- The PSF configuration file must be named config.json, and should also be in the root folder of the package. The ID in the AppXManifest file for an Application must match the Application "id" of the config.json file. While there is a "name munging" used by some Microsoft software forcing the executable PsfLauncher64.EXE to use an id of "PSFLAUNCHERSixFour", this is NOT necessary for built packages and you are free to use IDs of your choosing.

- References in the config.json for fixup DLLs should not include the 32 or 64 portion of the filename (i.e. "FileRedirectionFixup.DLL"), as the Psf will determine whether the 32 or 64-bit version is needed and will adjust the filename automatically before attempting DLL loading.

- Signing of EXE and DLL files within the package is optional. For customers using security systems like AppLocker or Microsoft Defender Application Guard (MDAG), it is better that you sign your own components that are inside the package in addition to the package itself. Signing of the PSF components using your code-signing certificate is also a good idea (these are not signed by Microsoft). Signing of components sourced from third parties is not recommended.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 6: Testing and Debugging

There are several packaging tools available to build the MSIX package and AppXManifest.XML file, based on the user requirements and payloads. After building the MSIX package, the package may fail to deploy or may fail in runtime even though the package was built successfully by the packaging tool.

6.1 Handling installation errors

The MSIX installer shows the error codes and the error information during the installation if anything goes wrong and creates event log entries with the error details.

While traditional windows installer technologies usually generate a log file with the error details during the installation, MSIX installer does not generate any log file during the installation.

Instead, there will be a simple error code shown either in the AppInstaller GUI or in the error output of the PowerShell cmdlet. But additionally, there will be information in the Windows Event Logs.

To check the event viewer logs for the installed packages, open the event viewer by executing the *eventvwr.msc* from the run dialog. Then navigate to **Applications and Services Logs > Microsoft > Windows**, and check the logs under the categories below:

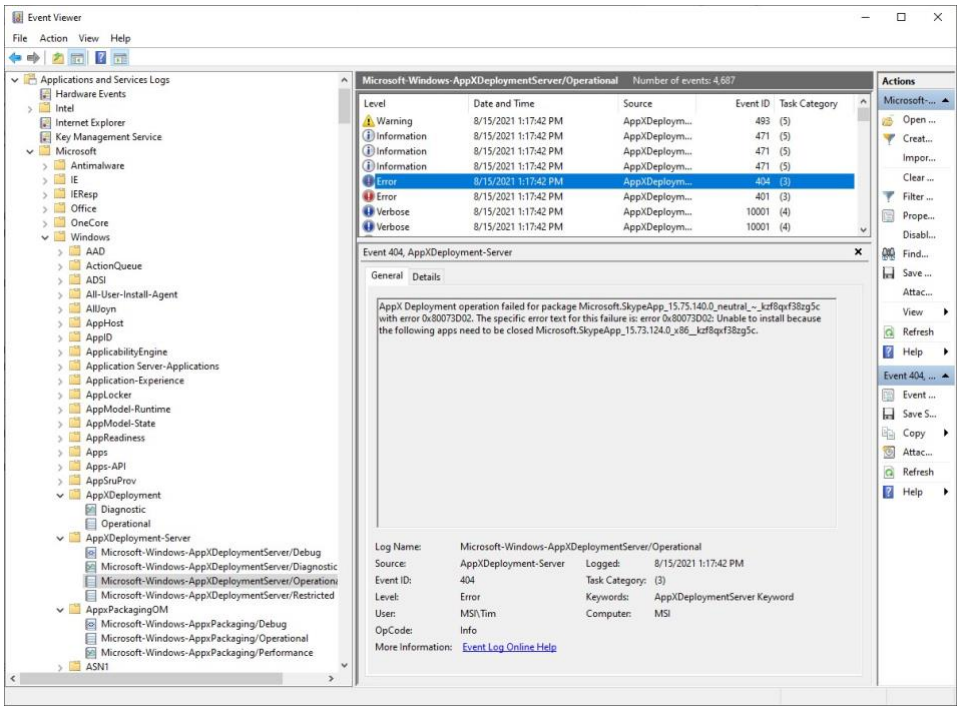
- **AppxPackagingOM > Microsoft-Windows-AppxPackaging/Operational**
- **AppXDeployment-Server > Microsoft-Windows-AppXDeploymentServer/Operational**

The higher-level ETW log of interest is the first log, while the AppXDeploymentServer log has more detailed items.

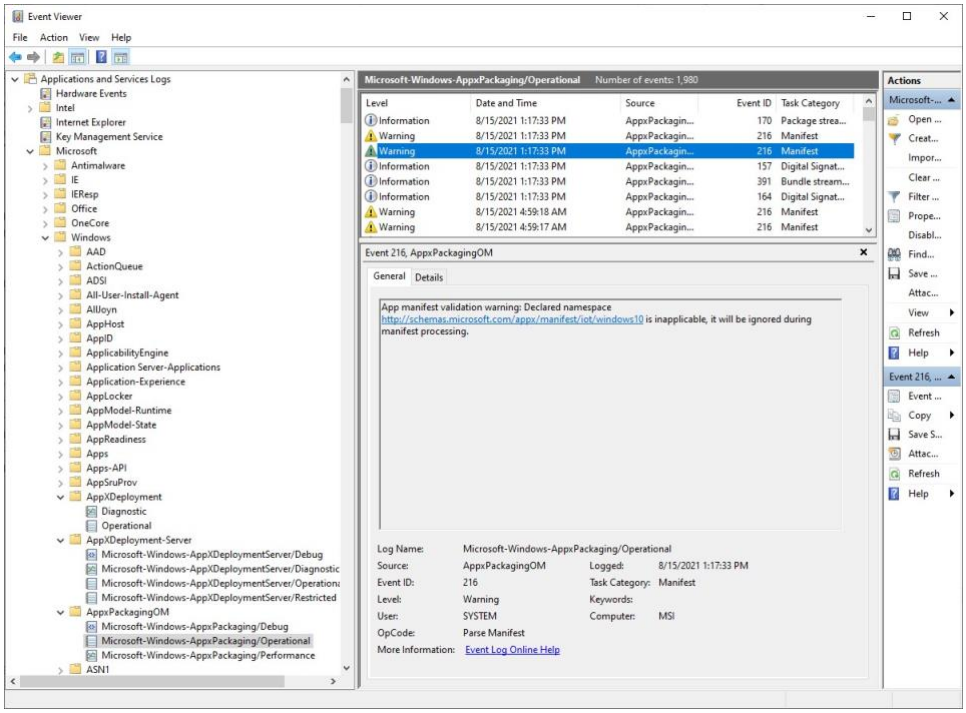
Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Here is an example screenshot of an error from the AppxPackaging log:



Here is an example screenshot of a benign warning from the second log:



If necessary, you can also enable some of those debug and diagnostic logs under the three AppX folders seen in the image.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

For more details on the common error code thrown during the MSIX package installation, refer to this document:

https://docs.microsoft.com/en-us/windows/Win32/appxpkg/troubleshooting#common- error-codes.

6.2 Digital code signing and validation errors

As we learned previously about code signing, MSIX packages need to be digitally signed to deploy the package, and the signing certificate subject name should match the Publisher attribute that is contained in the Identity element of the AppXManifest.XML file, or else the package will fail to install. Also, the signed certificate should be available in the target machine's trusted certificate store.

The following are examples of the types of issues commonly experienced when using SignTool to sign the package:

Publisher name mismatch with the certificate subject

MSIX package signing throws an error ("Error: SignerSign() Failed" -2147024885/0x8007000b) if the publisher entry in the AppXManifest file does not match the signing certificate subject.

To fix this error, be sure to update the publisher entry in the AppXManifest file with the signing certificate subject. To get the certificate subject, either double-click the certificate in File Explorer and then select the details tab or execute the "certutil-dump" command to dump the certificate information to the command line console window.

Bad PE certificate (0x800700C1)

Bad PE certificate error is thrown while signing the MSIX package if the package contains an executable with a corrupted certificate.

To identify the executables with a corrupted certificate, set an environment variable named APPXSIP_LOG with value "1" in a command prompt, and then sign the MSIX package by executing the SignTool.EXE with the certificate and the MSIX package details. This will dump the file names with the bad PE certificate in the console window.

6.3 Runtime Errors

The MSIX packaged application may fail to function during runtime either because of the application-specific errors or because of the restriction in the MSIX container. MSIX container-specific errors can be identified with the help of Process Monitor logs and other tooling.

6.3.1 Process Monitor

Process Monitor is typically the first tool used to help to trace the application's file and registry operations as well as their result. From the log, the file operation and the registry operations can be

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT. revenera

traced, and the result of the operation says either SUCCESS, PATH NOT FOUND, NAME NOT FOUND or ACCESS DENIED. So it is easy to identify why the application failed to execute in the MSIX container from the Process Monitor.

Applications originally designed for native deployment will often run into issues because of the more constrained environment of the “full trust” model of MSIX. Detecting the issue and modifying your code to avoid these issues is always the best policy, but adding in the Package Support Framework to provide better compatibility with your current code is always an option.

6.3.2 Package Support Framework

The Package Support Framework library helps to find and/or fix the runtime errors for the existing Desktop Application without modifying or rebuilding the Application code. Most Desktop Applications usually read or write files to the local directory or from a predefined folder, but a Desktop Application will fail to read or write the file in the MSIX container predefined folder because of restrictions imposed by the container. Including the Package Support Framework helps in converting the traditional Windows installer package into an MSIX package so that in many cases the Desktop Application can be run within the container without any code changes. The Package Support Framework provides multiple types of fixups to fix the Application runtime errors, such as the File Redirection Fixup, RegLegacyFixup, DynamicDLLFixup, and EnvVarFixup.

Additionally, the PSF includes a Trace Fixup and a special PsfMonitor program that provides a view of what occurs at the application API layer before hitting the container. Here is a quick overview of the fixup types available:

File Redirection Fixup is used to redirect attempts to write or read data in a directory that isn't accessible from an Application that runs in an MSIX container.

RegLegacy Fixup is used to modify registry calls to an acceptable form for being in the container.

DynamicDLL Fixup is used to ensure that your processes can find DLLs inside your package, overcoming certain MSIX limitations.

EnvVar Fixup is used to provide support for package-level environment variables.

Trace Fixup provides a detailed diagnostic view of the Application behavior, similar to Process Monitor. By default, the output from the Trace Fixup is sent to the attached debugger, or it can be viewed in the DebugView.

6.3.3 Starting a process inside the container

Invoke-CommandInDesktopPackage is a PowerShell cmdlet that can be used to run an Application inside the MSIX container. This allows users to run command prompt, registry editor, and PowerShell inside the MSIX container and browse through the merged file system and merged registry hive.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

NOTE: Originally this cmdlet only worked if the machine was placed in Developer Mode first. At some point (and we have found no documentation on when) Microsoft made a change to allow the cmdlet to run without enabling Developer Mode. It is known to work on Windows 21H1 and above, but the change may have occurred earlier.

The cmdlet below is run from a normal PowerShell window outside of any package. It launches the cmd.EXE from inside the specified MSIX container, and it helps navigate through the file system and the merged files from the MSIX container.

```
Invoke-CommandInDesktopPackage -AppId "App1" -PackageFamilyName " TestMultiplApps_d14rjvjqp047y"
-Command "cmd.exe" -Preventbreakaway
```

If the command to be run requires cmd line arguments, these may be added, such as

```
Invoke-CommandInDesktopPackage -AppId "App1" -PackageFamilyName " TestMultiplApps_d14rjvjqp047y"
 -Command "cmd.exe" -Preventbreakaway -Args "/c c:\filepath\file.cmd"
```

The cmdlet in the next example below launches the registry editor from the MSIX container, like the registry editor is running from inside the MSIX container, and it helps browse through the local machine and current user registry keys to identify the cause of the Application failure.

```
Invoke-CommandInDesktopPackage -AppId "App1" -PackageFamilyName " TestMultiplApps_d14rjvjqp047y"
 -Command "regedit.exe"
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

6.4 How do I know if a process is running inside a container?

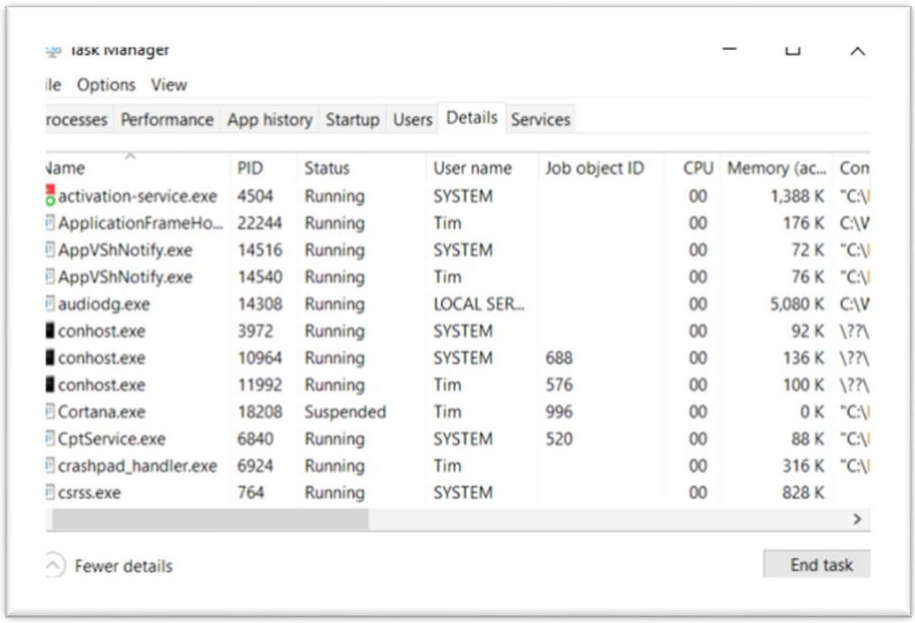
In general, once your software is running inside the container, any child process it creates should also run inside the container. However, there are exceptions to the general rule.

The child-process launching may be controlled by the caller by using EXTENDED_STARTUPINFO_PRESENT in the creation of the process to extend the startupinfo structure with new settings to request jumping outside of the container. As this is new, we don't expect that your current software would be using this so without action on your part the child should just run in the container as expected. Microsoft themselves uses this technique for certain types of scripting operations in the Package Support Framework.

But if you want to check for yourself, there are some simple techniques that you can use to determine if a process is inside a container. These techniques rely upon the notion that all MSIX and UWP processes running inside the container also run in a Job Object. While there are other forms of processes running inside Job Objects (for example Microsoft App-V virtualized applications run in Jobs in Windows 10), you can rely up these tests to understand if your software is running in the container.

1. Start the Windows Task Manager, and go to the Details Tab. Right click in the column header area and add in the Column named "Job object ID." Locate the rows with the processes you are interested in. Processes running in the same container will have the same Job object ID. Processes without a Job ID are running natively without a container.

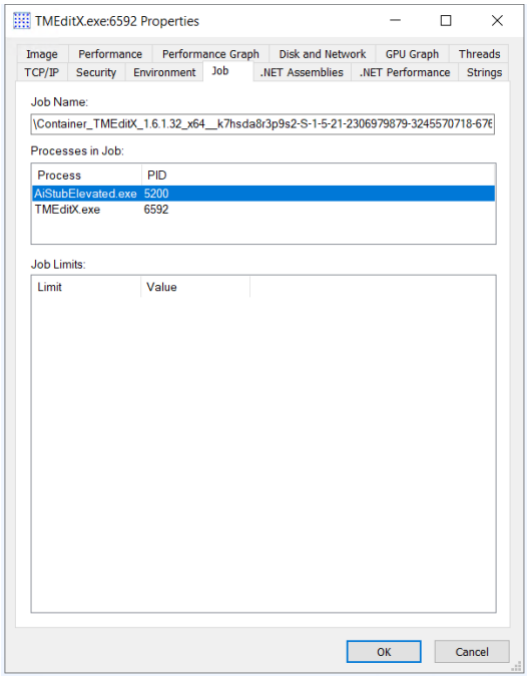
   Most of the time, the same Job object ID is used for all processes running inside the same container, but there are situations where two processes may have different Job object IDs showing in Task Manager and yet be running in the same container.



Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

2. Start Process Explorer with Run As Administrator. Locate the process of concern and double-click it. In the dialog, click on the Job tab (this tab is missing if you did not “RunAsAdministrator.” This tab will display a job name, and all processes running inside the job. There may also be Job Limits displayed, the Job Limits add controls to whether child processes may/must break away from the job.



Using the Process Explorer method appears to cover cases where the Task Manager shows different Job object IDs.

At this time, we have not found any indication in Process Monitor captured events to reliably determine if the process is running inside a container. This can make it difficult to determine if a short-lived process is running inside the container or not. If the PSF is used, you can check the Debug Console output in Visual Studio or SysInternals DebugView to determine if PsfRuntime was ever injected into the process.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

6.5 Debug the app with Visual Studio

Microsoft Visual Studio offers many ways to debug applications. When working with MSIX projects, the following methods are useful.

6.5.1 Using F5

The MSIX packaged Application can be debugged from Visual Studio by stepping through the source code to find where the Application is failing and the root cause. If the MSIX package is built using Visual Studio, then the Application can be debugged directly by pressing F5 or **Debug->**Start from the menu.

Keep in mind, however, that the application will not be running inside the MSIX container if debugged this way.

6.5.2 Using Installed app Packages

Visual Studio supports debugging the packages that are built using other packaging tools by attaching the app Package from the **Installed app Packages** list. To get the Installed app Packages list, select **Debug->Other Debug Targets->Debug Installed app Package.**

6.5.3 Using WaitForDebugger and AttachToExistingProcess

You can also build your project with the PSF Launcher and configure the WaitForDebugger option, install your package, and use the VisualStudio **Debug->Attach To Existing Process** menu. This involves the following steps:

- Add the appropriate architecture bitness (32/64) of the following components, matching the bitness of your program: PsfLauncher.EXE PsfRuntime.DLL WaitForDebuggerFixup.DLL. These files go into the root folder of your package.
- Add a config.json file to configure the program (see example below). This also goes into the root folder.
- Modify your AppXManifest file. Look for the Application element for your program EXE and change the name to that of the PsfLauncher. Set the AppID field to the appropriate value for the launcher used (either “PSFLAUNCHERThreeTwo” or “PSFLAUNCHERSixFour”).
- Build, Install, and Launch your program. PsfLauncher will start your program and inject the PsfRuntime, which will inject WaitForDebuggerFixup. When WaitForDebugger activates (initializes), it will wait for a debugger to attach. This prevents your application initialization code from running until you attach.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- In Visual Studio or other debugger, open the project or solution with your program code. You may need to select **Debug**->**Other Debug Targets**->**Native**. Then use the menu **Debug->Attach to Process** to attach the debugger into your paused application. When you attach a breakpoint will be hit in the WaitForDebugger code.
- Set any breakpoints in your code. Then continue debugging.

You can also avoid the PSF by studying the WaitForDebugger code and including the same thing directly in your code debug build. Assuming your code is not C++, then adding the DLL in the linker phase will work also.

Here is a sample template for a minimal config.json file for this purpose:

```
{
   "enableReportError": true,
   "applications": [
      {
         "id": "PSFLAUNCHERSixFour",
         "executable": "XYZ.EXE",
         "arguments": "",
         "workingDirectory": ""
      }
   ],
   "processes": [
      {
         "executable": "^PsfLauncher.*",
         "fixups": []
      },
      {
         "executable": ".*",
         "fixups": [
            {
               "DLL": "WaitForDebuggerFixup.DLL"
            }
         ]
      }
   ]
}
```

This config exempts the Launcher from any fixups but will cause any other process started in the container to use the fixup. If you have other PSF components that you need, they may also be added to the fixup list. If you have several processes but only want it to apply to the starting EXE, change the regex ".*" for the executable to be a regex match for your process name.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Chapter 7: AppxManifest Schema

In 3.9 Basic AppXManifest structure, the basic structure and most important elements of the AppXManifest file are shown. The syntax required and/or allowed for your AppXManifest.XML file is defined by a series of schemas.

Both AppX and MSIX use a base schema enhanced by sets of extension schemas that modify and extend the overall AppXManifest definition. These schema extensions cover a particular area, and we like to group schemas that cover the same areas into we call schema Sets.

Generally, there is an initial schema and the others in the schema set are extensions to that schema, as these schema sets have been extended over time to add more types of application features that may be supported.

A schema defines the rules for what is and is not valid in an XML file. This includes the names of elements and attributes and the hierarchy of where they belong in the XML. Additionally, the schema may place restrictions on how the XML is formed, including:

- The names of elements.
- The names of attributes of an element and whether the attribute is required or mandatory.
- The syntax of what is allowed as the value of an attribute. This may be specified as a list or by using regex patterns.
- The expected or allowed sub-elements of an element. When multiple sub-elements of the same name are allowed, there may be minimum/maximum limits on the number allowed.
- The syntax of what is allowed as the content of an element. This may be specified as a list or by using regex patterns.
- Possible restrictions on combinations of sub-elements under an element.
- Uniqueness restrictions, generally to ensure there is not an attribute value used in two different places.

The schemas used by MSIX can also include important comments that, while not affecting validation imposed by the schema files, inform you of additional restrictions that need to be met. There are also limitations imposed by the version of the operating system that the MSIX package is to be run on. As you learn about these schemas, you will see that over time a schema set has been extended many times. The newer extensions are add-ons that extend what is allowed, but they might require a newer operating system to be able to make use of the extended functionality. On a down-rev OS version, these new extensions should be silently ignored. The attributes of the TargetDeviceFamily child of the Dependencies element of the AppXManifest file (see 7.6.4.1 TargetDeviceFamily sub-element) might also affect what the application installation and/or runtime allows.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

If you use a vendor tool to create your AppXManifest for you, the details of schema-imposed syntax should be taken care of by that vendor. However, if you need to build your own AppXManifest file, you must adhere to the schemas in order to create a valid package. A validation check of the AppXManifest will be made during at least three phases:

- When running MakeAppx to build the package.
- When running SignTool to sign the package.
- When installing the package on an end-user system.

Individually, these validations might only validate a subset of the requirements imposed by the schema set, so being able to run MakeAppx does not form a certainty that your AppXManifest is correct.

7.1 Schemas supported

Microsoft does not generally publish the schema files for MSIX, but instead provides online documentation for them. The most common set applicable to MSIX is documented at this link: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/schema-root. But many of the schema files may be found installed on your system with Visual Studio and/or the Windows SDKs. They are also included, although not necessarily up to date, in GitHub projects like makemsix.

Developers working on projects for delivery of apps containing traditional Win32/DotNetFramework components to Windows 10 devices will need to use the Foundation, UAP, and RestrictedCapabilites schema sets, and possibly the Desktop, Com, WindowsCapabilities, and Virtualization extension schema sets. There also exist schema extension sets for IOT, Mobile, and Holographic that probably do not apply, and recently an extension set for Microsoft Teams was also introduced (we are not covering this set because of both the newness of these extensions and the factthat such applications would be new rather than conversions).

The table below provides an overview of the known schema files. The year listed is often used in the organization of folders containing the schema files when we find them; the year seems to represent the year they were first completed internally within Microsoft. Keep in mind that when additional schema extensions are added, many of these files (especially the AppXManifestTypes file) are also updated.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.1.1 Foundation schemas

The schemas in this set are needed for all Appx/MSIX packages.

Name	Year	Purpose
FoundationManifestSchema.xsd	2015	Base schema set required for any package.
FoundationManifestSchema_v2.xsd	2015	Minor extension that is rarely used.
AppXManifestTypes.xsd	2015	Used to define common element types and attribute value restrictions. Always needed.

The FoundationManifestSchema is the default schema in your project and is required. The AppXManifestTypes file defines details for complex and simple elements used throughout the schema sets. Both of these files are updated regularly.

7.1.2 UAP schemas

These schemas are common to application packages being deployed against any OS platform supporting UWP. Before MSIX, these extensions applied only to UWP, but these extensions are a basis for MSIX as well. Your package AppXManifest will reference a few of these, at the minimum.

As updates to schemas become defined, they are implemented into the OS over time. This OS implementation, however, is on a per-feature basis, so we cannot tie support of a schema file to an OS version. Microsoft will sometimes add language to the online documentation of an extension element to show required OS versions for that element. At other times, Microsoft will list a minimum build number against a feature.

Name	Year	Purpose
UapManifestSchema.xsd	2015	Used to define application interfaces that are generally supported across all platforms supporting universal apps, including Windows, IOS, and Android
UapManifestSchema_v2.xsd	2015	Additional elements and attributes
UapManifestSchema_v3.xsd	2015	Additional elements and attributes
UapManifestSchema_v4.xsd	2016	Additional elements and attributes
UapManifestSchema_v5.xsd	2017	Additional elements and attributes
UapManifestSchema_v6.xsd	2017	Additional elements and attributes

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

UapManifestSchema_v7.xsd	2018	Additional elements and attributes
UapManifestSchema_v8.xsd	2018	Additional elements and attributes
UapManifestSchema_v10.xsd	2019	Additional elements and attributes
UapManifestSchema_v11.xsd	2019	Additional elements and attributes
UapManifestSchema_v12.xsd	2020	Additional elements and attributes

7.1.3 Desktop schemas

These schemas are specific to Windows 10 and are not portable to the other operating systems. A majority of MSIX packages will include a reference to some of these.

Name	Year	Purpose
DesktopManifestSchema.xsd	2015	Used to define application interfaces that are specific to Microsoft Windows
DesktopManifestSchema_v2.xsd	2016	Additional elements and attributes
DesktopManifestSchema_v3.xsd	2017	Additional elements and attributes
DesktopManifestSchema_v4.xsd	2017	Additional elements and attributes
DesktopManifestSchema_v5.xsd	2018	Additional elements and attributes
DesktopManifestSchema_v6.xsd	2018	Additional elements and attributes
DesktopManifestSchema_v7.xsd	2020	Additional elements and attributes
DesktopManifestSchema_v8.xsd	2021	Additional elements and attributes
DesktopManifestSchema_v9.xsd	2021/H2	ClassicShellContextMenus

7.1.4 COM schemas

These schemas are specific to Windows 10 COM and are not portable to the other operating systems. Packages with COM components will need to reference these for reference by anything outside the package, especially the Explorer shell.

We would recommend continuing to include the old-style COM registration values in the application registry hive for internal access to the com-based components.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Name	Year	Purpose
ComManifestSchema.xsd	2015	Used to define interfaces to traditional COM components
ComManifestSchema_v2.xsd	2017	Additional elements and attributes
ComManifestSchema_v3.xsd	2019	Additional elements and attributes
ComManifestSchema_v4.xsd	2020	Additional elements and attributes

7.1.5 WindowsCapabilities schemas

These schemas are used to define "Capability" elements that inform the consumer about situations where this application might integrate with other applications and/or other application data. You are probably more familiar with this on phone apps that inform you that the app uses your contacts or camera. Modern Windows apps should also declare these capabilities.

All MSIX packages that are submitted to the Windows Store will undergo a verification that if the Windows Capabilities defined in these schemas are used, they are declared in the AppXManifest. Failure to do so will result in submission rejection.

For packages that do not go through the Windows Store certification process, there may also be additional runtime checks within the OS that would prevent the app from using this capability. For now, these restrictions seem to be implemented only in UWP application interfaces, so it is possible that traditional Win32 and Dot Net Framework-based apps packaged into MSIX would not need these at this time. Still, it is a best practice to declare these capabilities when used.

Name	Year	Purpose
WindowsCapabilitiesManifestSchema.xsd	2015	Used to define capabilities declarations.
WindowsCapabilitiesManifestSchema_v2.xsd	2015	Additional elements and attributes
WindowsCapabilitiesManifestSchema_v3.xsd	2016	Additional elements and attributes

7.1.6 RestrictedCapabilities schemas

These schemas are used to define "Capability" elements that, when declared in a package, are not automatically approved for use in apps to be published in the Windows Store. These elements are generally believed to modify the behavior of the installation or runtime container to perform riskier actions than originally envisioned for UWP apps. Some of these capabilities seem to be automatically

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT. revenera

approved when explained in the notes of the submission, but others are unlikely to gain approval without significant interaction.

All MSIX packages will need to include at least one of the restricted schema elements, as that is needed to extend the container to provide required support.

The presence of these elements may or may not be acceptable for Store apps, although often it just means that a manual approval is required. There is no restriction approval needed for MSIX packages delivered through other means, but there are cases where inclusion of certain restricted elements might cause a UAC prompt for credentials/elevation when installing the package.

Name	Year	Purpose
RestrictedCapabilitiesManifestSchema.xsd	2015	Used to define capabilities of the restricted set.
RestrictedCapabilitiesManifestSchema_v2.xsd	2015	Additional elements and attributes
RestrictedCapabilitiesManifestSchema_v3.xsd	2016	Additional elements and attributes
RestrictedCapabilitiesManifestSchema_v4.xsd	2017	Additional elements and attributes

7.1.7 Virtualization schemas

The purpose of defining this new schema set is not clear, but perhaps as we see additional extensions to this set it will become clearer.

Currently, this schema adds two new elements that are used to prevent virtualization. These elements allow the package to define Folders and Registry keys that should be treated as passed through the package to the underlying system. This might be useful when for example, the app defines a registry key in the application hive to define some things, but we want the app to see and write to the local user registry rather than the application hive.

Name	Year	Purpose
VirtualizationManifestSchema.xsd	2020	Used to define overrides for virtualization.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.1.8 CloudFiles schemas

These schemas seem rather specific to One Drive, but possibly the intent is to support any cloud-based file shares. These seem to be related to use of UWP APIs and would not be of interest to any app not using them.

Name	Year	Purpose
CloudManifestSchema.xsd	2019	Used to define interfaces for cloud-based file shares.

7.1.9 Preview schema sets

These schemas are mentioned here only for completeness. They should probably not be used unless required by Microsoft:

- PreviewManifestSchema_MsixAppCompatSchema.xsd
- PreviewManifestSchema_MsixAppCompatSchema_v3.xsd

7.1.10 Additional schema sets

These schema sets are generally not of interest for Windows apps. Including them in the AppXManifest, even as a reference to the schema only, will likely cause errors to be recorded in the event log when the application is installed, although the installation will not be interfered with.

Name	Year	Purpose
HolographicManifestSchema.xsd	2015	Seems to be for HoloLens-specific interfaces.
IotManifestSchema*.xsd	2015,17	Internet of Things
MobileManifestSchema.xsd	2015	Phone
ServerManifestSchema.xsd	2015	For Windows Server
XboxManifestSchema.xsd	2015	Xbox

7.1.11 Unofficial schema sets

Although not defined by the Microsoft schema for MSIX, it is common for tooling vendors to also include the BuildMetadata schema into the AppXManifest. This schema may be easily found in the SDK for Visual Studio. Generally, Microsoft tools will silently ignore the elements from this schema, but it does need to be properly referenced on the package element to be included.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.2 Examples of how to read a schema file

Even if you are familiar with reading an XML schema file to understand what is valid for the XML, Microsoft use of schemas for MSIX includes the use of many lesser-known features available to schema writers.

7.2.1 Types in schema

The XML schema syntax allows for an element to be defined as a simple type or complex type. Attributes may only be simple type.

Basically, a simple type means that it contains a single thing, while a complex type can be, well, complicated!

Often, an element or attribute will be defined in the schema with the definition for syntax, and restrictions are separated into type definitions found elsewhere. For example, in the FoundationManifestSchema, the “identity” element is defined as:

```
<xs:element name="Identity" type="CT_Identity"/>
```

By tradition, Microsoft will name the types starting with ST_ or CT_ to indicate that the underlying type will be simple or complex. In the example above, the definition of that type is located elsewhere in the same schema file:

```
<xs:complexType name="CT_Identity">
    <xs:attribute name="Name" type="t:ST_PackageName" use="required"/>
    <xs:attribute name="ProcessorArchitecture"
                  type="t:ST_Architecture_v2" use="optional"/>
    <xs:attribute name="Publisher" type="t:ST_Publisher_2010_v2"
                  use="required"/>
    <xs:attribute name="Version" type="t:ST_VersionQuad"
                  use="required"/>
    <xs:attribute name="ResourceId" type="t:ST_ResourceId"
                  use="optional"/>
</xs:complexType>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

You may notice that the value of type might also start with a namespace, as seen on the attributes above. This namespace indicates the schema file that would contain the type definition. The “t” namespace points to the AppXManifestTypes schema file, and this type definition is shown here:

```
<xs:simpleType Name="ST_PackageName">
     <xs:restriction base="ST_AsciiIdentifier">
          <xs:minLength value="3"/>
          <xs:maxLength value="50"/>
     </xs:restriction>
</xs:simpleType>
```

It is common for the restrictions to be based on another more common type, and then add additional restrictions, as is done in this case:

```
<xs:simpleType name="ST_AsciiIdentifier">
     <xs:restriction base="ST_NonEmptyString">
          <xs:pattern value="[-.A-Za-z0-9]+"/>
     </xs:restriction>
</xs:simpleType>
```

And the further restrictions are base defined as:

```
<xs:simpleType name="ST_NonEmptyString">
     <xs:restriction base="xs:string">
          <xs:minLength value="1"/>
          <xs:maxLength value="32767"/>
          <xs:pattern value="[^\s]|([^\s].*[^\s])"/>
     </xs:restriction>
</xs:simpleType>
```

The patterns used are standard ECMI (aka JavaScript) Regex patterns.

7.2.2 Abstraction, choice, and substitution group

Another schema trick commonly used in these schemas is abstraction, choice, and substitutionGroup. This allows the base schema to define an element or attribute but indicate that it might be replaced by the definition in additional schemas. This allows for the base schema file to be updated less frequently.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The definition of the Capabilities element in the FrameworkManifest schema provides a great example of this:

```
<xs:complexType name="CT_Capabilities">
      <xs:sequence>
            <xs:element ref="CapabilityChoice" minOccurs="0"
             maxOccurs="100"/>
            <xs:element ref="CustomCapabilityChoice" minOccurs="0"
             maxOccurs="1000"/>
            <xs:element ref="DeviceCapability" minOccurs="0"
             maxOccurs="1000"/>
      </xs:sequence>
      </xs:complexType>
            <xs:element name="CapabilityChoice" abstract="true"/>
            <xs:element name="Capability"
             substitutionGroup="CapabilityChoice">
                  <xs:complexType>
                        <xs:attribute name="Name"
                                      type="t:ST_Capability_Foundation"
                                      use="required"/>
                  </xs:complexType>
            </xs:element>
      </xs:complexType>
</xs:complexType>.15
```

Other referenced schema files may now add their own capability by referencing the CapabilityChoice abstraction. Here is the definition used in UapManifestSchema:

```
<xs:element name="Capability" substitutionGroup="f:CapabilityChoice">
     <xs:complexType>
           <xs:attribute name="Name" type="t:ST_Capability_Uap"
                         use="required"/>
     </xs:complexType>
</xs:element>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Here is an example of how an AppXManifest file might combine capabilities from different schemas:

```
<Package>
    …
    <Capabilities>
        <Capability Name=”internetClient” />
        <rescap:Capability Name=”runFullTrust”/>
        <rescap:Capability Name=”allowElevation” />
        <DeviceCapability Name=”webcam” />
    </Capabilities>
    …
</Package>
```

7.2.3 Schema conclusion

To understand how to write a valid AppXManifest file, it is necessary to read and understand the schema files that define the allowed and/or expected syntax. Microsoft uses some of the more obscure ways in which to write these schemas and get them to work together to form the overall AppXManifest schema, which can make understanding these files difficult. The online Microsoft documentation is often easier to understand, although guidance on when/how to use the elements is often missing.

In the rest of this chapter, we will provide an explanation of the elements that you will use, with links to the appropriate Microsoft documentation. The Microsoft documentation is good at defining syntaxes but often weak on meaning and intent; we will attempt to fill in some of those gaps.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.3 Identity Element

The Identity element is a top-level element (under Package) that contains a set of attributes that uniquely define this package. This element is required. Technically, the schema does not require this element for validation, but the package is useless without it.

7.3.1 Schema source

The element is defined in the FoundationManifestSchema, with help from AppXManifestTypes.

7.3.2 Microsoft documentation

The Identity element is documented at this link:
https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-identity

7.3.3 Element attributes

The element has five potential attributes, three of which are required.

Required attributes:

- **Name:** Often the value of this attribute will be in the form of “VendorName- Product” or “VendorName.Product.” Restrictions exist in what may be in a name. For example, the underscore and space characters are not allowed, and the name must not match a list of old DOS device ports (like “COM1”. See the schema or documentation on this element for details.
- **Publisher:** The value of this attribute will be a string that matches the Subject field of the certificate to be used to sign the package. Often this will be in the form of “CN=VendorName.”
- **Version:** The standard four-part version numeric/string “Major.Minor.Build.Revision.” Note that for packages to be uploaded to the Windows Store, the Revision field of this string must be set to 0, as they reserve the right to control that revision field. Packages delivered by other means do not have this limitation.

Optional attributes:

- **ProcessorArchitecture:** This string is used to define the type of processor supported. Although it's listed as optional, we highly recommend including this attribute. This is most often used when this package will be included in an msixbundle that contains different packages for different architectures. In that case the entire bundle is used, and the installer will select the most appropriate package from the bundle.
- **ResourceId:** Used when this is a Resource package for an application. An application package can have multiple resource packages that are used in different scenarios, and this ID would be matched up with the OS resource identifier to select the correct language/region.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.3.4 Sub-elements

None.

7.3.5 Implementation notes

The attributes, other than Version, are used to form a unique identifier for this package. The Version field is used by the installer to determine if this is a reinstallation attempt of a package already present or an upgrade/downgrade scenario. In the former case, the re-installation will be skipped, and in the latter case, the existing application will be replaced by this version. All user data associated with the existing application will be migrated.

As the package must be digitally signed, the signing requires an exact match in the string used in the Publisherattribute here when compared to the subject field of the code-signing certificate. As signing is a secondary step, if these do not match, you will create a valid MSIX package file, but the signing will fail.

7.3.6 Example

While not required, by tradition the identity element is usually placed as the first sub-element of a Package.

```
<Package …>
    <Identity Name=”Vendor-Application”
              Publisher=”CN=Vendor, OU=VenderOu”
              Version=”1.2.3.0” ProcessorArchitecture=”x64” />
    …
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.4 Properties element

The Properties element is used to contain information that affects how the package appears to the end-user. This element is required. Technically, the schema does not require this element for validation, but the package is useless without it.

7.4.1 Schema source

The element is defined in parts, in the following schema files:

- FoundationManifestSchema
- AppXManifestTypes
- UapManifestSchema
- UapManifestSchema_v6
- UapManifestSchema_v10
- DesktopManifestSchema_v6
- RestrictedCapabilitiesSchema_v6

7.4.2 Microsoft documentation

The Identity element is documented at this link:
https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-properties

7.4.3 Element attributes

None

7.4.4 Sub-elements

There are three required sub-elements, and the remaining sub-elements are optional. The schema technically supports multiple entries of the optional sub-elements, but this is not used in practice.

Required sub-elements:

- **DisplayName**: A "friendly" version of the PackageName attribute of the Identity element that is used whenever an end-user interfaces with the package as a whole; for example, the AppInstaller GUI displays this name when installing the package. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-displayname.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- **PublisherDisplayName**: A "friendly" version of the Publisher attribute of the Identity element that is used whenever an end-user interfaces with the package as a whole; for example, the AppInstaller GUI displays this name when installing the package. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-publisherdisplayname.

- **Logo**: Reference to a single png file (using a path relative to the root of the package) used whenever a graphic is needed to show with the package as a whole; for example, the AppInstaller GUI displays this logo when installing the package. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-logo.

Optional sub-elements:

- **Description**: A "friendly" description of the package. This field is not currently displayed by the AppInstaller GUI, but it should be considered that this field may be used in other situations or even in future versions of AppInstaller. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-description.

- **rescap6:ModificationPackage**: Marks this package as a modification to another package, also sometimes called optional packages. Modification packages act as an overlay onto the package they modify, both for file and registry values. Modification packages may not contain Application elements, including any way for the user to start an application running in the package (e.g., start menu, fta shell verb, uri protocol, or shell extension). The developer may use the modification package to contain a package plug-in; enterprise customers might use it to override configuration information. Microsoft has indicated that store apps are unlikely to have modification packages approved if submitted. This element is a Boolean that defaults to false when the element is not present. When set to true, the primary package will need to be added as a reference in the form of a MainPackageDependency element under Dependencies. In the future it is possible that this feature could become deprecated in favor of the Connected Containers feature. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-rescap6-modificationpackage.

- **Framework**: Contains the value "true" if this package is a framework package. A framework package is used as a dependency to be included in other packages that need it. Common examples of framework packages include versions of Microsoft.Net Framework or VC Runtimes. It is assumed that this value is false when the element is not specified, and Microsoft documentation indicates that you should not include the element to set it to false. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-framework.

- **ResourcePackage**: Contains the value "true" if this package is a resource package. A resource package is used as a supplement to be included in other packages that need it. Common examples of resource packages include language packs. It is assumed that this value is false when the element is not specified, and Microsoft documentation indicates that you should not include the element to

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

set it to false. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-resourcepackage.

- **uap:SupportedUsers**: This element may be used to indicate whether a package application is designed to support multiple users. The element value may be a string of either "single" or "multiple," and the documentation indicates that this element, when present, might be used at package installation time to determine eligibility. Presumably this refers to a Remote Desktop Server or Multi-User Windows 10 scenario. No information is available at this time about what would happen if the element were not included in those scenarios. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/ element-uap-supportedusers.

- **uap6:AllowExecution**: This element is also a Boolean, and the value of the element in this case is assumed to be true. Microsoft documentation says that this element indicates whether the contents of the package are allowed to execute. We assume you shouldn't set it. The scenario where this would be used is unknown. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-uap6-allowexecution.

- **desktop6:RegistryWriteVirtualization**: This element is a string set to either "enabled" or "disabled," and it defaults to "enabled" when the element is not present. The documentation indicates that this element is for use in certain games, and when disabled is specified, it allows other applications to read or write the same registry entries as your application. The documentation further states that the RestrictedCapability unvirtualizedResources must also be expressed in the package. This feature predates both the newer virtualization:RegistryWriteVirtualization element and the newer Connected Containers feature and possibly will be deprecated in the future. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-desktop6-registrywritevirtualization.

- **desktop6:FileSystemWriteVirtualization**: This element is a string set to either "enabled" or "disabled," and it defaults to "enabled" when the element is not present. The documentation indicates that this element is for use in certain games, and when disabled is specified, it allows other applications to read or write the same file system entries as your application. The documentation further states that the RestrictedCapability unvirtualizedResources must also be expressed in the package. This feature predates both the newer virtualization:FileSystemWriteVirtualization element and the Connected Containers feature and possibly will be deprecated in the future. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-desktop6-filesystemwritevirtualization.

- **virtualization:FileSystemWriteVirtualiation**: This element is defined to provide specific file directories in the package, presumably as a more granular approach that is provided for by the desktop6 equivalent. Microsoft has not documented this element at the time of this writing.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- **virtualization:RegistryWriteVirtualization**: This element is defined to provide specific file registry keys in the package, presumably as a more granular approach that is provided for by the desktop6 equivalent. Microsoft has not documented this element at the time of this writing.

- **uap10:AllowExternalContent**: A Boolean that is assumed to be false when not present. When this element is present and contains the value true, it indicates that this package is allowed to use content that is outside this package in the form of a sparse package. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-uap10-allowexternalcontent.

- Additional information on the use of sparse packages may be found here: https:/docs.microsoft.com/windows/apps/desktop/modernize/grant-identity-to-nonpackaged-apps.

- **uap10:PackageIntegrity**: This element contains a single sub-element that has a simple on/off attribute value (not quite a Boolean) that defaults to "off." (**NOTE**: value must be lowercase!) Microsoft already goes to great lengths to prevent the package contents from being modified after installation, and they already perform checks to detect any tampering whenever the applications in this package are used. This feature appears to extend that detection to perform restoration of the original package contents. Why this is set as a package property and not left to an external Group Policy setting is unclear. The Microsoft MSIX Packaging tool sets this element and value to "on," suggesting that they recommend it be used.

  This feature is not believed to affect any data modifications that the user makes and stores in his or her profile. Documentation reference for this element is https://docs.microsoft.com/uwp/schemas/appxpackage/uapmanifestschema/element-uap10-packageintegrity.

- **uap13:Auto Update**: This new element appears to allow the auto update mechanisms supported by the AppInstaller XML to be defined right in the package. To be used, however, the package would be required to know exactly where the updated package installer will be located. This element has not been documented at the time of this writing, so perhaps clarification is coming.

7.4.5 Implementation notes

Currently, the PublisherDisplayName is shown when the AppInstaller GUI is used to install the package. It may be visible in other tooling as well.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.4.6 Example

Although the order of elements under the Package element is not important, most packages will place the Properties element right after the Identity element. Sub-elements of Properties may appear in any order. An example of Properties is shown below:

```
<Package …>
     …
     <Properties>
           <DisplayName>A friendly name for the package</DisplayName>
           <PublisherDisplayName>A version of the Publisher name
            suitable for display to the end-user</PublisherDisplayName>
           <Logo>Assets/Logo.png</Logo>
           <uap10:PackageIntegrity>
                 <uap10: Content Enforcement=”on” />
           </uap10:PackageIntegrity>
     </Properties>
     …
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.5 Resources element

The Resources element in the AppXManifest.XML declares options included in the package that may be selected based on the environment of the deployed operating system and user. It the union of all languages, display scales, and DirectX Feature Levels for the resources.pri file that the package contains.

The resources.pri file is explained later in this section. Every package must have a resources.pri file containing at least one language. If the current language in use is not included in the Resources, a default language will be selected. Therefore, the Resources element is necessary in every package AppXManifest file, and despite the schema validation restrictions, it must have at least one Resource sub-element with a Language attribute.

7.5.1 Schema source

The element is defined in the following schema files:

- FoundationManifestSchema
- AppXManifestTypes
- UapManifestSchema

7.5.2 Microsoft documentation

The Resources element is documented at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-resources

7.5.3 Element attributes

In the schema, there are no attributes to the element defined. However, in the current Microsoft documentation, there is one optional attribute listed:

Default is documented to be a Boolean value (accepting “true” or “false”), but the documented description is that it is for “the name of the capability.”

As the purpose of this attribute is unclear and missing in the schema files, we suspect it is simply a documentation bug. We would recommend not using it.

7.5.4 Sub-elements

The Resources element may contain a sequence of up to 200 **Resource** sub-elements. The schema also includes a validator for uniqueness of the Resource sub-element attributes.

Each Resource element consists of three attributes, all optional:

- **Language:** The values of this attribute are typical language strings, such as “en-us.”

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

- **uap:Scale** This refers to the text-scaling setting of the user's desktop.
- **uap:DxFeatureLevel** Probably not needed anymore.

7.5.5 Implementation notes

Resources and Resource PRI background:

Resources, such as the graphics files placed in the Assets folder of your package, are compiled into a file (called resources.pri) that is packaged with the app and then is called upon by the Windows runtime to resolve resources when the app runs in a user context. This includes not only when the app is running but also other Windows runtime components, such as the start menu, reference your package.

A few points to note:

- Every package must declare at least one resource language.
- The scale attributes declare the different text scaling levels supported by the package. Like languages, there must be at least one, but best practices include all five levels. The PRI may then have file references for a given graphic image, such as the icon used in the start menu, for each of these text scaling levels. The user's current text scaling value would then be used to select which graphic to display. The idea is that the graphic files might contain digitized text, and you might want the text to be larger on a setup asking for 400x text scaling than 100x.
- The scale and DirectX feature-level attributes are independent of the resource language in use.
- The PRI may contain actual string resources, as well as an indexed set of file paths (e.g., assets) that refer to various files in the package.
- The PRI file is data-only and does not use the portable executable (PE) format. It is specifically designed to be data-only as the resource format for Windows. It replaces resources contained within DLLs in the Win32 app model when you have UWP code only.
- An MSIX package without UWP code will still need a resources.pri file with at least one language.

Please refer to https://docs.microsoft.com/en-us/previous-versions/windows/apps/jj552947(v=win.10) for more details.

7.5.6 Examples

7.5.6.1 How to create the resources.pri file for your package

A resources.pri file contains Package Resource Indexes. Almost every tooling vendor who announced MSIX support will build this file automatically, based on project configurations (part of existing Win32 setup or MSIX project) like visual assets and language settings.

Here are the ways to generate the PRI file manually using your own custom builds:

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- MakePRI.EXE tool from Win10 SDK: PRI files can be created and dumped with the tool MakePRI.EXE. This utility comes with Windows 10 SDK and is usually located under (C:\Program Files (x86)\Windows Kits\10\bin\x86) or in its respective dedicated SDK folder.

- Package resource indexing (PRI) APIs (recommended by Microsoft over MakePRI.EXE): A set of package resource indexing APIs (see https://docs.microsoft.com/en-us/windows/desktop/menurc/pri-indexing-reference) was introduced in the Windows SDK for Windows 10, version 1803. The APIs take the form of Win32 Windows APIs, which means that you have a few options for calling them. You can call them directly from a Win32 app, or you can call them via platform invoke from a .Net app or even from a UWP app.

  Please refer to the Microsoft documentation for sample code (https://docs.microsoft.com/en-us/windows/uwp/app-resources/pri-apis-scenario-1), additional information, and examples of how to consume PRI APIs.

7.5.6.2 How to view and/or extract the PRI file contents

The PRI file is a binary file, and its content can be dumped using the MakePRI.EXE tool from the SDK (as shown in the command line below) or using PRI APIs (mentioned in the previous section).

```
C:\Program Files (x86)\Windows Kits\10\bin\10.0.19631.0\MakePri.exe dump /if
“C:\MsixTest\resources.pri” /of “C:\MsixText\resources.pri.XML”
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.5.6.3 Example of resources.pri file content

Each PRI file contains a named collection of resources referred to as a resource map. When a PRI file from a package is loaded, the resource map name (as below) is verified to match the package identity name.

This example shows the basic form of the PRI file dumped out as XML:

```
<?XML version=”1.0” encoding=”UTF-8” standalone=”true”?>
<PriInfo>
     <ResourceMap primary=”true” version=”1.0” name=”Msix-Resources-
     Test">
          <Qualifiers>
               <Language>FR-CA,EN-US,JA-JP</Language>
               <Scale>100,125,150,200,400</Scale>
          </Qualifiers>
          <ResourceMapSubtree name=”Files”>
               …
          </ResourceMapSubtree>
          <ResourceMapSubtree name=”resources”>
               …
          </ResourceMapSubtree>
     </ResourceMap>
</PriInfo>
```

7.5.6.4 Example of resources.pri file with files and scaled assets

Here is a more complete example that shows how scaled asset files and localized resources are defined in a resources.pri file and referred from the AppXManifest.XML file.

Microsoft Store Apps should support displays of different resolutions of visual assets. Below are all the various types of visual assets that can be referred in the AppXManifest.XML file.

- Small Tile, Medium Tile, Wide Tile, Large Tile.
- app Icon.
- Splash Screen.
- Badge Logo.
- Package Logo.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Additionally, each of these asset types may have different asset files based on the text scaling in use on the customer's system.

The resources.pri file contains the entries for the above asset file types and corresponding scaled asset files in the ResourceMapSubtree element. Below is a typical example as seen in the MakePri dump:

```
<ResourceMapSubtree name="Files">
    <NamedResource name="Calc-AppList.png" url="ms-resource://Msix-
    Resources-Test/Files/Assets/Calc-AppList.png">
        <Candidate type="Path" isDefault="true" qualifiers="Scale-
        400">
            <Value>Assets\Calc-AppList.scale-400.png</Value>
        </Candidate>
        <Candidate type="Path" isDefault="true" qualifiers="Scale-200">
            <Value>Assets\Calc-AppList.scale-200.png</Value>
        </Candidate>
        <Candidate type="Path" isDefault="true" qualifiers="Scale-150">
            <Value>Assets\Calc-AppList.scale-150.png</Value>
        </Candidate>
        <Candidate type="Path" isDefault="true" qualifiers="Scale-125">
            <Value>Assets\Calc-AppList.scale-135.png</Value>
        </Candidate>
        <Candidate type="Path" isDefault="true" qualifiers="Scale-100">
            <Value>Assets\Calc-AppList.scale-100.png</Value>
        </Candidate>
    </NamedResource>
    <NamedResource name="CalcMedTile.png" url="ms-resource://Msix-
    Resources-Test/Files/Assets/Calc-MedTile.png">
        ...
    </NamedResource>
    <NamedResource name="StoreLogo.png" url="ms-resource://Msix-
    Resources-Test/Files/Assets/StoreLogo.png">
        ...
    </NamedResource>
</ResourceMapSubtree>
```

For more information on the topic of creating these scaled image files, refer to the article How do I create image scaling (https://docs.microsoft.com/en-us/windows/uwp/app-resources/images-tailored-for-scale-theme-contrast).

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.5.6.5 Example resource language (localization)

Language localization may also be used to impact references in the AppXManifest file itself, allowing for things like a language-specific installation experience.

In the AppXManifest.XML of the Resources package are localized string resources that in turn refer to parts of the resources.pri file. This is achieved by using the "ms-resource" keyword. Here is an example of a portion of the AppXManifest.XML file of the Resources package.

```
<Package …>
     ...
     <Properties>
          <DisplayName>ms-resource:Name</DisplayName>
          <PublisherDisplayName>ms-resource:Publisher
          </PublisherDisplayName>
          <Logo>Assets\StoreLogo.png</Logo>
     </Properties>
     <Resources>
          <Resource Language=”en-US”/>
          <Resource Language=”ja-JP”/>
          <Resource Language=”fr-CA”/>
     </Resources>
     …
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The string resources referred to above in the AppXManifest.XML file are defined in the resources.pri file as shown below. Note that the contents below are extracted from the resources.pri file using the makepri.EXE utility.

```
<ResourceMapSubtree name=”resources”>
     <NamedResource name=”Name” url=”ms-resource://Msix-Resources-
      Test/Files/Assets/Name”>
           <Candidate type=”String” qualifiers=”Language-JA-
           JP”>
                <Value>Msix Resource Test in Japanese
                </Value>
           </Candidate>
           <Candidate type=” String” qualifiers=”Language-FR-CA”>
                <Value>French Canadian version of Msix Resource
                Test</Value>
           </Candidate>
           <Candidate type=” String” qualifiers=”Language-EN-US”>
                <Value> Msix Resource Test </Value>
           </Candidate>
     </NamedResource>
     <NamedResource name=”Publisher” url=”ms-resource://Msix-Resources-
      Test/Files/Assets/Publisher”>
           <Candidate type=”String” qualifiers=”Language-JA-JP”>
                <Value>Vendor name in Japanese</Value>
           </Candidate>
           <Candidate type=” String” qualifiers=”Language-FR-CA”>
                <Value>Votre nom d’enterprise</Value>
           </Candidate>
           <Candidate type=” String” qualifiers=”Language-EN-US”>
                <Value> The greatest vendor name ever</Value>
           </Candidate>
     </NamedResource>
</ResourceMapSubtree>
```

For more detail, including a more detailed example than this, see the article "How the Resource Management System chooses resource candidates based on a particular context," at https://docs.microsoft.com/en-us/previous-versions/windows/apps/jj552943(v=win.10).

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.6 Dependencies element

The Dependencies element of the Package is used to contain a set of sub-elements that are external packages that this package depends upon.

It is the responsibility of the implementor (the person responsible for installing your package) to ensure that these dependencies are met, although in certain customer-based situations this might occur through an automatic trigger of installing your package.

7.6.1 Schema source

The Dependencies element is defined in the FoundationManifestSchema, with help from AppXManifestTypes and optionally versions 3, 4, 5, 6, 7, 10, and 13 of the UAP schema set.

7.6.2 Microsoft documentation

The documentation page for the Dependencies element is found at this link: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-dependencies.

7.6.3 Element attributes

None.

7.6.4 Sub-elements

The Dependencies element supports a sequence of sub-elements of different types. According to the schema, none are required; however, in practice there should be at least one TargetDeviceFamily sub-element present.

7.6.4.1 TargetDeviceFamily sub-element

The **TargetDeviceFamily** sub-element is used by AppInstaller to determine if this package is suitable with the device it is being installed on. As long as the requirements of one of the TargetDeviceFamily elements are met, the package may be installed.

This sub-element is documented here:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-targetdevicefamily.

The sub-element has three attributes, all required. They are:

- **Name**: For Win32 and Dot Net Framework applications running under MSIX, this will be the string “Windows.Desktop.”

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- **MinVersion**, a four-part string representing the minimum OS version, such as 10.0.19041.1. You should set this value to the minimum version for the API set your code needs or minimum version needed for the MSIX schema features you use (whichever is higher). If you set it to a high build number, you may prevent certain customers from installing your application. And if you set it too low things might not work.

- **MaxVersionTested**. This item shouldn't prevent installation but might affect feature availability. For example, when Microsoft added changes in VFS layering in the runtime to support Modification Packages, the layering changes only worked if the MaxVersionTested attribute was at or above the OS build when the changes were released. You should normally set this to the highest version known (and, of course, test it).

Wikipedia maintains a document showing known Windows build numbers here: https://en.wikipedia.org/wiki/Windows_10_version_history.

Below is an example of the TargetDeviceFamily in the AppXManifest.XML file:

```
<Package …>
     …
     <Dependencies>
          <TargetDeviceFamily Name=”Windows.Desktop”
           MinVersion=”10.0.16299.0” MaxVersionTested=“10.0.19041.1”/>
          …
     </Dependencies>
     …
</Package> \
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.6.4.2 PackageDependency sub-element

The PackageDependency sub-element defines a prerequisite package on which your MSIX application is dependent. These dependencies must be met before your main application package can be installed.

This sub-element is documented here: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-packagedependency

Below is an example dependency in an AppXManifest.XML file for the packaged version of a VC Runtime:

```
<Package …>
    …
    <Dependencies>
        < TargetDeviceFamily …/>
        <PackageDependency Name=”Microsoft.VCLib.140.00.UWPDesktop”
        MinVersion=”14.0.24217.0” Publisher=“CN=Microsoft
        Corporation, O=Microsoft Corporation, L=Redmond S=Washington,
        C=US”/>
        …
    </Dependencies>
    …
</Package>
```

Traditionally in MSIs, application dependencies (like visual studio runtimes and .Net framework) are installed by bootstrapper executables provided by several independent software vendors. The application includes a copy of the dependency installer, and if it detects it is not installed yet, it will unpack and install the dependency automatically.

Declaring the dependency in this way is similar, but your package does not include the dependent installer, and listing it does not necessarily trigger the install.

Instead, you might consider that with MSIX you can simply include the needed runtime components within your package as files. While your package will be larger than the method above, you are not adding size to your installer over the MSI you use today with the dependency. Upon installation of your package, you are sure to have the required components directly inside your package without intervention. Additionally, because of single-instance storage, if multiple packages install the same runtime, only one copy is actually stored on the end-user's system.

7.6.4.3 MainPackageDependency sub-element

The **MainPackageDependency** sub-element helps create a modification package that stores customizations to the main package. These entries indicate that this package is dependent on another package that is a regular package, not a framework or resource package.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The FoundationManifest supports two flavors for abstracted choices:

- MainPackageDependencyChoice from Uap3
- MainPackageDependencyChoice from Uap4

The Uap3 form allows only a single MainPackageDependency to be specified, and it may only be referenced by a package name.

The Uap4 form supports multiple MainPackageDependency entries to be specified, and each may add an optional Publisher field to the package name in the attributes.

Modification package can be closely related to a Transform (.mst) concept for an MSI package. While transform files contain a set of instructions that impact the actions performed by the MSI setup, MSIX Modification Packages are applied as an overlay of an MSIX package not at installation time, but upon execution of the application. The key feature of a modification package is that it is not tied to a specific version of its main MSIX package, which gives you an advantage in being able to createa modification package once and use it for all the current and future versions of its main MSIX packages. Moreover, it can be applied or removed at any point in time, and changes to the application are visible only when the app is launched. As with a transform file, modification packages cannot be installed on a machine independently. The main MSIX package must be already installed on a machine to install its modification package on the same machine to apply customizations.

See Microsoft documentation for more information on modification packages:
https://docs.microsoft.com/en-us/windows/msix/modification-packages

Below is an example of a modification package in an AppXManifest.XML file that has customizations for the main app package called Main.app:

```
<Package …>
     …….
     <Properties>
          …
          <rescap6:ModificationPackage>true</rescap6:ModificationPackage>
     </Properties>
     <Dependencies>
          < uap4:MainPackageDependency Name=”Main.app”
          Publisher=“CN=Microsoft Corporation, O=Microsoft Corporation,
          L=Redmond S=Washington, C=US”/>
          …
     </Dependencies>
…
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Note that AppXManifest must include the rescap6:ModificationPackage sub-element to ensure that modification packages work in version 1903 or later. If you have the modification package created before version 1903, it is necessary to edit the AppXManifest file to update the MaxVersionTested attribute to 10.0.18362.0.

7.6.4.4 DriverDependency sub-element from Uap5

Uap5:DriverDependency is defined in the Uap5 schema. This element may list several DriverConstraint elements, which can specify the needed device driver via Name, optionally with MinVersion, and MinDate.

7.6.4.5 OSPackageDependency sub-element from Uap7

Uap7:OSPackageDependency is defined in the Uap7 schema. This element identifies OS requirements by name and version.

The relationship between this element, the TargetDeviceFamily element, and the uap10:HostRuntimeDependency element, is not clear at this time.

7.6.4.6 HostRuntimeDependency sub-element from Uap10

Uap10:HostRuntimeDependency is defined in the Uap10 schema. This element allows for specification of the dependent package by a combination of Name/Publisher/MinVersion.

The relationship between this element and the TargetDeviceFamily element, uap7:OSPackageDependency, and this one is not clear at this time.

7.6.5 Implementation Notes

7.6.5.1 Installing Dependencies

As noted previously, the declaration of a dependency in a package will generally prevent the package from installing when the dependency is not already installed at the time of installation of this package. It is assumed that external measures, including deployment tooling, will take care of this in advance.

Dependencies available from the Windows Store
One exception is when the package installed from the Microsoft Store and the dependency packages defined in the package AppXManifest are also available from the Windows Store. In this case, they also get installed automatically along with the actual MSIX package.

DriverDependency Installation via Windows Update
Another exception to this prevention is believed to be the DriverDependency. If this package is deployed by the Microsoft store and it contains a driver dependency, the installation of this package is allowed, and the installation of the driver dependency will be triggered in the background via Windows Update. We have heard that this may also be done with WSUS distribution of the driver, but we would expect

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

that in an enterprise deployment scenario, the driver would have already been assigned and installed in advance anyway.

AppInstaller File option

Microsoft expanded the capabilities of the AppInstaller File (https://docs.microsoft.com/en-us/windows/msix/app-installer/app-installer-file-overview), an XML file placed outside the MSIX package. This file could be used to install the primary app packages and its dependencies from remote sources, like file shares and websites, when the store option is blocked.

Additionally, this form of declaration supports the concept of optional dependencies, where the dependency will be used if present but does not block the installation.

Package dependencies in the AppInstaller file can be declared using the dependencies section the URI attribute referring to the package deployment path with App package definitions are created as .XML text files with an .appinstaller extension. The files can be created with a text editor or Visual Studio or they can be generated with third-party vendor tools.

Please refer to 3.7 AppInstaller (XML) file for more details on how to configure the app installer file to refer to dependencies.

Include common dependencies from Microsoft redistributable sources

Sometimes the customer may need to deploy the package in an environment where internet access is not permitted or is restricted (high-security zones), or the user may need to perform internal testing by sideloading option.

Visual Studio 2019 and the Windows SDK include the latest framework packages for Visual C++ and .Net Runtime in the following folders:

C:\Program Files (x86)\Microsoft SDKs\Windows Kits\10\ExtensionSDKs\Microsoft.VCLibs.Desktop\

You may want to create a Bundle package that includes your package and the corresponding dependencies like language-specific assets, varying image-scale assets, or resources that apply to specific device architectures and have the same identity.

Bootstrapper executables

The bootstrapper executables provided by third-party vendors can be used to bundle both main and dependency packages together and distribute them as a single setup. This setup performs the silent installation of both main and dependency packages.

This is basically the same technology used for years for setup exefiles, simply replacing the MSI with the MSIX and the offline VC Runtime installers with the dependency package.

While this remains a viable option, it does blunt some of the value of MSIX clean installations, and we suspect that Microsoft would encourage you not to use this option and go for the bundle instead.

PowerShell scripts

To be very clear: Microsoft doesn't want you to use this method to deploy dependencies either. Scripts

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

provide a flexibility to customize an application dynamically to the user's environment after it is packaged using MSIX, and if you don't mind an elevation prompt, they can natively install dependencies.

You can use the Package Support Framework (PSF) to run one PowerShell script with PowerShell cmdlets (like Add-AppPackage) to install dependency packages. Note that PSF scripts can be configured to run once only on the first app launch and without showing the PowerShell window. If used to install a dependency, the script should have some checking mechanism to see if the dependency is already installed on the user's machine by some other package.

Custom Install Actions

As described in 4.15 Custom install actions, there is a possibility of using the new Custom Install Actions element of the AppXManifest file.

7.6.5.2 Dependency package types

The dependency packages themselves in some cases are regular packages, but more often they are either framework or resource-only packages. For example, apps developed using C# or VB will often require the .Net Runtime and .Net framework packages; likewise, C++ applications generally require the VC Runtime Libraries.

Microsoft references regarding these two resource package types may be found at these links:

- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-framework
- https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-resourcepackage

By default, a package cannot take a PackageDependency on another package if the dependency package is not declared to be a framework or resource package.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Below is an example of a package AppXManifest file that declares itself to be a framework package:

```
<Package …>
     …
     <Identity Name=”Microsoft.VCLib.140.00.UWPDesktop”
               ProcessorArchitecture=”x86”
               Publisher=“CN=Microsoft Corporation, O=Microsoft
     Corporation, L=Redmond S=Washington, C=US”
               Version=”14.0.24217.0” />
     <Properties>
          <Framework>true</Framework>
          <DisplayName>Microsoft Visual C++ 2015 UWP Desktop Runtime
          Package </DisplayName>
          …
     </Properties>
     …
</Package>
```

On the other hand, Resource packages are asset-only packages that allow your app to adapt to multiple display sizes and system languages. This means that the AppXManifest does not attempt to expose any external program access points, including shortcuts, file associations, protocol handlers, and COM-based interface points.

Although an app package can contain several resources, the OS will only download the relevant resources per user device, saving bandwidth and disk space. If the user adds a new language to the list of OS languages in the region and language settings or changes the display configuration across an automatic store update, the OS will fetch applicable resource packages for all installed apps on the device. Declarations for Resource packages was discussed in 7.5 Resources element.

7.6.6 Example

See 7.6.5 Implementation Notes for example.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.7 Capabilities element

Capabilities are defined in the package AppXManifest file under the Package element, used by both the UWP (Universal Windows Platform) apps as well the desktops apps in the MSIX or Appx package.

Modern applications are expected to declare what hardware and software they interact with. Most of us experienced this first with our phone applications, but UWP and MSIX applications now embrace this concept as well.

The purpose of this is to inform the person installing the application about these interactions. With hardware interactions, an example might be to declare that the app makes use of a camera, with software to access your contacts.

The Microsoft AppInstaller GUI lists the capabilities declared in the AppXManifest file when the application is requested to be installed, allowing for a consented acceptance. There is no option to allow or disallow certain capabilities in the installation; it is a simple notification.

To check the app permissions after installation, the end-user may go to the Windows Settings page for Apps, which will display the capability declarations for the app.

Applications submitted to the Microsoft Store are subject to a verification by Microsoft that the application does not use any undeclared capabilities. Failing this verification will prevent Microsoft from accepting the application.

But even outside store distribution, it would be a very bad practice to not declare capabilities used by your application. Security-conscious enterprise and individual consumers will notice you use undeclared capabilities, and they are genuinely concerned when they find an app cheating.

Additionally, the Windows runtime also understands the application features by checking the capabilities, in case the capability that is required for the application function is not defined in the AppXManifest file, which means the related API will fail with an unauthorized access exception when the application is executed.

Package capabilities are sometimes referred to as application permissions because the capability entry allows access to the system resources. Later, the application permission can be changed in the application settings page.

Additional information on capabilities use and requirements may be found in 7.5.5 Implementation notes.

7.7.1 Schema source

The element is defined in the FoundationManifestSchema with help from AppXManifestTypes, with additional capabilities added in Uap, Uap 2-4 and 6-7 and 11, Wincap, Rescap, Iot, and Mobile Schemas.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The FoundationManifestSchema_v2 may also be added in the AppXManifest to extend the number of device capabilities supported.

7.7.2 Microsoft documentation

The Microsoft documentation page for the Capabilities element itself is https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-capabilities.

Better information on using Capabilities is at this Microsoft document: https://docs.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations.

7.7.3 Element attributes

None.

7.7.4 Sub-elements

The Capabilities sub-element is a generic array of sub-elements of the three different element types (independent of the different schemas):

Capability. This is the most common form of capabilities.

CustomCapability. Primarily used by Microsoft partners as directed by Microsoft.

DeviceCapability. Declares devices that the app utilizes. This is needed for UWP code that checks for the capability declaration for the UWP API to access the device. Currently this is not required for MSIX packages, but it should be included by vendors to be good citizens.

7.7.4.1 Capability sub-element

This element will contain a required attribute called “Name” and no sub-elements. The values supported come from many schemas but are ultimately defined in the ManifestTypes schema.

Microsoft does support chaining many capabilities into the same Capability element value or the Name attribute (as long as each comes from the same schema file) fusing the form:

```
<Package …>
    …
    <Capabilities>
        <Capability Name= “cap1” | “cap2” | ”cap3” />
        <uap:Capability Name=”cap4” | “cap5” />
        …
    </Capabilities>
</Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

However, the general practice by vendor tooling seems to only name a single capability per element.

These names represent the capability, but be aware that different words might be used when presenting the capability to the user.

The following table lists the Capability Names currently supported:

Schema	Name
Foundation	internetClient internetClientServer privateNetworkClientServer allJoyn codeGeneration
Uap	documentsLibrary picturesLibrary videosLibrary musicLibrary enterpriseAuthentication sharedUserCertificates userAccountInformation removableStorage appointments contacts phoneCall blockedChatMessages objects3D voipCall chat
Uap_2	phoneCallHistoryPublic spatialPerception
Uap_3	backgroundMediaPlayback userNotificationListener remoteSystem
Uap_4	offlineMapsManagement userDataTasks
Uap_6	graphicsCapture
Uap_7	globalMediaControl gazeInput
Uap_11	graphicsCaptureProgrammatic graphicsCaptureWithoutBorder
Mobile	recordedCallsFolder
IOT	systemManagement lowLevelDevices
Wincap	[Special strings]
Rescap	[Special strings]

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The Wincap and Rescap namespace attribute values are not defined in the schema (although there is a naming pattern that is validated) but are based on understood values, most of which are not documented and were added according to a private agreement between Microsoft and their developer partners that required them. Some have some documentation on this page: https://docs.microsoft.com/en-us/windows/uwp/packaging/app-capability-declarations

There are, however, a few names under the rescap:Capability attribute that are more generally required for MSIX (when used) and may or may not have official documentation:

Schema	Name	Purpose
Rescap	runFullTrust	Opens the container to the "full trust model" needed for MSIX (as opposed to the "app model" used by UWP apps). This is what defines the package as "MSIX" and allows Win32 and DotNetFramework code to run in your package. It is always required for MSIX.
	packagedServices	Triggers an external creation of a Windows service when this package is installed. Include if you have Windows services in your package.
	localSystemServices	Include when a Windows service is designed to run under the Local System account.

For more information on these three rescap items, see the Implementation Notes section below.

7.7.4.2 Custom capability sub-element

The purpose of this element seems to be to help software vendors work around issues with customized patches from Microsoft. We do not expect most developers to ever use it.

This element will contain a required attribute called "Name" and no sub-elements. While the schema imposes restrictions on the form of the value, the value is per private arrangement with Microsoft. Generally, the value is in the form "company.capabilityname_publisherid."

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.7.4.3 Device capability sub-element

The purpose of this element is to declare devices that the application needs to access. Many system devices are allowed to work without declaration; however, the following list items are known to have a requirement for declaration:

- USB
- humaninterfacedevice
- bluetooth.genericAttributeProfile
- bluetooth.rfcomm

This element will contain a required attribute called “Name” and a sequence of two different sub-elements.

The name is either an alphanumeric string or a GUID. The sub-elements of DeviceCapability are elements called Device and AdditionalDeviceChoice.

Please see this Microsoft document for more detail on DeviceCapability: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-devicecapability

7.7.5 Implementation notes

7.7.5.1 Capabilities and Microsoft store publishing

The Microsoft Store lists all the declared capabilities of an app in the product page when the App is selected in the Store. This allows the user to choose not to install the App if the user does not want the App to use the resources declared. Thus, you only should declare the capabilities that are required for the App. Keep in mind that submission testing by Microsoft will detect if your application runtime uses these capabilities, so you need to specify all the capabilities used by the App while submitting to the store in the submission form.

Capabilities from the Restricted schema set need to go through an additional submission process while submitting to the Store. In your submission notes you need to justify the usage of restricted capabilities in the package. Common restricted capabilities are routinely approved when you ask, but this does involve a human process that might delay acceptance of the submission.

7.7.5.2 RunFullTrust capability

The **runFullTrust** restricted capability is what separates an AppX package from an MSIX package. It allows apps to run at the full trust permission level on the user's machine rather than in the AppX/UWP App Model level. This capability is required to use the FullTrustProcessLauncher API. This capability is required for a desktop application that is delivered as an appx or msix package. The runFullTrust restricted capability allows the desktop apps to access all the available Windows resources so that most of the Win32 code will work just as it is.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Don’t be fooled by the name; runFullTrust only allows the application inside the container to perform actions that a native Win32 or .Net Framework app could have done. If the Win32 app needed elevation for permissions, so will your MSIX app.

The runFullTrust capability must be declared in containerized apps that use Win32 or .Net Framework code (MSIX App packages). In addition to declaring the capability, Applications elements within the package will need to request the **Windows.FullTrustApplication** Application EntryPoint in the AppXManifest.

7.7.5.3 Capabilities required for UWP code

Capabilities are required to specify for the UWP-based app functionality during runtime, just aswebcam device capability is required to specify whether the App wants to access the camera to capture photos.

You may also require the use of Device capabilities declared in the package to use certain UWP APIs, or else the App will throw an error during runtime.

MSIX packages accessing similar functionality by Win32 or Dot Net Framework APIs do not appear to be subject to this restriction, although you should strive to declare what you use anyway.

7.7.5.4 Capabilities required for desktop applications

UWP Apps include runtime API checks against the capability declaration before accessing the system resources; however, the Win32 desktop apps developed in the earlier technologies do not check the capabilities before accessing the system resources.

Allow Elevation
The allowElevation restricted capability allows Apps to keep existing desktop functionality that requires auto elevation on launch or during an app's lifetime. If the desktop app is configured to run with elevated privileges in the application executable’s manifest file, then it must be required to specify allowElevation capability; otherwise, the application launch will fail. Enabling the allowElevation capability alone will not make the App execute in the Elevated mode; it's also necessary to configure in the executable manifest, or else the capability will be ignored, and the App will launch without elevation. This capability is supported starting in Windows 10 version 1809, to declare in the AppXManifest file to support elevation at startup.

At an undocumented later time, possibly in the 2004 version of the OS, the check that prevented elevation without this capability declaration was removed. Whether this is a bug or by design, we recommend including the capability in all packages that require it. Microsoft Store certification continues to check for this capability.

Packaged Services
Support for packaged services was added in the 2004H1 OS. Although MSIX packages are generally installable by standard users without elevation, those with the packagedServices capability declared will require administrative rights to install the package itself.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The inclusion of traditional Windows services (Package element with extensions with the windows.service category) requires the packagedServices Capability and often also the Local System Services capabilities to function.

Packaged Services: The packagedServices restricted capability is required if the package is configured to install Windows services along with the App, by declaring the windows.service extension in its package AppXManifest. These services can be configured to run under the Local Service, Network Service, or Local System accounts. Local Service and Network Service services only require the packagedServices capability. Services that you want to run under the Local System account require both the packagedServices and localSystemServices capabilities.

Local System Services: The localSystemServices restricted capability is required if the package is configured to install one or more Local System services along with the App (that is, the App can declare the StartAccount for the services to be LocalSystem). This scenario also requires the packaged Services capability.

Other Devices

AutoPlay Device and AutoPlay Content extension require Removable Storage capability to access the files on the removable storage.

The Windows Runtime API also checks for capability declaration while executing the service, and if the capability is not defined, the API throws an error. Similarly, Bluetooth capability is required for the APIs in the Windows.Devices.Bluetooth name space, even if accessed from DotNetFramework code.

Example

```
<Package …>
     …
     <Capabilities>
          <Capability Name=”internetClient” />
          <rescap:Capability Name=”runFullTrust”/>
          <rescap:Capability Name=”allowElevation” />
          <DeviceCapability Name=”webcam” />
     </Capabilities>
     …
</Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.8 Package extensions element

NOTE: There are two forms of the “Extensions” and “Extension” elements, one that is a child of the “Package” element and another a child of the “Application” element. This section covers the version of extensions that lives under the Package element.

The AppXManifest schemas have two elements named “Extensions”: one that is a child of an Application element and another that is a child of the Package element. Application extensions are to add functionality for a particular app, defined under the Application element in the AppXManifest. Package extensions are the extensibility points to define the configuration for the package. This section covers the case of the Package-level Extensions element.

Package Extensions are the extensibility points by which the package can add functionalities in a manner defined by the operating system. An extensibility point is a location where an app can register to execute code or use resources of the current package. Package Extensions are declared in the AppXManifest file under the Package/Extensions element with a specific extension category. The schema for each extensibility point is category-specific, and the configuration and the information required for each extension to register with the system vary based on the extension category.

MSIX packages support different types of Package extensions, such as to define the InProcessServer, OutProcessServer, Certificates, FirewallRules, or ProxyStub.

In the traditional Windows installer, all these configurations are done in the registry during the installation. Windows installer creates the necessary keys in the registry based on the information stored in the Windows installer tables, as the ProxyStub interface information will be stored in the Registry table and will update the target machine registry during the installation.

7.8.1 Schema source

The Package-level Extensions element is defined in the FoundationManifestSchema, with help from AppXManifestTypes and, optionally, some of the following schemas for the child elements:

- Com and COM 2 and 4
- Desktop 2, 6, 7, and 8
- Rescap 4
- Wincap 3
- Uap 6,7, 8, 10, and 11, 12

7.8.2 Microsoft documentation

The documentation page for the Package Extensions element is https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-extensions.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.8.3 Element attributes

None.

7.8.4 Sub-elements

The package-level Extensions element may contain a sequence of Extension elements. Each Extension element will have a parameter named Category, which defines the content form allowed for that extension type.

The following package-level extension categories exist in the schema sets. Links to the specific page of the Microsoft documentation are given when they exist:

Schema	Category	Microsoft Documentation
Com	windows.comServer	com:ComServer
	windows. comInterface	com:ComInterface
Com2	windows.comServer	com:ComServer
	windows.comInterface	com:ComInterface
Com4	windows.comServer	com:ComServer
	windows.comInterface	com:ComInterface
Desktop2	windows.firewallRules	desktop2:FirewallRules
	windows.desktopEvent Logging	desktop2:DesktopEventLogging
Desktop6	windows.mutablePackageDirectories	desktop6:MutablePackageDirectories
	windows.customInstall	desktop6:CustomInstall
Desktop7	windows.shadowCopyExcludeFiles	None
	windows.errorReporting	None
Desktop8	windows.mutablePackageDirectories	None
	windows.userMutablePackageDirectories	None
	windows.eventTracing	None
Desktop9	windows.fileExplorerClassicContextMenuHandler	None, nor a schema file

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Rescap4	windows.primaryInteropAssemblies	None
	windows.classicAppCompatKeys	rescap4:ClassicAppCompatKeys
Uap6	windows.loaderSearchPathOverride	uap6:LoaderSearchPathOverride
Uap7	windows.sharedFonts	uap7:SharedFonts
	windows.enterpriseDataProtection	uap7:EnterpriseDataProtection
Uap8	windows.dataProtection	None
Uap10	windows.hostRuntime	uap10:HostRuntime
	windows.installedlocationVirtualization	uap10:InstalledLocationVirtualization
	windows.mediaContentDecryptionModule	None
WinCap3	windows.internal.*	None

It is worth noting that some of these categories are not documented and/or the documentation discourages use of them.

7.8.5 Implementation notes

Care must be taken when reading the schema, as the element called Extension includes both Application-and Package-level extensions. The overall schema sets generally define which categories of Extension belong to the Package versus Application level through the use of an abstract name of "ExtensionChoice," "PackageExtensionChoice" (for package level), or "ApplicationExtensionChoice." Unfortunately, the ExtensionChoice may combine categories for application-level extensions with package-level ones. When this occurs, the Extension technically may reside in either location for schema validation purposes, but you must look to the comments in the schema to determine proper placement if you plan to have the functionality you expect. When in doubt, you should look for the simpleType definition in the AppxManifestTypes schema that is referenced by the schema in question.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

For example, the Desktop2 schema uses the simpleType ST_ExtensionCategory_Desktop2, which is defined (currently) as:

```
<xs:simpleType name="ST_ExtensionCategory_Desktop2">
    <xs:restriction base="xs:string">

          <!-- Application Extensions -->
          <xs:enumeration value="windows.appPrinter"/>
          <xs:enumeration value="windows.searchFilterHandler"/>
          <xs:enumeration value="windows.searchPropertyHandler"/>
          <xs:enumeration value="windows.mailProvider"/>

          <!-- Package Extensions -->
          <xs:enumeration value="windows.firewallRules"/>
          <xs:enumeration value="windows.desktopEventLogging"/>
    </xs:restriction>
</xs:simpleType>
```

Thus, the comments indicate that only firewallRules and desktopEventLogging categories may be used as a package-level extension. Note there are cases in the schema sets, such as ComInterface, where the category is legal under either Package or Application; you must read through the documentation clearly to determine which is the correct placement for your usage.

Finally, it should be noted that the schemas also place many unique restrictions that can trip you up. We have seen instances where even Microsoft's own software adds COM components under an application level, causing a validation failure due to a repeated COM GUID when the same component was registered to another application in the package. When this occurs, moving the definition to the Package level might be appropriate.

7.8.6 Examples

The four most important of the package-level Extensions categories are covered in detail in the sections that follow.

7.8.6.1 In-Process COM server

An In-Process Server component is a COM-based DLL that is loaded directly into the memory space of the calling code.

The extension for an In-Process Server registers the ActivatableClassId and its implementation DLL when the package is installed, so when the application attempts to activate the class, the system can find it and load the DLL in process. This extension allows developers to easily share code between Win32 apps (inside or outside the container) and UWP apps running in the app container that can be loaded in-proc.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT. revenera

Existing self-registration code in the COM component is no longer required, but it will also do no harm.

The In-Process Server extension can be defined with the extension category of type "windows.activatableClass.inProcessServer." The app then includes a dynamic link library (DLL) that exposes one or more activatable classes.

```
<Package ...>
    …
    <Extensions>
        <Extension Category="windows.activatableClass.inProcessServer">
            <InProcessServer>
                <Path>MathUtility.dll</Path>
                <ActivatableClass ActivatableClassId="MathFunctions
                        .SimpleMath" ThreadingModel="both" />
            </InProcessServer>
        </Extension>
    </Extensions>
</Package>
```

Path specifies the path to the DLL, and the ActivatableClass defines the runtime classes and the threading model supported by the runtime class.

7.8.6.2 Out-of-Process server

Like the In-Process Server extension, the Out-of -Process Server extension also registers the ActivatableClassId and its implementation EXE in the package, so when the application attempts to activate the class, the system can find it and execute the EXE as out of process. This extension allows developers to easily share code between Win32 apps and UWP apps running in the app containerthat can be executed as an Out-of-Proc server.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Out-of-Process Server extension can be defined with the extension category of type "**windows.activatableClass.outOfProcessServer.**" The app uses an executable (EXE that exposes one or more activatable classes.

```
<Package ...>
    …
    <Extensions>
          <Extension Category="windows.activatableClass.
          outOfProcessServer">
                <OutOfProcessServer ServerName="Samples.MathUtility">
                      <Path>Samples.MathUtility.exe</Path>
                      <Instancing>singleInstance</Instancing>
                      <ActivatableClass ActivatableClassId="Samples.
                      MathUtility" />
                </OutOfProcessServer>
          </Extension>
    </Extensions>
</Package>
```

Path specifies the path to the executable, the **ActivatableClass** defines the runtime classes, and **Instancing** specifies whether the executable runs as a single instance or multiple instances.

In the App package, the application can be configured to run in full trust by specifying the application entry point with "EntryPoint="Windows.FullTrustApplication" for Win32 Apps, or the application can be configured to run with low trust level without the "Windows.FullTrustApplication" entry point for UWP Apps. The different trust-level Apps create different instances of the servers when they create the activation class instances. This model is called ActivateAsActivator, or AAA, and it creates separate instances of the server based on the client.

However, the server instancing behavior can be controlled by specifying ActivateAsPackage (AAP), a single instancing behavior for servers in the package. The single instance for the package can be specified in the <**OutOfProcessServer**> element attribute **IdentityType="activateAsPackage**;" with this multiple clients can call the server, but a single instance of the server will be executed.

7.8.6.3 Firewall exception rules

The Windows Firewall is part of the Windows operating system to secure the system as well as the network by blocking unauthorized access to the system and allowing authorized communication to and from the system.

Windows Firewall allows to configure the inbound and outbound traffic rules and exceptions for the applications so that the application can access the network and Internet without any barrier.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Firewall rules for the MSIX packaged application can be configured by defining the FirewallRules package extension of category "**windows.firewallRules.**"

```
<Package ...>
     …
     <Extensions>
          <desktop2:Extension Category="windows.firewallRules">
               <desktop2:FirewallRules Executable="App.exe">
                    <desktop2:Rule Direction="in" IPProtocol="TCP"
                    LocalPortMax="6868" LocalPortMin="6868"
                    Profile="all"/>
               </desktop2:FirewallRules>
          </desktop2:Extension>
     </Extensions>
<Package>
```

The extension shown above defines the rule for the App.EXE to allow the inbound traffic with the TCP protocol over the port 6868.

Direction is the direction of the rule, either “**in**” or “**out**.” **IPProtocol** is the protocol for the rule. Profile is the network, domain, private, domainAndPrivate, public, or all. **LocalPortMin** is the minimum value for the local port between 1 and 65535. **LocalPortMax** is the maximum value for the local port, an integer between 1 and 65535.

7.8.6.4 InstalledLocationVirtualization

This package-level extension is used to allow an application to modify files that are part of the package folder by using a copy-on-write technique. This is a possible alternative to using the PSFFileRedirectionFixup. The feature has an unknown OS limitation, likely to be 20H2 or above (but possibly later).

```
<Package ...>
     …
     <Extensions>
          <uap10:Extension Category="windows.installedLocation
          Virtualization">
               <uap10:InstalledLocationVirtualization>
                    <uap10:UpdateActions ModifiedItems="keep"
                                         DeletedItems="reset"
                                         AddedItems="keep" />
               </uap10:InstalledLocationVirtualization >
          </uap10:Extension>
     </Extensions>
<Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

It is believed that the feature does not affect VFS files within the package but is intended such that end-user configuration could be stored in the package folder and changes would be redirected to the end-user's AppData area for the package. (**NOTE**: The documentation says AppData, but we believe is it LocalAppData, similar to the PSF.)

7.8.6.5 Desktop9 package extension for ContextMenuHandler

Although nothing is documented in the Microsoft online documents and we have not located a schema file for it, we have been made aware that there is a Desktop9 Package Extension for the category

"windows.fileExplorerClassicContextMenuHandler."

The word is that use of this new package extension will require you to be running Windows 11 (Windows 10 21H2 was not mentioned in our correspondence, but we consider support on Windows 10 21H2 to not be out of the question).

We are further told that this should support ContextMenuHandlers that do not work using the Desktop4 approach, like the one in NotePad++.

Presumably, this new option will be preferred once non-supported OS versions go out of maintenance. Until then, the application extension "windows.fileExplorerContextMenus" should be used. See 7.12.6.5 Example context menu handler extension for an example of the desktop4 application extension.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Here is an example of using the new desktop9 version. A file explorer context menu always involves an in-process COM DLL that is implemented by the Windows Explorer, allowing the user to right-click on a file of associated file type and displaying a menu that is context aware and generated by the COM DLL on the fly.

```
<Package ...
     xmlns:desktop9="http://schemas.microsoft.com/appx/manifest/
     desktop/ windows10/9"
     xmlns:com="http://schemas.microsoft.com/appx/manifest/com/
     windows 10"
>
     …
     <Extensions>
          <com:Extension Category="windows.comServer">
               <com:ComServer>
                    <com:SurrogateServer DisplayName="Notepad++ Shell
                    Extensions">
                         <com:Class Id="B298D29A-A6ED-11DE-BA8C-
                         A68E55D89593"
                         Path="VFS\ProgramFilesX64\Notepad++\NppS
                         hell_06.dll"ThreadingModel="STA" />
                    </com:SurrogateServer>
               </com:ComServer>
          <com:Extension>

          <desktop9:Extension Category="windows.fileExplorerClassic
          ContextMenuHandler">
               <desktop9:FileExplorerClassicContextMenuHandler>
                    <desktop9:ExtensionHandler Type="*"
                    Clsid="B298D29A-A6ED-11DE-BA8C-A68E55D89593"/>
               </desktop9:FileExplorerClassicContextMenuHandler>
          </desktop9:Extension>
     </Extensions>
<Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.9 Applications element and application element

In the MSIX package, the MSIX App process executes inside the container, isolated using file system and registry virtualization. The Application element in the package AppXManifest file helps to create the application entry in the start menu, so that the container application can be launched from outside.

Hence, Application is one of the important elements in the AppXManifest file if the package contains an App to be executed. Application elements represent the packaged App, configured to update the system start menu and the tile views. Multiple Apps can be configured in the package AppXManifest, and during installation the system start menu will be updated with all the application entries.

However, it is not necessary to have the Application element in the AppXManifest. MSIX packages for framework dependency, like Microsoft .Net Core Runtime or Microsoft Visual C++ runtime packages, will not have the Application element configured. It's only meant to install the framework libraries to the target system to resolve the dependency for the main package.

There are lot of Packaging tools available in the market that supports generating MSIX package from the existing Windows installer package automatically and adding the Application entries into the package AppXManifest, based on the shortcuts configured in the Windows installer package.

The Applications element, a child of the Package element, is simply a container to hold a list of Application elements.

7.9.1 Schema source

The Applications and Application elements are defined as part of the FoundationManifest, with support from AppXManifestTypes.

Within the Application, there may be a variety of elements that come from numerous schemas. These will be described separately.

7.9.2 Microsoft documentation

The Applications element is described at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-applications.

7.9.3 Implementation Notes

The order of Application elements within Applications is unimportant, with one exception. When a package is installed using the AppInstaller GUI interface, there is an option for AppInstaller to launch after completing installation. AppInstaller will choose the first Application element in the AppXManifest.

Generally speaking, it is good practice to keep sub-elements of an Application element together when they are related to the executable listed as the Application parameter Executable. However, this is often not a hard requirement.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

For example, there may be COM components used by multiple applications, and validation rules prevent you from declaring the COM component more than once in the package. In that case, we often declare the component in the first application that needs it and leave it out of the others.

It is also possible to create a dummy Application, as there is no requirement that the file referenced in the Executable parameter of the Application element actually exist in the package.

7.9.4 Examples

Coverage for the sub-elements of the Application element will be covered in separate sections, but the examples here will include some of the more common cases.

The AppXManifest below is from a package created for the UWP application. "**EntryPoint**" is configured to the activatable class, and the "**Executable**" is the UWP App.

```
<Package …>
    …
    <Applications>
        <Application EntryPoint="App1.App" Executable="App1.exe"
                Id="App">
            <uap:VisualElements BackgroundColor="transparent"
                    Description="App1"
                    Square44x44Logo="Assets\Square44x44Logo.png"
                    Square150x150Logo="Assets\Square150x150Logo.png"
                    DisplayName="App1">
                <uap:DefaultTile Wide310x150Logo="Assets\Wide310
                x150Logo.png"/>
                <uap:SplashScreen Image="Assets\SplashScreen.png"/>
            </uap:VisualElements>
        </Application>
    </Applications>
</Package>
```

Yes, it is possible to combine UWP applications and MSIX applications within the same package!

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

The next example is from a package created for the Win32 application to package up the SysInternals Process Explorer as an MSIX application. The "**EntryPoint**" for the Win32 desktop application will be "Windows.FullTrustApplication," and the "Executable" will be the packaged application procexp64.EXE in this case.

```
<Package>...
     ...
     <Applications>
          <Application EntryPoint="Windows.FullTrustApplication"
               Executable="VFS\ProgramFilesX86\ProcExp\procexp64.exe"
               Id="NewApplication1">
             <uap:VisualElements Square44x44Logo="Assets\NewApp1-
             AppList.png"
             Square150x150Logo="Assets\NewApp1-MedTile.png"
             BackgroundColor="transparent"
             Description="Procexp64"
             DisplayName="Procexp64">
                     <uap:DefaultTile ShortName="Default
                     Visual Asset"/>
            </uap:VisualElements>
            <Extensions/>
          </Application>
     </Applications>
</Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The third example is from the Hosted Web App created for the https://www.revenera.com website. The Application attribute "**StartPage**" was changed to https://www.revenera.com, and the child "ApplicationContentUriRules" elements are defined to open the web pages in the App. If the **StartPage** is specified, then both **Executable** and **EntryPoint** attributes are not required to be specified.

```
<Package …>
    …
    <Applications>
        <Application Id="App" StartPage="https://www.revenera.com/">
            <uap:ApplicationContentUriRules>
                <uap:Rule Type="include"
                Match="https://www.revenera.com/"WindowsRuntimeAcc
                ess="all" />
                <uap:Rule Type="include"
                Match=https://*.facebook.com/*
                WindowsRuntimeAccess="none" />
            </uap:ApplicationContentUriRules>
        </Application>
    </Applications>
</Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.10 Applications: VisualElements element

Application entries in the AppXManifest may include the Uap VisualElements element to define various graphics that will be used to represent the application entry. This, essentially, is the replacement for start menu shortcuts used in traditional applications. MSIX has no replacement for desktop shortcuts. Similarly, there is no equivalent to the MSI “advertised shortcut.”

While traditionally Windows has used icon files (which support multiple sizes inside a single icon file), under MSIX all these graphic files are individually sized png or jpg files.

Additionally, a splash screen and preferred rotations may be declared.

7.10.1 Schema source

The schemas needed to define VisualElements and Splash Screen are UAP, plus the AppXManifestTypes.

7.10.2 Microsoft documentation

You can find documentation on the VisualElements element of the AppXManifest at the following links, but the text that follows will offer some practical advice.

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-visualelements

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap3-visualelements-manual

The child elements may be found here:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-splashscreen

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-initialrotationpreference

7.10.3 Element attributes

DisplayName: This is a name used for display purposes. It will be used, for example, in the start menu. This can be a localized string using the Resource.pri.

Description: This string may be used when a further description of the application is possible. Examples include tooltips when hovering and the Windows settings “Apps” dialogs. This can be a localized string using the Resource.pri.

BackgroundColor: Either a standard named color or #rrggbb. This is used behind a transparent logo when specified. When added to the first application element in the file, this color is also used for buttons like those in AppInstaller, and in the Microsoft Store if distributed there.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

If not specified, the system will use the end-user's start menu background color or select a color to contrast a calculated average color of the image.

Square150x150Logo and **Square44x44Logo** contain relative paths to two jpg files. Traditionally, these files will be in the Assets folder. They are used to display in the start screen and other system places. They are used for initial display in the start menu and are not related to pinned tiles. The attribute names indicate the required size of the graphic files in pixels.

AppListEntry is a value of either "default" or "none." When specified as none, this application will not appear in the "All Apps List" of the start menu. Setting this to none in essence allows you to create an application entry so that you can specify application extensions that may be unrelated to any single application, such as fonts and shell extensions. Although you must specify a valid-looking name for an EXE file in the parent Application element, the file referenced need not actually exist.

VisualGroup is only available as an attribute of **uap3:VisualElements**. This attribute was added to bring back the concept of folders in the start screen. It is ignored on operating systems prior to 2004. The value of the attribute is the folder name to be used to hold the application entry. The folder will not be implemented unless there are multiple applications using the folder within the package. Folders are not recommended unless you have quite a few applications with start menu elements within your package/packages, as end-users are learning to work with the flat start screen alphabetical listing. Without the folders, you should ensure that your start screen display names all start with the same set of characters to keep them grouped together.

7.10.4 Sub-elements

7.10.4.1 uap:DefaultTile

This element contains the information needed to display the app on the start screen pinned area. It represents the default display, which could change if the application supports the now- deprecated "live tiles" functionality. The attributes for the default tile include three sizes of jpg files for tiles (**Square71x71Logo, Square310x310Logo,** and **Wide310x150Logo**) and a ShortName that may be used. The default tile also has several optional child elements.

Documentation on this sub-element may be found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-defaulttile

7.10.4.2 uap:InitialRotationPreference

This element allows for the specification of a preferred rotation when the app is started on a device that supports rotations (mostly tablets/phones). This preference applies to the Splash Screen (if specified) and application launch. This element is only needed if the application supports rotation, your app design has a clear preference for ease of use, and you want to lead the user into using that rotation. More information on this sub-element may be found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-initialrotationpreference

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.10.4.3 uap:LockScreen

Used only by UWP apps that support the live display of a badge on the lock screen by using a background task. The concept was put in place for things like email and social media notifications to be visible when the OS is in a locked state. There were even a few hardware designs with a special place to display them (like on the cover of a tablet), but they didn't catch on. More information on this sub-element may be found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-lockscreen

7.10.4.4 uap:SplashScreen

The SplashScreen element is only used in UWP apps. The concept was to separate the display of the splash screen from the EXE code to get the splash starting sooner. Both UWP and MSIX apps are free to have EXE-based code that displays a traditional splash screen independent of this element. The element is ignored on any app that is set as Windows.FullTrustApplication. More information on this sub-element may be found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-splashscreen

7.10.5 Implementation notes

The VisualElements attributes are primarily associated with the start menu. Application entries in the AppXManifest for other purposes should include a VisualElements entry but set the AppListEntry attribute to "none" and not add the DefaultTile sub-element.

As discussed in 4.5.4 Assets and the start menu, the start menu uses different sizes of graphics for the start menu itself and for pinned items (which may be pinned as small, medium, large, or wide). And for any of those, we may have different files for systems using different text scaling for high-count DPI monitors. Basically, in the start menu Microsoft treats your graphics as graphics including text, whether or not you have any. So, if the end-user has requested text to be displayed at 400% (a user session-wide setting), the 4x-sized version of your graphic is used (e.g., 310x310.scall-400.png).

It is likely that your existing project uses icon files that have varying sizes of graphics embedded in a single file. You will need to create new graphic files for a variety of sizes, most of which will be larger than you would currently have.

- Someone should first make a high-quality, large-sized square graphic file in png or jpg format. Without a splash screen, the base image your designer creates is 1240x1240 (310x310 for systems with text scaling set to x400).
- From that image, make scaled-down sizes for each of the sizes needed. Generally, a resizing of the largest will be acceptable, but it may be possible that the smaller sizes will need a less complex version of the original image. These will be used for the package logo, all the start menu graphics (except for the "wide" format ones), and badges (if used).

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

- A second base image of 1240x600 is needed for start menu pinned “wide” format. While you can paste a scaled-down copy of the original image, it will look odd, and a fuller image should be developed.
- From the second image, scaled-down copies are also needed.
- If using the new splash screen, a separate image (2480x1200) for that would also be produced and scaled down.

The background of these graphics may also be set. While a transparent background might have been interesting in the past, user customization to enable “dark mode” or otherwise set the background color of the start menu itself means that you should probably set a background color for your graphic if the file uses transparency.

Microsoft uses the term “unplated” to describe a situation where the graphic is used as is and “plated” to describe one in which a background gets applied. In some situations, the system will consider the unplated asset but look at the colors to determine if the plated should be used instead. See documentation at https://docs.microsoft.com/en-us/windows/uwp/design/style/app-icons-and-logos for a fuller discussion on asset sizes, uses and recommendations.

Adding the SplashScreen sub-element and image is only effective if your application contains UWP code to implement the splash screen. If your software doesn’t call these UWP APIs or if the application EntryPoint property points to Windows.FullTrustApplication, any entries declared here are ignored. This setting is independent of any splash screen code that might be in traditional Win32-or .Net Framework-based programs, so it is not necessary to convert to the newer form if your current project works well. Once Project Reunion becomes a reality, it may be possible to convert your code over if desired. See the Microsoft documentation at https:// docs.microsoft.com/en-us/windows/uwp/launch-resume/splash-screens for more information on adding a splash screen using package AppXManifest. If your app takes a while to load, consider adding an extended splash screen. For step-by-step guidance, see https://docs.microsoft.com/en-us/windows/uwp/launch-resume/create-a-customized-splash-screen.

The InitialRotationPreference sub-element is also only effective if your application is UWP based. It will be ignored otherwise.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.10.6 Example

Below is a sample of an AppXManifest file for an application that converted from a .Net Framework-based application. The tooling used automatically added the unnecessary SplashScreen and InitialRotationPreference sub-elements; they are shown here for completeness.

```
<Package … >
     …
     <Applications>
           <Application EntryPoint="Windows.FullTrustApplication"
                        Executable="ProgramApp.exe" Id="Program">
                 <uap:VisualElements AppListEntry="default"
                             Description="Application Description"
                             Square44x44Logo="Assets\YourApplication
                             Square44x44Logo.png"
                             Square150x150Logo="Assets\YourApplication
                             Square150x150Logo.png"
                             DisplayName="TheProgram"
                             BackgroundColor="transparent">
                       <uap:DefaultTile
                             Square71x71Logo="Assets\YourApplication
                             Square71x71Logo.png"
                             Square310x310Logo="Assets\YourApplication
                             Square310x310Logo.png"
                             Wide310x150Logo="Assets\YourApplicationWide
                             310x150Logo.png"/>
                       <uap:SplashScreen Image="Assets\YourApplication
                       SplashScreen.png"/>
                       <uap:InitialRotationPreference>
                             <uap:Rotation Preference="portrait"/>
                             <uap:Rotation Preference="landscape"/>
                       </uap:InitialRotationPreference>
                 </uap:VisualElements>
           </Application>
     </Applications>
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Your original project likely importedinstructions to build lnk or URL files when the installer ran. When building your MSIX-based installer, these would go away, but if your installer took lnk or url files for distribution, you could consider removing them. It is not necessary to remove lnk or url files from your package, but they will have no effect.

7.11 Application: ContentURI element

Windows 10 allows you to have your application include end-user access to a start menu item that is a UWP-based application to control access to certain hosted websites. This project type is called Hosted Web App in the developer tools. The web application running on your web server can be configured to render from a UWP App. So the Hosted Web App can be packaged and published to the Windows Store as an App, and the end-user can download and install it as a normal UWP App.

Content URI rules are a part of Hosted Web Apps to provide additional security to the App. Only domains or URLs listed in the Content URI rules can be viewed within the Hosted Web App, and other linked pages in the web application will be opened outside the App in the default browser. Content URI rules can be configured to list a particular url or a domain. You can then set them to “include” to have them included in the App scope or “exclude” to have them considered out of scope.

The Content URI rules are also used to determine the WinRT API access for each URI. For security reasons, only the trusted pages will be configured with Windows WinRT API access. With this, the Hosted Web Apps have the same advantages as any other Windows app by accessing the platform-specific APIs to get system features such as Live Tiles, Cortana interactions, Location, Camera and Sensors.

For each URI, the level of WinRT library access can be specified by:

- **all:** Allows access to the entire UWP API surface; defined through the App Capabilities.
- **allow for web only:** Web application can execute custom WinRT components included in the app package but can’t access all UWP APIs.
- **none:** Web application cannot access local UWP APIs.

URIs will tell the Hosted Web App what pages can access Windows Runtime (WinRT) libraries and what level of access those URIs can have and include only URIs that are directly used within the application and only those that need access to UWP APIs.

The URIs in this list must use HTTPS and may contain subdomain wildcards (for example, "https://.microsoft.com"), but they can't contain domain wildcards(for example, "https://.com"and "https://").

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.11.1 Schema source

The schemas needed to define VisualElements are UAP and the AppXManifestTypes.

7.11.2 Microsoft documentation

The documentation page for this element is https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-applicationcontenturirules

7.11.3 Example

NOTE: Example Project used in this section is available on this GitHub page:

https://github.com/flexera-public/MSIXDevGuide/tree/main/Chapter_7_AppxManifest_Schema/7.11_ApplicationContentURI

Content URI rules are defined in the package AppXManifest by defining the "uap:ApplicationContentUriRules" under the Application element. The child element "uap:Rule" will be defined for each web page with the WinRT access level.

```
<Package>
    …
    <Applications>
        <Application Id="InstallShieldHome"
                     StartPage="https://www.revenera.com/">
            <uap:ApplicationContentUriRules>
                <uap:Rule Type="include"
                          Match="https://www.revenera.com/"
                          WindowsRuntimeAccess="all" />
                <uap:Rule Type="include"
                          Match="https://*.facebook.com/*"
                          WindowsRuntimeAccess="none" />
            </uap:ApplicationContentUriRules>
        </Application>
    </Applications>
</Package>
```

The above AppXManifest is from the Hosted Web App created for the revenera.com website. The Application "StartPage" was changed torevenera.com, and the "ApplicationContentUriRules" are defined to open the web pages in the App.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.12 Application: Extensions element

NOTE: There are two forms of the “Extensions” and “Extension” elements, one which is a child of the “Package” element and another a child of the “Application” element. This section covers the version of extensions that lives under the Application element.

Application Extensions are the extensibility points to extend the Application functionality such as to integrate the application with the Windows 10 system or to interact with other applications. Application Extensions are declared in the AppXManifest file under the Application/Extensions node with a specific extension category. The schema for each extensibility point is category-specific, and the configuration and the information required for each extension to register with the system vary each based on the extension category.

A package may contain several Application elements, and in theory the extensions placed at the application level apply to that application only. However, this should be considered a guideline, as some application extension elements affect the entire package anyway.

The Application extension helps the Application to integrate and register the application with the Windows 10 system for the various system functions, such as to register file type association to open the file in the registered Application by double-clicking on the file, or to register the Application as a Windows service to start during the system start, or to register the fonts with the system or to expose a COM object to other Applications.

In the traditional Windows installer, all these configurations are done in the registry during the installation. Windows installer creates the necessary keys in the registry based on the information stored in the Windows installer tables; for example, the file type association information will be stored in the Verb table and the COM component information will be stored in the Registry table or updated during the installation.

Each “Extension” element will have a mandatory “Category” attribute. The value in the category dictates what sub-elements may be included under this extension.

7.12.1 Schema source

The Application-level Extensions element is defined in the FoundationManifestSchema, with help from AppXManifestTypes and, optionally, some of the following schemas for the child elements:

- Com, Com2, and Com4
- Desktop, Desktop2, Desktop3, Desktop4, Desktop6, and Desktop7
- Rescap, Rescap2, and Rescap3
- Wincap 2, and Wincap3
- Uap, 2, 3, 4, 6, 8, 10, 11, and 13

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.12.2 Microsoft documentation

The documentation page for the Application Extensions element is https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/appxmanifestschema2010-v2/element-applicationextensionchoice

7.12.3 Element Attributes

The Extensions element has no attributes.

7.12.4 Sub-elements

The application-level Extensions element may contain a sequence of Extension elements. Each Extension element will have a parameter named Category, which defines the content form allowed for that extension type.

The following application-level extension categories exist in the schema sets. Links to the specific page of the Microsoft documentation are given when they exist:

Schema	Category	Microsoft Documentation
Com and Com2 and Com4	windows.comServer	com:ComServer
	windows.comInterface	com:ComInterface
		NOTE: Each schema version of COM has separate implementations of the sub-elements: however, the documentation is for all three schemas. See the actual schema for small differences between different schema versions.
Desktop	windows.fullTrustProcess	desktop:FullTrustProcess
	windowsstartupTask	None
	windows.toastNotificationActivation	None
	windows.searchProtocolHandler	None
Desktop2	windows.appPrinter	desktop2:AppPrinter
	windows.searchFilterHandler	desktop2:SearchFilterHandler

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

	windows.searchPropertyHandler	desktop2:SearchPropertyHandler
	windows.mailProvider	desktop2:Extension (see Remarks)
Desktop3	windows.cloudFiles	desktop3:CloudFiles
	windows.autoPlayHandler	desktop3:AutoPlayHandler
Desktop4	windows.fileExplorerContextMenus	desktop4:FileExplorerContextMenus
Desktop6	windows.service	desktop6:Service
Desktop7	windows.approvedShellExtension	None
	windows.controlPanelItem	None
	windows.service	None
	windows.mailProvider	None
	windows.Shortcut	None
	windows.applicationRegistration	None
	windows.desktopAppMigration	None
	windows.systemFileAssociation	None
Rescap	windows.settingsApp	None
Rescap2	windows.userDataAccountsProvider	rescap2:Extension (See example)
Rescap3	windows.lockScreenComponent	rescap3:Extension (See remarks)
	windows.desktopAppMigration	rescap3:DesktopAppMigration
Uap	windows.fileTypeAssociation	uap:FileTypeAssociation
	windows.protocol	uap:Protocol
	windows.autoPlayContent	uap:AutoPlayContent

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

windows.autoPlayDevice	uap:AutoPlayDevice
windows.shareTarget	uap:ShareTarget
windows.search	None
windows.fileOpenPicker	uap:FileOpenPicker
windows.fileSavePicker	uap:FileSavePicker
windows.cachedFileUpdater	None
windows.cameraSettings	None
windows.accountPictureProvider	None
windows.printTaskSettings	None
windows.lockScreenCall	None
windows.appointmentsProvider	uap:AppointmentsProvider
windows.alarm	None
windows.webAccountProvider	uap:WebAccountProvider
windows.dialProtocol	uap:DialProtocol
windows.appService	uap:AppService
windows.mediaPlayback	uap:MediaPlayback
windows.print3dWorkflow	None
windows.lockScreen	uap:LockScreen
windows.aboveLockScreen	None
windows.personalAssistantLaunch	None
windows.voipCall	uap:VoipCall

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Uap2	windows.webAccountProvider	None
Uap3	windows.appExtensionHost	uap3:AppExtensionHost
	windows.appExtension	uap3:AppExtension
	windows.appUriHandler	uap3:AppUriHandler
	windows.appointmentDataProvider	uap3:AppointmentDataProvider
	windows.emailDataProvider	uap3:EmailDataProvider
	windows.contactDataProvider	uap3:ContactDataProvider
	windows.appExecutionAlias	uap3:AppExecutionAlias
	windows.appService	uap3:AppService
	windows.protocol	None (is extension to uap:protocol)
	windows.fileTypeAssociation	None (is extension to uap:fileTypeAssociation)
Uap4	windows.sharedFonts	uap4:SharedFonts
	windows.userDataTaskDataProvider	uap4:UserDataTaskDataProvider
	windows.mediaCodec	uap4:MediaCodec
	windows.contactPanel	uap4:ContactPanel
	windows.loopbackAccessRules	uap4:LoopbackAccessRules
	windows.devicePortalProvider	uap4:DevicePortalProvider
	windows.printWorkflowBackgroundTask	uap4:Extension (See Remarks)
	windows.printWorkflowForegroundTask	None
Uap5	windows.userActivity	uap5:UserActivity
	windows.mediaSource	uap5:MediaSource

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

	windows.videoRendererEffect	uap5:VideoRendererEffect
	windows.activatableClass.outOfProcessServer	uap5:OutOfProcessServer
	windows.startupTask	uap5:StartupTask
	windows.appExecutionAlias	uap5:AppExecutionAlias
Uap6	windows.barcodeScannerProvider	uap6:BarcodeScannerProvider
	windows.localExperiencePack	uap6:LocalExperiencePack
	windows.barcodeScannerPreviewProvider	None
Uap8	windows.posPaymentConnector	None
Uap10	windows.protocol	None
U1p12	windows.printSupportSettingsUI	None
	windows.printSupportExtension	None
	windows.printSupportWorkflow	None
	windows.printSupportJobUI	None
Uap13	Windows.phoneCallActivation	None
WinCap2	windows.internal.*	None
WinCap3	windows.internal.*	None

It is worth noting that some of these categories are not documented when we were writing the book, and many others had significant and obvious errors. We have reported these to Microsoft, and documentation needs may have been addressed since. When missing, refer to the referenced schema file for definitions. Also note that in some cases when items are documented, the documentation discourages use of the item, so also check the Microsoft documentation before using.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.12.5 Implementation notes

See the Implementation Notes section for the Package-level Extensions for a discussion on understanding both Package and Application Extensions.

7.12.6 Examples

This section contains examples of the most popular/interesting application extensions.

7.12.6.1 Example file type association

File type association extension registers the Application to handle files of the specified types so that the file can be processed by the application on double-clicking on the file. However, certain file extensions are reserved for use by built-in apps and the operating system. Attempts to register the Application with a file extension will be ignored by the system.

In order to register file type association for an MSIX packaged Application, it is required to add the File type association extension with the "windows.fileTypeAssociation" category under the Application element in the AppXManifest file.

```
<Package …>
     ...
     <Applications>
          <Application …>
               …
               <Extensions>
                    <uap:Extension
                    Category="windows.fileTypeAssociation">
                         <uap:FileTypeAssociation Name="ftatest">
                              <uap:SupportedFileTypes>
                                   <uap:FileType
                                   ContentType="html/text">.flex<
                                   /uap:FileType>
                              </uap:SupportedFileTypes>
                              <uap:Logo>Images\StoreLogo.png<
                              /uap:Logo>
                              <uap2:SupportedVerbs>
                                   <uap2:Verb Id=”Open” >
                              </uap: SupportedVerbs >
                         </uap:FileTypeAssociation>
                    </uap:Extension>
               </Extensions>
          </Application>
     </Applications>
</Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The **uap:FileTypeAssociation** attribute Name is mandatory to specify the unique descriptive name for the file type association in small letters without any space. This element supports many optional attributes and sub-elements, and only a few common ones are shown in this example. See documentation on this element for more information at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-filetypeassociation

NOTE: We have been made aware that there is a new desktop9 schema that is not yet published that has a new way to expose FileTypeAssociations. This new extension is not an application extension but a package one.

The **uap:SupportedFileTypes** holds the actual **FileType** to be associated and is specified with the leading period character. The **ContentType** attribute is optional, though it is a good practice to specify. The documentation link for this element is at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-supportedfiletypes

The optional **Logo** element has a relative file path used when system components (like File Explorer) display files of this type in image form. This must be a png file and may not be an embedded resource in an EXE or DLL file. Documentation for the Logo element is found https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element- uap-logo

The optional **SupportedVerbs** may be used to define shell integration verbs, allowing the user to start the associated application by a static right-click of the file. (Note that shell extensions like context menu are handled separately.) Documentation links for these are at

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap2-supportedverbs

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap2-verb

7.12.6.2 Example Windows service extension

Microsoft Windows services, formerly known as NT services, enable you to create long-running executable applications that run in their own Windows sessions. These services can be automatically started when the computer boots, they can be paused and restarted, they can operate in the background and they do not show any user interface.

From Windows 10, version 2004, Microsoft started supporting Windows services as part of the MSIX package. Windows services can be configured and installed as part of the MSIX package by adding an Application extension with the "**windows.service**" extension category. This category of extension will be ignored on down-rev operating systems.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

```
<Package …>
     …
     <Capabilities>
          <rescap:Capability Name=”runFullTrust”/>
          <rescap:Capability Name=”packagedServices”/>
          <rescap:Capability Name=”localSystemServices”/>
     </Capabilities>
     ...
     <Applications …>
          <Application>
               …
               <Extensions>
                    <desktop6:Extension Category="windows.service"
                    EntryPoint="Windows.FullTrustApplication"
                    Executable="ServiceExecutable.exe">
                         <desktop6:Service Name="Service Executable"
                         StartAccount="localSystem"
                         StartupType="auto"/>
                    </desktop6:Extension>
               </Extensions>
          </Application>
     </Applications>
</Package>
```

Desktop6: Service element

The **desktop6:service** element specifies the service that is installed and registered along with the app. The services can be configured to run under the Local Service, Network Service or Local System accounts. The service element has the attributes below to specify the service startup type, account and arguments.

The attributes for the service element are:

Name: The name of the service, a string between 1 and 32767 characters in length with non-white space characters at the beginning and end.

StartupType: The startup type for the service. The values supported are auto, manual and disabled.

StartAccount: The type of account in which to run the service. The values supported are localSystem, localService and networkService.

Arguments: Optional arguments to pass to the service; a string between 1 and 32767 characters in length with non-whitespace characters at the beginning and end.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Documentation on the service element may be found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-desktop6-service

7.12.6.3 Example Windows shared fonts extension

Fonts are supported to install and register as part of theMSIX package so that the fonts will be available to other applications installed for the user. To install fonts as part of the MSIX package, it is required to add an Application extension with the category windows.sharedFonts to register the font.

```
<Package …>
    ...
    <Applications>
        <Application …>
            …
            <Extensions>
                <uap4:Extension Category="windows.sharedFonts">
                    <uap4:SharedFonts>
                        <uap4:Font File="FontFile" />
                    </uap4:SharedFonts>
                </uap4:Extension>
            </Extensions>
        </Application>
    </Applications>
</Package>
```

The font file package path needs to be updated in the **File** attribute of the **Font** element, and the font file should be part of the MSIX package. Likewise, it is supported to install multiple fonts as part of the same extension.

Under Uap4, this application extension must be part of an Application element, but there is no required relationship between the application and these fonts. The implementation is really at the package level, and these fonts are also available to applications running outside this package container. Thus, it is OK to add this extension to any application element in the package.

NOTE: In UAP7, Microsoft introduced windows.sharedFonts as a package-level extension category. As the font implementation isn't specific to an individual application, this seems to be a better place to put the fonts, but only if you are deploying to systems with this support. This will become the preferred option when older operating systems that don't support UAP7 are out of maintenance. The UAP7 extension would still use the uap4:SharedFonts and uap4:Font elements.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.12.6.4 Example startup task extension

The Startup Task extension configures the Application to run at system startup or when the user logs in to the system. It is required to add the "windows.startupTask" extension under the Application extension in the AppXManifest file to run the Application at system startup or when the user logs in.

```
<Package …>
    …
    <Applications>
        <Application …>
            …
            <Extensions>
                <uap5:Extension Category="windows.startupTask"
                Executable="DesktopApp.exe"
                EntryPoint="Windows.FullTrustApplication">
                    <uap5:StartupTask TaskId="StartupId"
                    Enabled="false"
                    DisplayName="Desktop App" />
                </uap5:Extension>
            </Extensions>
        </Application>
    </Applications>
</Package>
```

The **Executable** and **EntryPoint** values of the extension should match those of the **Application** element that houses it. Under the extension is the **uap5:StartupTask** element that provides additional parameters:

TaskId: A unique identifier for the task. Using this identifier, the Application can call the APIs in this class to programmatically enable or disable a startup task.

Enabled: May be set to true for packaged desktop Applications to indicate that the Application is enabled for startup without first needing to call RequestEnableAsync.

DisplayName: The name of the task that appears in Task Manager.

The documentation for this extension type is found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap5-startuptask

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.12.6.5 Example context menu handler extension

Many Windows Applications have the requirement to add the context menu item to the Windows File Explorer context menu. A context menu is different from the static menu shown in the prior example for FileTypeAssociation in that it uses a Com-based component to determine which menu items should be shown based on the context of the right-click action against the associated type of file.

```
<Package …>
     ...
     <Applications>
          <Application …>
               …
               <Extensions>
                    <desktop4:Extension
                    Category="windows.fileExplorerContextMenus">
                         <desktop4:FileExplorerContextMenus>
                              <desktop4:ItemType
                               Type=".contoso">
                                   <desktop4:Verb
                                   Id="Command1"
                                   Clsid="d0c8bceb-28eb-49ae-
                                   bc68-454ae84d6264" />
                              </desktop4:ItemType>
                         </desktop4:FileExplorerContextMenus>
                    </desktop4:Extension>
                    <com:Extension Category="windows.comServer">
                         <com:ComServer>
                              <com:SurrogateServer
                              AppId="d0c8bceb-28eb-49ae-bc68-
                              454ae84d6264"
                              DisplayName="ContosoHandler">
                                   <com:Class
                                   Id="d0c8bceb-28eb-49ae-bc68-
                                   454ae84d6264"
                                   Path="ExplorerCommandVerb.dll"
                                   ThreadingModel="STA"/>
                              </com:SurrogateServer>
                         </com:ComServer>
                    </com:Extension>
               </Extensions>
          </Application>
     </Applications>
</Package>
```

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The MSIX package supports this integration with the Windows File Explorer context menu by specifying the "windows.fileExplorerContextMenus" Application extension in the AppXManifest.

A **desktop4:FileExplorerContextMenus** category application extension registers your context menu handler implementation by referring the **CLSID** of the class object, which is defined as part of another **ComServer** extension.

The **com:ComServer** element under this application extension registers a COM surrogate server with the class ID of your context menu handler implementation by implementing the ***IExplorerCommand*** or ***IExplorerCommandState*** interface.

Note that sometimes developers:

- Use the same context menu DLL with options to different applications.
- Combine several types of shell extensions into a single COM DLL.

You should be aware that although the FileExplorerContextMenu must be part of an Application-level extension, there is no requirement that it actually relate to that application. Thus, you may add all your ItemTypes and all your Verb elements for a given COM, and you can only register a COM Class ID GUID once in the AppXManifest, but that registration does not need to be in the same application element.

The Microsoft documentation for the extension and ComServer elements may be found here:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-desktop4-fileexplorercontextmenus

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-com-comserver

See also 7.8.6.5 Desktop9 package extension for ContextMenuHandler for the more modern Desktop9 extension for use on newer operating systems.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.12.6.6 Example desktop application migration extension

MSIX packaged Applications can coexist with the traditional desktop applications, and there is a need to migrate the existing shortcuts created in the task bar or in the start menu by the traditional installer to the newly packaged Application. The MSIX package supports the migration by adding a "**windows.desktopAppMigration**" Application extension.

```
<Package …>
      ...
        <Applications>
          <Application …>
            …
            <Extensions>
                <rescap3:Extension Category="windows.desktopApp
                Migration">
                      <rescap3:DesktopAppMigration>
                         <rescap3:DesktopApp AumId="[your_app_aumid]" />
                         <rescap3:DesktopApp ShortcutPath="
                         %USERPROFILE%\Desktop\[my_app].lnk"/>
                         <rescap3:DesktopApp ShortcutPath="%APPDATA%\
                         Microsoft\Windows\StartMenu\Programs\[my_app]
                         .lnk" />
                      </rescap3:DesktopAppMigration>
                </rescap3:Extension>
            </Extensions>
          </Application>
        </Applications>
</Package>
```

DesktopAppMigration requires specifyingthe Application User Model ID for the Application and the list of shortcut paths that are needed to migrate to target the MSIX packaged Application.

AumID: The Application User Model ID of your packaged Application.

ShortcutPath: The path to lnk files that start the desktop version of the Application.

NOTE: MSIX does not provide a way to trigger the uninstallation of the old application. This should be performed separately by the customer. Additionally, this feature does not attempt to migrate user data, although it is possible that the design of your application would find existing settings and data.

The documentation for this category schema is at this link:

https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-rescap3-desktopappmigration

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Microsoft also has a desktop7:DesktopAppMigration element in the desktop7 schema file, but at the time of writing this book, no documentation on this item was yet available.

7.12.6.7 Example protocol extension

Protocol extension registers the Application to become the default handler for a certain Uniform Resource Identifier (URI) scheme in Windows 10, by specifying the protocol extension in the AppXManifest file.

Usually, protocol information will be registered in the system registry during the Application installation, like how the default web browser is launched when we run the “http” web address, or the default mail client is opened when we run the “mailto” with mail address.

Certain URIs are reserved for use by built-in Applications and the operating system. Attempts to register the Application with a reserved URI will be ignored by the system.

In order to register the protocol with the MSIX package, it is required to specify the windows.protocol category Application extension as part of the package AppXManifest file. There are versions of the **windows.protocol** category in Uap, Uap3 and Uap7.

```
<Package …>
     ...
     <Applications>
          <Application …>
               …
               <Extensions>
                    <uap3:Extension Category="windows.protocol">
                         <uap3:Protocol Name="testprotocol"
                         Parameters=”args”/>
                         <Logo>images\logo.png</Logo>
                         <DisplayName>
                              Sample URI Scheme
                         </DisplayName>
                    </uap3:Extension>
               </Extensions>
          </Application>
     </Applications>
</Package>
```

Name: This is an attribute of the Protocol element. It is the name of the URI scheme, such as "mail to." This name must be unique for the package and have a string between 3 and 39 characters in length that contains numbers, lower case letters or a hyphen.

Parameters: This is an optional attribute added in Uap3and above variants.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

DisplayName: A friendly name that is displayed to users under the default Programs in the Settings.

Logo: A path to a file that contains an image to specify the logo to identify the URI scheme name in the default Programs in the Settings. This is not available in the original UAP variant.

Microsoft documentation for the Protocol element is only found for UAP, at this link: https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-protocol

7.12.6.8 Other common application extensions

The MSIX package AppXManifest supports a lot of Application extensions to extend the Application functionality. More information on the various Application extensions information can be found at https://docs.microsoft.com/en-us/uwp/schemas/appxpackage/uapmanifestschema/element-uap-extension.

The following category types are worthy of additional explanation:

- **COM Extensions:** To expose COM registrations to clients outside the Application package. Application extension "**windows.comServer**" and the "**windows.comInterface**" extension category expose the component interfaces to clients outside the MSIX package.
- **File Open Picker:** To make the Application choose and open the specified types of files by defining an Application extensibility point of type "**windows.fileOpenPicker**" extension. This is only effective in applications calling the UAP file open picker dialog.
- **File Save Picker:** To make the Application choose the file name, extension, and storage location for the specified types of files by defining an Application extensibility point of type "**windows.fileSavePicker**" extension. This is only effective in applications calling the UAP file save picker dialog.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

7.13 BuildMetadata element

Although not part of the formal MSIX AppXManifest schemas, the implementations for MSIX AppXManifest file allow you to specify custom build metadata tags. This metadata is of no interest to MSIX but may be used to help people and tools track build-specific properties.

Adding this metadata is optional and is not documented by the MSIX schema. But most of the third-party vendors available in the market add this build metadata information to the package during the conversion process, and Microsoft even includes build metadata in packages generated by the MSIX Packaging Tool.

7.13.1 Schema source

The Build Metadata schema is installed as part of the Windows SDK you are probably already using.

7.13.2 Microsoft documentation

None.

7.13.3 Element information

The **build:Metadata** element is typically used as a child of the Package Extensions element, although it does not use the Extension element and category like other package extensions.

As it is not formally part of the AppXManifest schemas, the Metadata element will be ignored by operating systems operations when installing your application package.

Although placement within the file is unimportant, most tooling will make it the last child of the Package. This element has no parameters, and it may have an unordered collection of sub-elements of type **build:Item**.

The **build:Item** element has a required parameter of **Name** and an optional **Version** parameter. Both parameters may contain any type of string as the value. You can consider these entries as name/value pairs, and no formal set of pairs is defined; the meaning of the pairs is up to the tooling that placed them there.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

7.13.4 Example

The basic use of build:Metadata in the AppXManifest file would look something like this:

```
<?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<Package
          xmlns:xs=”http://www.w3.org/2001/XMLSchema”
          targetNamespace="http://schemas.microsoft.com/appx/manifest
          /foundation/windows10"
          xmlns="http://schemas.microsoft.com/appx/manifest/foundatio
          n
          /windows10"
          xmlns:t="http://schemas.microsoft.com/appx/manifest/types"
          xmlns:build="http://schemas.microsoft.com/developer/appx/20
          15/build" >
    <Identity ... />
    <Properties … />
          ...
    </Properties>
    <Dependencies/>
    <Applications/>
    <Extensions/>
    <build:Metadata>
          <build:Item Name="InstallShield" Version="2021 R1"/>
          <build:Item Name="OperatingSystem"
          Version="10.0.19041.1023"/>
    </build:Metadata>
</Package>
</Package>
```

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 8: PSF Json Syntax

The json file used to configure the PSF is documented on GitHub and spread out as part of the markdown pages for the PSF itself at the following link:

https://github.com/microsoft/MSIX-PackageSupportFramework

Some of that information is reproduced here for reference and clarity.

Json is a semi-structured format not unlike XML. There exists an XML option to create the PSF configuration, plus a tool in the MSIX toolkit to convert it into json, but the tool does not seem to keep up with json changes all the time.

This config.json file may be placed in the root folder of the package, or optionally anywhere else. The PsfRuntime will start looking at the root package folder and then enumerate subfolders until it finds a file matching that name. Placing this file at the root of any package files under a VFS\ProgramFilesX** folder will avoid any possible file name conflicts with files of your package that happen to use the same file name.

8.1 About Json

If you are not familiar with Json, the Json syntax consists of a sequential list of objects, with each object in the form of “item” : “value.” When two objects follow each other in a sequence, a comma is used for separation. The start of each item is traditionally placed on a new line, and indentation for readability is optional. Any object value may be an array of objects, in which case the array elements are also a comma-separated sequence of objects. Square brackets are used to designate an array. Curly brackets designate an object. A Json file consists of a single high-level object. If a value is a file path, forward slashes are used.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

For example:

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

Traditionally, the item names are presented in “pascal case,” where the first letter is lower case and each new word has the first letter capitalized, as in “workingDirectory.” This is not a hard requirement, and some items in the config.json do not happen to conform to this.

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

8.2 General items

General reference for Json entries:

Array	Key	Value
applications	id	Use the value of the ID attribute of the Application element in the package AppXManifest.
applications	executable	The path to the executable that you want to start. This path is typically specified relative to the package root folder. In most cases, you can get this value from your package AppXManifest file before you modify it. It's the value of the Executable attribute of the Application element. Pseudo-variables and Environment variables are supported for this path.
applications	arguments	(Optional) Command line arguments for the executable. If the PsfLauncher.EXE receives any arguments, these will be appended to the command line after those from the config.json file. Pseudo-variables and Environment variables are supported in the arguments.
applications	workingDirectory	(Optional) A path to use as the working directory of the application that starts. This is typically a relative path of the package root folder, but full paths may be specified. If you don't set this value, the operating system uses the System32 directory as the application's working directory. If you supply a value in the form of an empty string, it will use the directory of the referenced executable. Pseudo-variables and Environment variables are supported in the workingDirectory.
applications	monitor	(Optional) If present, the monitor identifies a secondary program that is to be launched prior to starting the primary application. A good example might be PsfMonitor.EXE. The monitor configuration consists of the following items:
		"*executable*"— This is the name of the executable relative to the root of the package.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

		'arguments' — This is a string containing any command line arguments that the monitor executable requires. Any use of the string "%MsixPackageRoot%" in the arguments will be replaced by the string containing the actual package root folder at runtime.
		'asadmin' — This is a Boolean (0 or 1) indicating whether the executable needs to be launched as an admin. To use this option, set it to 1, and you must also mark the package with the RunAsAdministrator capability. If the monitor executable has a manifest (internal or external), it is ignored. If not expressed, this defaults to a 0.
		'wait' — This is a Boolean (0 or 1) indicating whether the launcher should wait for the monitor program to exit prior to starting the primary application. When not set, the launcher will WaitForInputIdle on the monitor before launching the primary application. This option is not normally used for tracing and defaults to 0.
applications	stopOnScriptError	(Optional) Boolean. Indicates that if a startScript returns an error, the launch of the application should be skipped.
applications	ScriptExecutionMode	(Optional) String value that will be added to the PowerShell launch of any startScript or endScript.
applications	startScript	(Optional) If present, used to define a PowerShell script that will be run prior to running the application executable. The startScript configuration includes the following items:
		'waitForScriptToFinish' — (Optional, default=false) Boolean. When true, PsfLauncher will wait for the script to complete or time out before running the application executable.
		'timeout' — (Optional, default is none) Expressed in ms. Only applicable if waitForScriptToFinish is true. If a timeout occurs it is treated as an error for the purpose of 'stopOnScriptError.' The value 0 means an immediate timeout. If you do not want a timeout, do not specify a value.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

		'runOnce' — (Optional, default=false) Boolean. When true, the script will only be run the first time the user runs the application.
		'showWindow' — (Optional, default=true). Boolean. When false, the PowerShell window is hidden.
		'scriptPath' — Relative or full path to a ps1 file. May be in package or on a network share. Use of pseudo-variables or environment variables is supported.
		'scriptArguments' — (Optional) Arguments for the 'scriptPath' PowerShell file. Use of pseudo-variables or environment variables is supported.
applications	endScript	(Optional) If present, used to define a PowerShell script that will be run after completion of the application executable. The endScript configuration includes the following items:
		'showWindow' — (Optional, default=true). Boolean. When false, the PowerShell window is hidden.
		'scriptPath' — Relative or full path to a ps1 file. May be in package or on a network share. Use of pseudo-variables or environment variables is supported.
		'scriptArguments' — (Optional) Arguments for the 'scriptPath' PowerShell file. Use of pseudo-variables or environment variables is supported.
Processes	executable	In most cases, this will be the name of the executable configured above with the path and file extension removed.
fixups	DLL	Package-relative path to the fixup, .msix/.appx to load. It is traditional to leave the x86 or x64 out of the fixup name in this field but include it in the actual filename; the PsfRuntime will find the appropriate version for the process.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

fixups	config	(Optional) Controls how the fixup DLLs behave. The exact format of this value varies on a fixup-by-fixup basis, as each fixup can interpret this "blob" as it wants.
packageRelative	base	Base folder designation of potential sources, to specify where the redirection needs to be applied. This base folder path is used in pattern matching. Base folder path is relative to the installation folder of the package, or a folder path relative to the drive that the package is installed to (nominally C:) or even a known folder identifier
packageRelative	patterns	An array of RegEx strings used on sources under the base folder path for matching.

8.3 Use of RegEx patterns

Several items in the json refer to the values using RegEx patterns. The PSF uses the RegEx algorithm from the C++ std library, which uses the default syntax for ECMAScript, sometimes referred to as the JavaScript variant (see https://docs.microsoft.com/en-us/cpp/standard-library/regular-expressions-cpp?view=vs-2017).

An excellent website to test RegEx patterns against strings to help ensure your pattern is correct is http://www.RegEx101.com.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Chapter 9: Bundle Schema

Once you have created an MSIX package, you may find that you need to create separate packages of the same application for deployment to different scenarios. This is handled through the creation of a MSIX bundle package, which in simplest terms is a zip-compressed file consisting of a BundleManifest and a set of MSIX packages.

You should not need to manually create the bundle's manifest XML file directly. Instead, use tooling, such as the makeappx utility, to create the .msixbundle file that includes this manifest along with the packages to be part of the bundle. This chapter serves only as a reference for anyone trying to figure out what was created.

There exists both an AppXBundle file format definition and an MSIXBundle file format definition. The AppXBundle file type is supposed to be for a bundle consisting of AppX files, and the MSIXBundle file type is supposed to be for a bundle consisting of MSIX files. But in reality, installer technology doesn't care whether you use the AppXBundle or MSIX bundle format; they are treated by the same utilities that just see packages.

The bundle was originally designed for Appx so that you could include a package for Windows and a different package for IOS or Android, which might expect different screen resolution formats, for example, or have different hardware to work with. With MSIX packages restricted to Windows operating systems, the bundle would more likely be used to allow you to have a 32-bit package and a 64-bit package in the bundle. Of course, your 32-bit app could be packaged up and used on a x64-bit version of Windows, but if you build both types and include them in a bundle, the most appropriate package will be selected upon installation.

The Bundle schema has been revised over time, as seen in the following table:

Name	Year	Purpose
BundleManifestSchema2013.xsd	2013	Initial schema for Appx Bundles
BundleManifestSchema2014.xsd	2014	Updated
BundleManifestSchema2016.xsd	2016	Updated
BundleManifestSchema2017.xsd	2017	Updated
BundleManifestSchema2018.xsd	2018	Updated
BundleManifestSchema2019.xsd	2019	Updated

Unlike the AppXManifest schemas, each version of the bundle schema is an independent replacement of the previous one. The root Bundle element has the targetNamespace attribute to tie it to the appropriate version.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

The Bundle manifest does make use of the AppXManifestTypes schema to define some simple types.

The latest version of the schema contains a sequence of the following top-level elements:

- **Identity:** Singleton containing Name, Publisher and Version attributes.
- **Packages:** Singleton with at least one but up to 10,000 sub-elements. These sub- elements may be either the 2019 Package elements or the 2018 elements.
- **OptionalBundle:** 0 to 10,000 sub-elements that are 2019 or 2018 Package elements. The OptionalBundle element contains attributes for the Name, Publisher, Version and File Name for these optional packages.

When the bundle format is used, your file servers and potentially management systems have fewer things to worry about. But when an installation occurs from a remote share, AppInstaller will read the manifest in the bundle, determine the appropriate package within the bundle and only download and install that specific package.

The complete documentation on the MSIX bundle is found at this link:
https://docs.microsoft.com/windows/msix/package/bundling-overview

Copyright ©2021 by Flexera, Revenera
and TMurgent Technologies, LLP

Flexera
Inform IT. Transform IT.
revenera

Chapter 10: PowerShell Cmdlets for MSIX

Microsoft supports managing MSIX packages using PowerShell cmdlets and provides PowerShell cmdlets to install and uninstall packages, as well as perform many package and bundle management activities in the base operating system. Like Windows installer MSI, Microsoft is not providing any option to install the MSIX package from the command line, so MSIX PowerShell cmdlets help install the MSIX package silently through the PowerShell scripts.

The commonly used PowerShell cmdlets that you would use in testing your packages are below. Note that many of the cmdlets were originally introduced for UWP and later extended to support MSIX. Commandlets like “Add-AppPackage” normally have an alias like “Add- AppXPackage,” and most scripts that you will find online will use the AppX form. It does not matter which form you use for either UWP or MSIX.

10.1 Add-AppxPackage

To install the signed MSIX package, execute the Add-AppPackage cmdlet with the package path:

Add-AppxPackage -Path "C:\Users\user1\Desktop\MyApp1.msix"

To install the MSIX bundle package, specify the MSIX bundle path:
Add-AppxPackage -Path "C:\Users\user1\Desktop\MyApp1.msixbundle"

To install using an AppInstaller file:
Add-AppxPackage -AppInstallerFile “\\server\share\folder\Myapp1.appinstaller”

The documentation page for this cmdlet for Windows 10 and Server 2019 is at https://docs.microsoft.com/en-us/powershell/module/appx/add-appxpackage?view=windowsserver2019-ps

10.2 Remove-AppxPackage

To uninstall the MSIX package installed, execute the Remove-AppxPackage with the package full name.

Remove-AppxPackage -Package "TestMultiplApps_1.0.0.0_x86 d14rjvjqp047y"

The MSIX package installs per user account. Add -AllUsers to remove the package from all user accounts in the system.

Get:

The documentation page for this cmdlet for Windows 10 and Server 2019 is at https://docs.microsoft.com/en-us/powershell/module/appx/remove-appxpackage?view=windowsserver2019-ps

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

10.3 Get-AppxPackage

To list all the packages installed for the user account, execute the Get-AppxPackage cmdlet which lists all the packages. Specify the -AllUsers to list the packages installed for all user accounts in the system.

Get-AppxPackage -AllUsers

To list the package information for a particular package, use the -Name parameter to specify the name of the package. Name parameter support * wild character to list the packages with specified name patterns.

Get-AppxPackage -Name "TestMultiplApps"

The documentation page for this cmdlet for Windows 10 and Server 2019 is at https://docs.microsoft.com/en-us/powershell/module/appx/get-appxpackage?view=windowsserver2019-ps

10.4 Get-AppxPackageManifest

To get the AppXManifest file of an installed MSIX package, execute the Get-AppxPackageManifest cmdlet with the package full name.

Get-AppxPackageManifest -Package “TestMultiplApps_1.0.0.0_x86 d14rjvjqp047y”

This cmdlet returns the read-only XML document object containing the information about the App package. To list the capabilities for a particular package, get the specific package using Get-AppxPackage cmdlet, as the Get-AppxPackageManifest accepts an array of AppxPackage objects. For example you can pipe a set of application package objects info this cmdlet and display the package Capabilities settings using the following command:

(**Get-AppxPackage** -Name "TestMultiplApps" | **Get-AppxPackageManifest**).Package.Capabilities

The documentation page for this cmdlet for Windows 10 and Server 2019 is at https://docs.microsoft.com/en-us/powershell/module/appx/get-appxpackagemanifest?view=windowsserver2019-ps

10.5 Invoke-CommandInDesktopPackage

To execute an application inside a Windows package container, execute the Invoke-CommandInDesktopPackage cmdlet with the package family name and the Application ID from the package AppXManifest, and the command to execute.

Invoke-CommandInDesktopPackage cmdlet requires the device to be in Developer mode for Windows 10 builds prior to 18922.

The cmdlet below invokes the Procmon.EXE from the package “TestMultiplApps_d14rjvjqp047y” under the Application element “App1.”

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

Invoke-CommandInDesktopPackage -AppId "App1" -PackageFamilyName" TestMultiplApps_d14rjvjqp047y" -Command "Procmon.EXE"

The command below launches the cmd.EXE from the package container; basically, this cmdlet helps debug the packaged application.

Invoke-CommandInDesktopPackage -AppId "App1" -PackageFamilyName" TestMultiplApps_d14rjvjqp047y" -Command "cmd.EXE" -Preventbreakaway

The documentation page for this cmdlet for Windows 10 and Server 2019 is at https://docs.microsoft.com/en-us/powershell/module/appx/invoke-commandindesktoppackage?view=windowsserver2019-ps

10.6 Shared package container cmdlets

Shared package containers allow you to create a shared runtime container for an MSIX packaged application. A shared package container is defined by an .XML file, which may be installed on an end-user's computer to tell the operating system that these packages should share a single container rather than be isolated from each other.

Add-AppSharedPackageContainer <path> cmdlet deploys the shared package container definition for the user.

Remove-AppSharedPackageContainer -Name <name> cmdlet removes the shared package container definition for the user.

Get-AppSharedPackageContainer -Name <name> cmdlet gets information about the shared package container and shows what packages are inside the shared package container.

Reset-AppSharedPackageContainer -Name <name> cmdlet deletes all the application data of the container.

There is no documentation page for these cmdlets at this time.

10.7 PowerShell references

The following references may also be useful:

https://docs.microsoft.com/en-us/windows/msix/desktop/powershell-msix-cmdlets

https://docs.microsoft.com/en-us/windows/msix/manage/shared-package-container#powershell-commands

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP

Flexera Inform IT. Transform IT.
revenera

About Flexera

Flexera delivers SaaS-based IT management solutions that enable enterprises to accelerate and multiply the return on their technology investments. We help organizations ***inform their IT*** with total visibility into their complex hybrid ecosystems, providing the IT insights that fuel better-informed decisions. And we help them ***transform their IT*** with tools that allow IT leaders to rightsize across all platforms, reallocate spend, reduce risk and chart the most effective path to the cloud.

More than 50,000 customers subscribe to our technology value optimization solutions, delivered by 1,300+ passionate team members worldwide. To learn more, visit flexera.com.

About Revenera

Revenera helps product executives build better products, accelerate their time to market and monetize what matters. Revenera's leading solutions help software and technology companies drive top line revenue with modern software monetization, understand usage and compliance with software usage analytics, empower the use of open source with software composition analysis and deliver an excellent user experience — for embedded, on-premises, cloud and SaaS products. To learn more, visit revenera.com.

Copyright ©2021 by Flexera, Revenera and TMurgent Technologies, LLP