

Bring existing desktop applications to the Universal Windows Platform

**Authors**

Adrian Fernandez Garcia – Windows AppConsult Engineer ([adrifer@microsoft.com](mailto:adrifer@microsoft.com))

Matteo Pagani – Windows AppConsult Engineer ([matteo.pagani@microsoft.com](mailto:matteo.pagani@microsoft.com))

## Objectives

After completing this lab, you’ll be able to:

* Take an existing desktop application and convert it into an app package, that can be easily deployed in an enterprise environment or published on the Windows Store
* Learn how the Desktop Bridge isn’t just a technology to convert an existing desktop app into an app package, but it’s a journey made of different steps, and the developer can decide to move at his own pace through each of them
* Learn how you can start to leverage some of the new Windows 10 features (like Cortana, live tiles, toast notifications, etc.) in a desktop app
* Learn how you can extend your existing desktop app by adding support to component that belong to the Universal Windows Platform, like background tasks or app services
* Learn how you can overcome some of the limitations of the Universal Windows Platform by creating a UWP app that can run on every Windows 10 device but, only on desktop, can invoke and communicate with a Win32 process

## Prerequisites

* Basic knowledge of C# development
* Basic knowledge of client development with the .NET framework
* Basic knowledge of Windows 10 and the Universal Windows Platform

## The requirements

* A computer with Windows 10 Anniversary Update or Windows 10 Creators Update. If you want to use the Desktop App Converter with an installer, you will need at least a Pro or Enterprise version, since it leverages a feature called Containers which isn’t available in the Home version.
* Visual Studio 2017 with the tools to develop applications for the Universal Windows Platform. Any edition is supported, including the [free Community one](https://www.visualstudio.com/vs/community/).

## Overview of the lab

The purpose of the lab is to guide you through all the steps offered by the Desktop Bridge: from the basic one (take an existing installer and convert it into an app package) to the most advanced one (convert all the existing desktop codebase into a Universal Windows Platform app).

The lab will be made by 5 different exercises, one for each step of the journey:

1. Convert
2. Enhance
3. Extend
4. Migrate
5. Reach all

# Getting things ready

In the main folder, you will find everything you’ll need to perform the lab. For each exercise, you will find a specific folder with:

* A subfolder called **01-Start**, that you can use in case you want to start directly from that exercise.
* A subfolder called **02-End**, with the final implementation of the exercise. This is how the definitive version of your exercise should look like if you did everything in the proper way.

Since it’s a journey, the various exercises are incremental: the starting point for each exercise will be the end of the of the previous one (except for Exercise 4, about the Migrate phase, since it refers to a different approach than the one described in the previous exercises).

By following the manual from the beginning to the end, you will be able to start from a traditional Windows Forms application and to get, in the end, a completely rewritten version as a Universal Windows Platform app. In the middle, you will learn all the intermediate steps of the journey, which are equally important: one of the most powerful features of the Desktop Bridge is that you can move at your own pace, and you aren’t required to take all the steps. Based on the project you’re working on, you may decide, for example, to stop at a specific step of the journey and leverage anyway some of the new Universal Windows Platform features, without being required to completely rewrite the app using the new platform.

## The starting application

Flight Tracker is an application to track flights. The application is built with Windows Forms, it’s based on .NET Framework 4.6.2, and it offers three main features:

1. Persist the data about the flight you want to track, so that it isn’t lost when you close and reopen it. For the sake of simplicity, we won’t use a real database, but we will allow the user to save just one flight using the system registry.
2. Export a boarding pass, which you can use to skip the check-in procedure at the airport. Again, to keep the application simple, the boarding pass will be just a text file created on the desktop of the computer. To simulate that the operation may take some time to be completed (for example, a real application would generate a proper PDF, which requires more work) we have added an artificial delay: the exporting procedure will always take 5 seconds to be completed.
3. Check for updates, to check if an updated version of the app has been released. This scenario will be simulated too: a real application would check against a web server if there’s a new release and eventually trigger an update procedure. In our case, we’re just displaying a message to inform the user that there are no updates available.

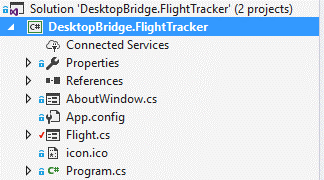
All these features are implemented using techniques that aren’t supported, at the time of the preparation of this lab, by the Universal Windows Platform:

1. The data about the flight is saved into the system registry, which can’t be accessed by a Universal Windows Platform app, since it runs inside a sandbox.
2. The boarding pass is exported directly on the user’s desktop, which is one of the folders that can’t be directly accessed by a Universal Windows Platform app.
3. The check for updates feature is technically supported (since it could be a simple HTTP call to a REST service or a file published on the web to check if an updated version is available), but a UWP app can’t update itself, since it’s automatically updated through the Store or any available enterprise distribution mechanisms (like a Mobile Device Management tool or the Windows Store for Business).

During the various exercises, you will gradually add new features (to better integrate with the Windows 10 ecosystem) and you will migrate the existing ones in a way that will allow you to convert your app to a native UWP one, giving you accesso to all the benefits of the new platform (like the capability of running the same app on multiple devices, like computers, phones, Xbox One, etc.).

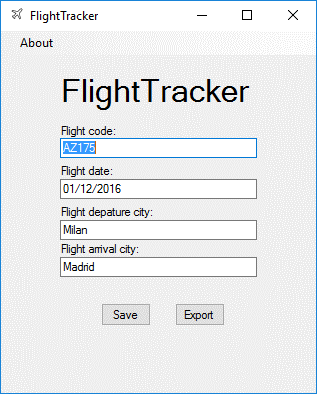
## The project

This is how the starting project looks like:

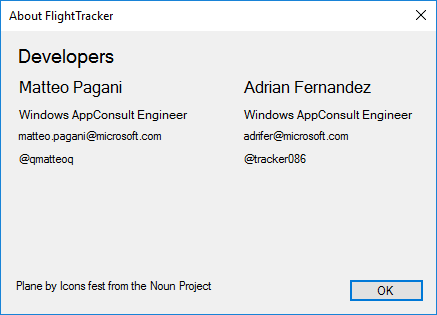


It’s a standard Windows Forms app, made by two windows:

1. The main one, stored in the **Flight.cs** file, which contains a form where the user can fill the data about the flight he wants to track and perform the various operations.

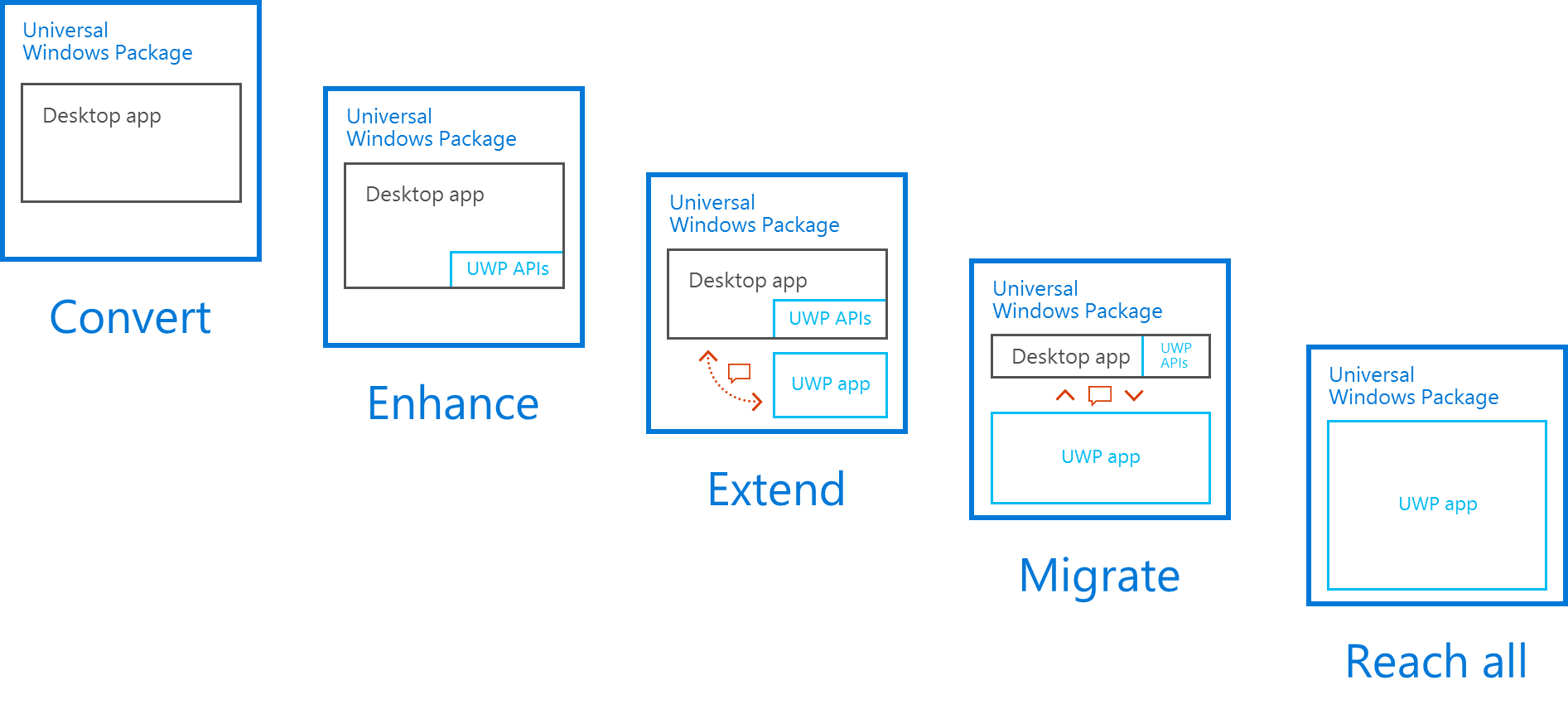


1. An about page, stored in the **AboutWindow.cs** file, which is accessible from the About menu, and displays some generic info about the app and the developers.



# The exercises

The exercises will cover all the five steps of the Desktop Bridge journey, which are highlighted in the following image:



1. **Convert:** in this step, you have the option to choose among different approaches (automatic with many tools, manual, etc.) to convert your desktop as it is in an app package, which is deployed as a Universal Windows Package.
2. **Enhance:** in this step, you can start modernizing your application by adding some specific Windows 10 features (like push notifications or live tiles). Thanks to the fact that the converted application is running inside the UWP container, in fact, you have access to most of the APIs included in the Universal Windows Platform, despite it’s based on a different runtime.
3. **Extend:** in this step, you can make a step forward in the modernizing process, by adding not just UWP APIs but components that belong to the new ecosystem, like background tasks or app services.
4. **Migrate:** in this step, you’ll start to see a different scenario. The desktop app isn’t anymore the protagonist, but a sidekick: the migration path will lead you to create a real UWP app that, when it’s running on the desktop, can invoke a Win32 process, so that you can perform operations that aren’t supported yet by the Universal Windows Platform or leverage an existing codebase which would be too complex to migrate for the moment.
5. **Reach all:** it’s the last, where your app has been completely migrated to the Universal Windows Platform. As such, the same binary package will be able to run on every platform supported by Windows 10: not just desktop computers, but also mobile phones, Xbox One, Surface Hub, Hololens or IoT devices like a Raspberry PI.

# Exercise 1 – Convert

The conversion process is the simplest step to get your desktop application converted into an app package and to automatically leverage many of the benefits of the Universal Windows Platforms, like a more streamlined and clean deployment / update / uninstall process. With this approach, your application will run as it is (so like a traditional desktop application), but it will benefit of the advantages of the application model of the Universal Windows Platform. There are many available options:

1. Using the Desktop App Converter, which is a tool developed by Microsoft based on PowerShell, which can convert into an app package either an installer or a folder with a self-executable application.
2. Using a manual approach, which means that, side by side with your traditional desktop app, you take care of creating all the companion files required by an app package (like a manifest file, the visual assets, the resources to support the various Windows scaling factors, etc.)
3. Using Visual Studio 2017, in combination with a WinJS projects (which helps to generate an app package starting from a classic desktop app developed with a Microsoft technology) and a special extension called Desktop Bridge Debugging Project (which helps to debug the converted version of the app, using the same code of the original desktop app).
4. Use a third-party tool: many applications to generate installers (like InstallShield, Wix, Advanced Installer, InstallAware, etc.) now supports the option to take an existing setup project and export it directly as an app package, in addition to a classic installer (like a MSI file).

In this exercise, we will focus on point 1, which is the most common scenario when you already have a desktop app, which might be developed with a non-Microsoft technology. Let’s not forget, in fact, that the Desktop Bridge works with any technology, not just with applications developed using Microsoft tools and platforms.

# The Desktop App Converter

To automatically convert an installer into an AppX package, the official option supported by Microsoft is called [Desktop App Converter](https://msdn.microsoft.com/en-us/windows/uwp/porting/desktop-to-uwp-run-desktop-app-converter), which is a command line utility based on a set of PowerShell scripts that can be downloaded for free from the Store. The tool can work in two ways:

1. You can start from an existing installer (which must use the MSI format or a custom one, but with support for silent installation, which means that no user intervention should be required). In this case, your computer must satisfy two requirements:
   1. You must have enabled the Container Windows feature, available starting from Windows 10 Anniversary Update in the Pro or Enterprise editions
   2. You must have a base image (which is, basically, a clean image of the operating system) which must match the same version of Windows you’re using. You will find all the available base images at the URL: <http://aka.ms/converterimages>

The reason of these requirement is that, in this scenario, Windows will start a container with a clean version of the operating system, it will run the setup inside the container, it will capture all the changes made by the installer (like copied files and folders, keys created in the system registry, etc.) and it will produce an app package as output that, when it’s deployed, will perform the same deployment operations executed by the traditional installer.

1. You can start from a folder which contains a self-executable application, which means that you can simply double click on the executable and the app will start, without the requirement of performing a real installation. In this case, the previously outlined requirements (the Container feature and the base image) aren’t needed, since the tool will simply take care of adding, to the desktop app, the missing files which are required to create a real app package (like the manifest, the assets, etc.).

You can learn more about using the Desktop App Converter in the official documentation: <https://msdn.microsoft.com/en-us/windows/uwp/porting/desktop-to-uwp-run-desktop-app-converter>

## Task1: Desktop App Converter with an installer

In this first exercise, we’re going to learn how to convert an installer using the Desktop App Converter. The first step is configuring it (assuming that you have already installed Visual Studio 2017 with the Windows 10 SDK, as specified in the initial requirements), by following these steps:

1. Download the tool from the Store using the following link: <https://www.microsoft.com/store/apps/9NBLGGH4SKZW>
2. Download the base image which matches your Windows installation from <http://aka.ms/converterimages>:
   1. If you have Windows 10 Anniversary Update, you need to download the file called **BaseImage-14393.wim**.
   2. If you have Windows 10 Creators Update, you need to download the file called **BaseImage-15063.wim**.
3. Run the Desktop App Converter with administrative rights: look for it in the Start menu, right click on the icon and choose **More -> Run as administrator**.
4. Run the following command first, which will allow to execute any PowerShell script:

Set-ExecutionPolicy bypass

1. Then run the following command to setup the converter and the base image:

DesktopAppConverter.exe -Setup -BaseImage "BaseImage-1XXXX.wim" -Verbose

In the -BaseImage parameter you will have to pass the full path of the base image you have previously downloaded in step 2.

The process may take a while and if you haven’t previously enabled the Container feature in Windows, it will require a reboot: the Container feature will be activated and, then, after the reboot, the base image expansion and setup will continue.

Once the operation is completed, you are ready to convert the installer of the Flight Tracker application.

## Task 1: Convert the installer

The starting point to convert the installer is the Desktop App Converter tool you have previously installed which, also in this case, must be executed with administrative rights.

In the folder that contains the workshop, you will find the traditional installer of the Flight Tracker application in the following path: **\01-Convert\Start\Binaries\Installer\FlightTracker.msi**

Copy the MSI installer on your machine and run the following command (for this sample, we’re assuming that you have copied the installer in a folder called *C:\FlightTracker* and that you want to get the output of the conversion in the *C:\FlightTracker-AppX* folder):

DesktopAppConverter.exe -Installer "C:\FlightTracker\FlightTracker.msi" -Destination "C:\FlightTracker-AppX" -PackageName "FlightTracker" -Publisher "CN=AppConsult" -Version "1.0.0.0" -MakeAppx -Sign -Verbose

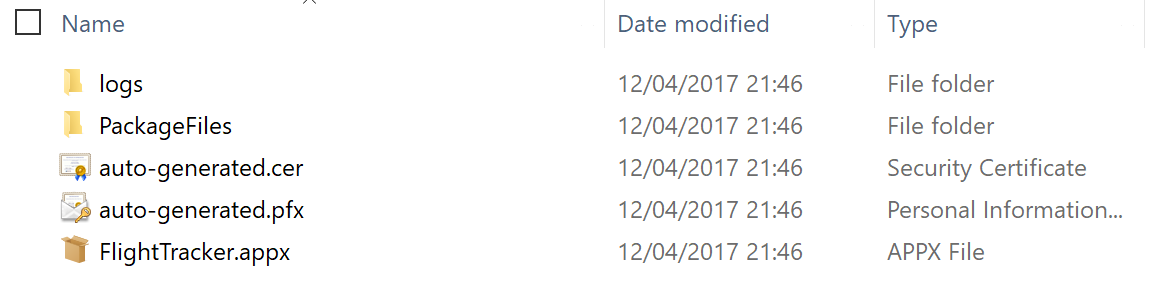
Here is the meaning of the various parameters:

* **-Installer** is the path to the installer
* **-Destination** is the path to the folder where you want to generate the output of the Desktop App Converter
* **-PackageName** is the name of the package you want to generate
* **-Publisher** is the publisher’s name. Typically, it matches the value assigned by the Dev Center (in case of an app that will be published on the Store) or the subject of the digital certificate that will be used to sign the package (in case of sideloading or enterprise distribution).
* **-Version** is the version number of the app. This parameter is important to handle updates in the proper way (the same package but with a higher version number will be installed as an update of the existing app, rather than as a new app).

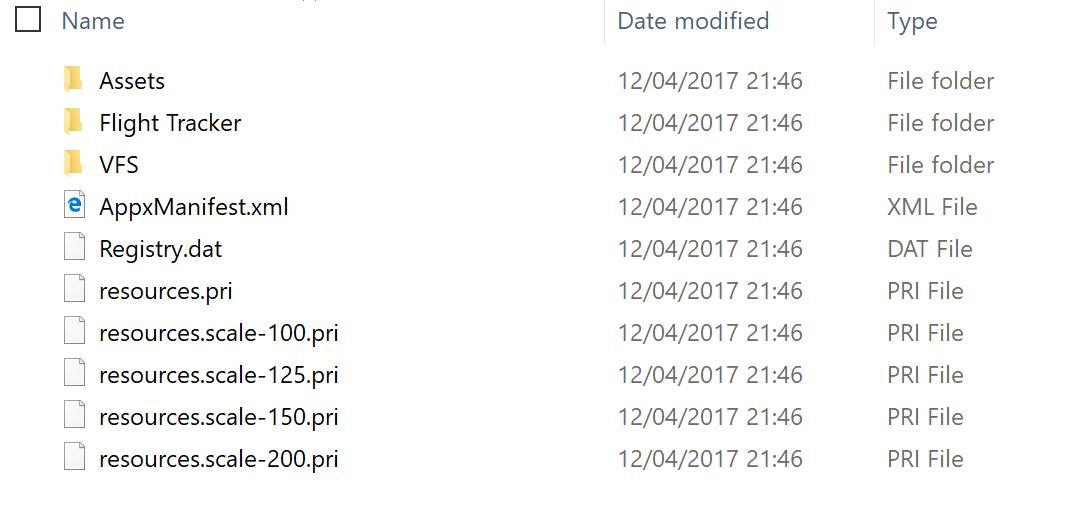
These are the basics parameter you need to specify: as output, you will get in the destination folder a subfolder called **PackageFiles**, which contains the app package of your application (the desktop application plus all the other files that are required by a converted app, like the manifest or the visual assets). Additionally, to make easier for you to test the application and to understand the status of the process, you can use some additional optional parameters:

* **-MakeAppx**, which will automatically generate an AppX file starting from the app package folder.
* **-Sign**, which will generate a test certificate and will sign the AppX with it.
* **-Verbose**, which will display a detailed log of the conversion process.

If the conversion has completed successfully, you will find in the path *C:\FlightTracker-AppX\FlightTracker* the following content:



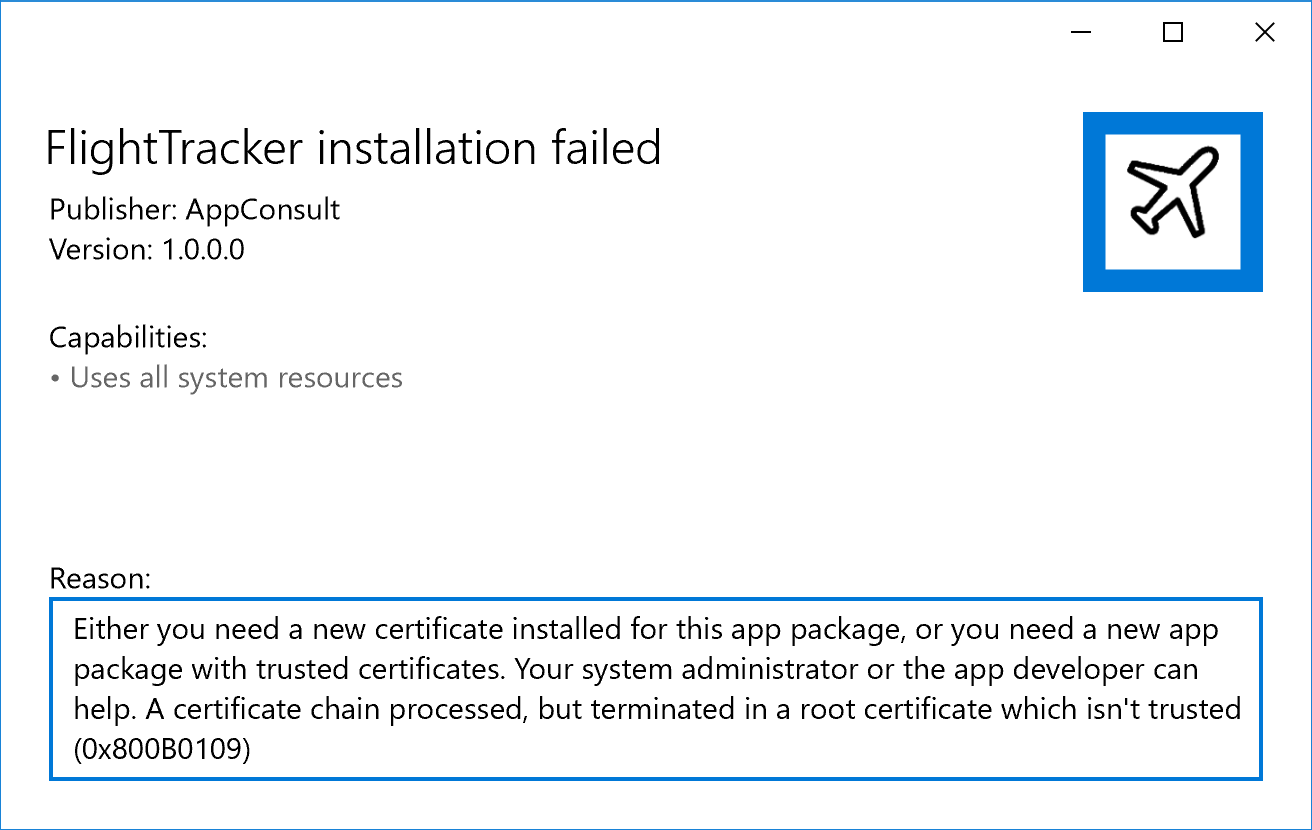
The **PackageFiles** folder is the one that contains the real output of the conversion, which are the files that compose the app package:



Here are the key components:

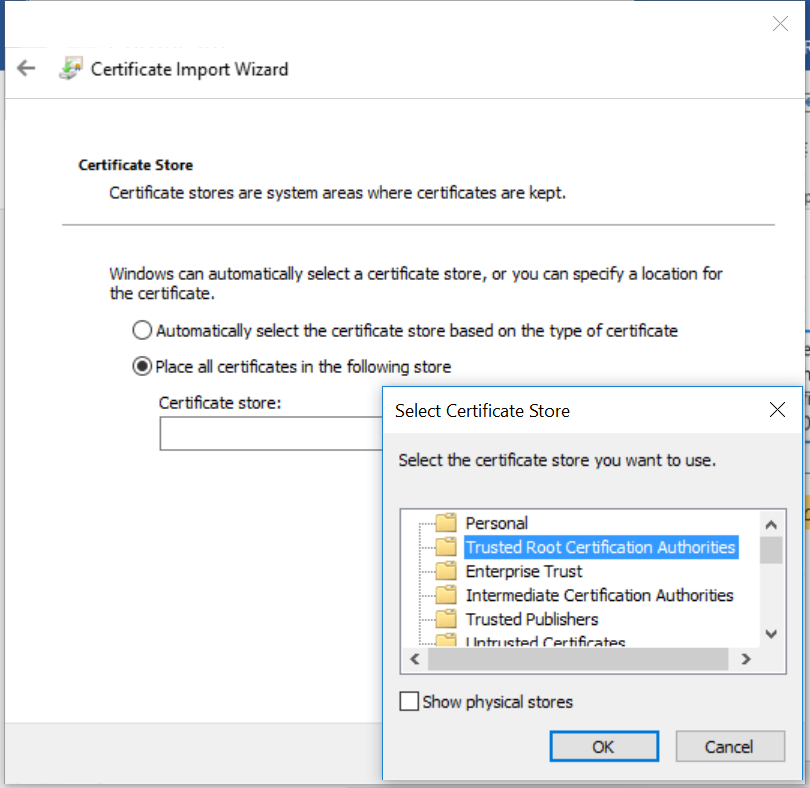
1. **Assets** is the folder that contains all the visual elements of the application, used in the Start menu, in the taskbar icon, for the tile, etc. By default, the image is automatically extracted from the icon of the app, and the tool will generate an image for each supported Windows scaling factor, so that that the icon will always look good, no matter which are the resolution, the screen size and the DPIs of the computer where the app is running.
2. **Flight Tracker** is the folder where the original desktop app is stored.
3. **VFS** is the virtual file system, a special folder that maps all the Windows folders. When the app will look for a dependency (like a framework or a library), it will look for it inside the package and not in the system. This allows to solve the problem known as “DLL Hell”, where multiple applications depends from the same framework and changing it (because, for example, another application requires a newer version) may break them.
4. **AppxManifest.xml** is the manifest file, which describes all the features of the application like:
   1. The entry point
   2. The assigned identity
   3. The visual assets
   4. The optional extensions and capabilities leveraged by the app
5. **Registry.dat** is the binary file where all the changes made to the registry by the installer are stored.
6. **A set of .pri files**, one for each supported scaling factor. The purpose of these files is to optimize the deployment of the package: the user will get only the set of images that match the scaling factor assigned by Windows to the computer, to avoid downloading visual assets that will never be used.

Since we have passed to the command both the **-MakeAppx** and the -**Sign** parameters, you will find in the *C:\FlightTracker-AppX\FlightTracker* also an AppX file, signed with the **auto-generated.pfx** certificate and ready to be installed. However, if you try to double click on the **FlightTracker.appx** and you click on the **Install** button, you will get the following error:



The reason is that the app package is signed with a valid certificate, but it’s not trusted by our machine. As such, we need first to add the file **auto-generated.cer** (which has been generated together with all the other files by the Desktop App Converter) to the **Trusted Root Certification Authorities** store of our computer. The steps to achieve this goal are:

1. Double click on the **auto-generated.cer** file.
2. Choose **Install certificate.**
3. Select, as Store Location, the option **Local machine**, then press **Next**.
4. In the following window, choose the option **Place all certificates in the following store** and press **Browse.**
5. In the new window, choose **Trusted Root Certification Authorities**, then press **Ok** and, when you’re back to the previous window, choose **Next.**
6. Press **Finish** and complete the process. A popup will inform you that the import has completed successfully.



Now try again to double click on the **FlightTracker.appx** file and to press the **Install** button. This time the installation will complete just fine and, once the process is finished, the **Install** button will turn into a **Launch** button, which will allow you to launch the converted version of your Desktop app out of the box.

Additionally, you will notice that, from a visual point of view, the application will act like a Universal Windows Platform App:

1. In the Windows Start menu, under the F letter, you will find an entry with an icon for the Flight Tracker application.
2. If you right click on the icon, you’ll be able to choose the **Pin to start** option. However, unlike a regular desktop shortcut, you will have access to the **Resize** menu and choose between multiple tile formats (Small, Medium, Wide and Large), while a classic desktop application can only use the Small and Medium sizes.

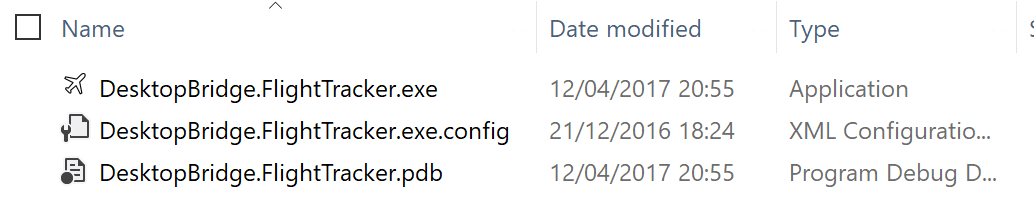
Another benefit of the converted version is that the uninstall process will be much cleaner since it will be performed out of the box by Windows: just right click on the icon in the Start menu, choose **Uninstall** and all the files that belong to the application will be automatically deleted, without any leftover.

Last but not the least, you should have noticed how we didn’t have to take care of authoring a setup (like creating a MSI installer with a 3rd party tool) to allow the user to install the application: it has been enough to create an AppX, then it’s up to Windows to take care of all the deployment / uninstall phase.

## Task 2: Convert the self-executable application

In this second exercise, you’re going to achieve the same goal of the first task, but without starting from an installer but from a portable application: a folder which contains an executable with all its dependencies that can simply run without being installed. It’s the typical scenario of the output you may get from a Visual Studio project, and this is exactly our case: we’re going to leverage, as a starting point, the output of the build of the original Windows Forms project.

You will find this output in the folder *\01-Convert\Start\Binaries\Portable* of the workshop and its content is very simple:



Copy the content of the *Portable* folder on your C: drive in a folder called *FlightTracker*, so that you will end up in having a duplicate in the *C:\FlightTracker* path.

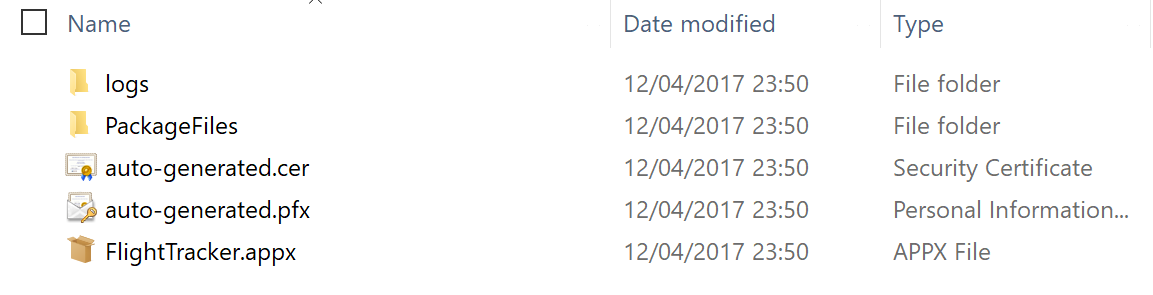
The first step, also in this case, is to run the Desktop App Converter with administrative rights, like we did in the first task. However, in this case, the command we’re going to execute will be slightly different:

DesktopAppConverter.exe -Installer "C:\FlightTracker" -AppExecutable "DesktopBridge.FlightTracker.exe" -Destination "C:\FlightTracker-AppX" -PackageName "FlightTracker" -Publisher "CN=AppConsult" -Version "1.0.0.0" -MakeAppx -Sign -Verbose

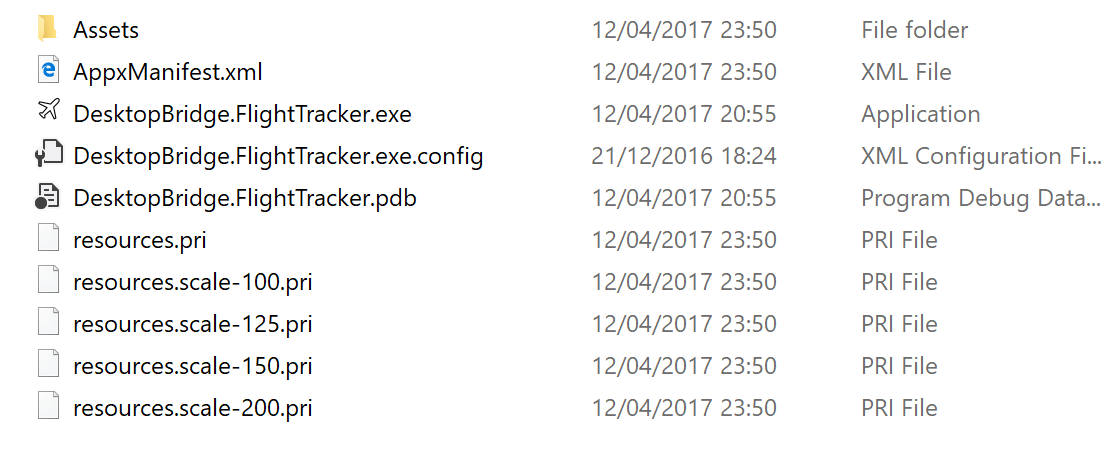
In yellow I have highlighted the two differences compared to the command that we have executed in the first task:

1. The **-Installer** parameter doesn’t point anymore to an installer, but to the folder that contains our self-executable application.
2. When the app is converted starting from an installer, the Desktop App Converter can automatically detect which is the main executable from the shortcut that the installer creates in the Start menu and / or on the desktop. In this case, we don’t have any setup process, so we must specify, using the **-AppExecutable** parameter, which is the main process that should be launched when you start the converted app.

When we run this command, the output we will find in the *C:\FlightTracker-AppX* folder of our computer will be like the one we got in Task 1:



You can spot some differences only if you open the **PackageFiles** subfolder:



Since, in this case, there isn’t an installation process performed inside a Container, the Desktop App Converted doesn’t capture any change but it just takes care of adding, to the original desktop app, the files that can turn it into an app package, like the manifest, the Assets folder (again, with the images extracted from the main icon of the app and already optimized for the different Windows scaling factors) and the resources files. There won’t be any **registry.dat** file (since there are no keys created in the system registry) or any **VFS** folder (since there isn’t any framework or library installed as a dependency).

The rest of the exercise can continue like the first task: if you want to try the converted version to the application, you will have first to install the **auto-generated.cer** certificate in the **Trusted Root Certification Authorities** store of your computer, then double click on the **FlightTracker.appx** package and press the **Install** button.

## Task 3: Update your app and generate a new package

Once you have generated a converted version of your app using the Desktop App Converter, it can become the starting point every time you want to release an updated version of the package. You are free, in fact, for example:

1. To change the images in the Assets folder, if you want to use a different image than the one extracted from the app icon.
2. Make changes to the app manifest, for example to leverage [one of the special extensions offered by the Desktop Bridge](https://docs.microsoft.com/en-us/windows/uwp/porting/desktop-to-uwp-extensions).
3. Update the executable of a DLL of your desktop app, in case you have made any change to the source code.

No matter which is your scenario, you will have to generate a new AppX file starting from the updated content of the **PackageFiles** folder to distribute it, either by sideloading, by using enterprise tools or by publishing it on the Store. However, for this scenario the Desktop App Converter isn’t useful any more: you aren’t starting from an installer, but from an already existing app package, that you have simply updated. For this purpose, the Windows 10 SDK offers a set of tools that can help you to:

1. Generate an AppX starting from an app package folder.
2. If the app isn’t meant for Store distribution, sign it with a digital certificate.

Let’s see how we can do this by using, as example, the option to add [one of the special Desktop Bridge extensions.](https://docs.microsoft.com/en-us/windows/uwp/porting/desktop-to-uwp-extensions)

We’re going to add a very simple one, just to show you how to perform the repacking: it’s the one called [App Alias](https://docs.microsoft.com/en-us/windows/uwp/porting/desktop-to-uwp-extensions#alias), which allows to assign a name to your application that is automatically registered in the global Windows path. This means that, no matter in which folder you are, or which shell you’re using (the Start menu, Cortana, the Run dialog), you will be able to launch your app simply by typing this alias.

To add the alias, you have first to open the file called **AppxManifest.xml** stored in the **PackageFiles** folder with a text editor. The first step is to add, in the <**Package>** definition, two new custom namespaces:

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

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

Then, you need to add them in the **IgnorableNamespaces** attribute, like in the following sample:

IgnorableNamespaces="uap3 desktop"

The last step is to add the special extension, which must be declared as a child of the **<Application>** node, like in the following sample:

<Extensions>

<uap3:Extension

Category="windows.appExecutionAlias"

Executable="DesktopBridge.FlightTracker.exe"

EntryPoint="Windows.FullTrustApplication">

<uap3:AppExecutionAlias>

<desktop:ExecutionAlias Alias="FlightTracker.exe" />

</uap3:AppExecutionAlias>

</uap3:Extension>

</Extensions>

All the values of the attributes are fixed, except for:

1. **Executable**, which must point to the main executable of the app (in this case, the file **DesktopBridge.FlightTracker.exe**).
2. **Alias** of the **<ExecutionAlias>** tag: it’s the name of the alias we want to assign to the main executable.

The last step is to increase the version number of the app, so that Windows can detect that the package we’re going to create is an update and it will install it over the existing application, but keeping at the same time all the existing local files. We can do this by changing the **Version** attribute of the **<Identity>** entry of the manifest, like in the following sample:

<Identity Name="FlightTracker" ProcessorArchitecture="x64" Publisher="CN=AppConsult" Version="1.1.0.0" />

To recap, this is how the final version of the manifest should look like after all the changes we have made:

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

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

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

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

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

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

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

IgnorableNamespaces="uap uap2 uap3 rescap desktop">

<Identity Name="FlightTracker" ProcessorArchitecture="x64" Publisher="CN=AppConsult" Version="1.1.0.0" />

<Properties>

<DisplayName>FlightTracker</DisplayName>

<PublisherDisplayName>AppConsult</PublisherDisplayName>

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

</Properties>

<Resources>

<Resource Language="en-us" />

<Resource uap:Scale="100" />

<Resource uap:Scale="125" />

<Resource uap:Scale="150" />

<Resource uap:Scale="200" />

<Resource uap:Scale="400" />

</Resources>

<Dependencies>

<TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.14393.0" MaxVersionTested="10.0.15063.0" />

</Dependencies>

<Capabilities>

<rescap:Capability Name="runFullTrust" />

</Capabilities>

<Applications>

<Application Id="FlightTracker" Executable="DesktopBridge.FlightTracker.exe" EntryPoint="Windows.FullTrustApplication">

<uap:VisualElements DisplayName="FlightTracker" Description="FlightTracker" BackgroundColor="transparent" Square150x150Logo="Assets\AppMedTile.png" Square44x44Logo="Assets\AppList.png">

<uap:DefaultTile Wide310x150Logo="Assets\AppWideTile.png" Square310x310Logo="Assets\AppLargeTile.png" Square71x71Logo="Assets\AppSmallTile.png">

<uap:ShowNameOnTiles>

<uap:ShowOn Tile="square150x150Logo" />

<uap:ShowOn Tile="wide310x150Logo" />

<uap:ShowOn Tile="square310x310Logo" />

</uap:ShowNameOnTiles>

</uap:DefaultTile>

</uap:VisualElements>

<Extensions>

<uap3:Extension

Category="windows.appExecutionAlias"

Executable="DesktopBridge.FlightTracker.exe"

EntryPoint="Windows.FullTrustApplication">

<uap3:AppExecutionAlias>

<desktop:ExecutionAlias Alias="FlightTracker.exe" />

</uap3:AppExecutionAlias>

</uap3:Extension>

</Extensions>

</Application>

</Applications>

</Package>

Now that we have made these changes, we are ready to prepare a new package. To achieve this goal, you need to have the Windows 10 SDK installed on your machine which, however, will come preinstalled with your Visual Studio 2017 installation if you have added the tools to develop for the Universal Windows Platform during the setup. The easiest way to get access to the tools we need is to open the Start menu, look for a folder with name **Visual Studio 2017** and launch a special command prompt, called **Developer command prompt for VS 2017**. It will look like a regular command prompt but, under the hood, it’s already configured to include, as environment variables, the Windows 10 SDK folders.

At first, move to the folder where you have generated the output of the Desktop App Converter, with the following command:

cd C:\FlightTracker-AppX\FlightTracker

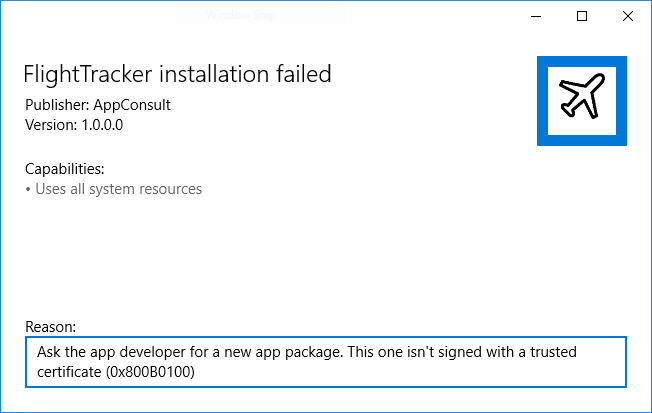
Then the first step is to turn the **PackageFiles** folder into an AppX file, by using a tool called **makeappx**. Here is the command to execute:

makeappx pack -d "PackageFiles" -p "FlightTracker.appx" -l

Here is the meaning of the parameters:

* **-d** is the path of the folder that contains the files that compose the app package
* **-p** is the path and name of the AppX file you want to generate
* **-l** is a parameter that tells the tool to ignore if the manifest file contains a reference to visual assets that don’t exist. The reason is that, in the manifest, you typically specify only the base name of the image (like **Logo.png**) but then, due to the Windows 10 scaling factor support, in the **Assets** folder you won’t find any file called **Logo.png**, but multiple files for each supported scaling factor (like **Logo-scale100.png**, **Logo-scale200.png**, **Logo-scale400.png**, etc.)

The result will be a new file called **FlightTracker.appx**. If your goal is to publish the app on the Windows Store or on the Windows Store for Business, you can stop here, since the Dev Center will take care of signing the package for you. However, if you would like to test the new package on your machine and you would try to install it, you’ll get a bad surprise:



As already previously mentioned, from this point of view there is no difference between an app package and a regular installer: Windows will refuse to install it if it isn’t signed with a trusted certificate.

To sign the package, we can reuse the same **auto-generate.pfx** certificate that has been previously generated by the Desktop App Converted in combination with another tool offered by the Windows 10 SDK, called **signtool**.

Here is the command that we can execute to sign the package:

signtool.exe sign /a /v /fd SHA256 /f "auto-generated.pfx" /p "123456" "FlightTracker.appx"

I will explain only the most important parameters of the command, which aren’t fixed but they may change in other scenarios:

* **/f** is the parameter used to specify the path and the filename of the certificate you want to use to sign the package
* A certificate can be password protected. In case of certificates generated by the Desktop App Converter, they are protected by the password **123456**. This is the meaning of the **/p** parameter. If the certificate isn’t password protected, you can just omit this parameter.
* The last parameter passed to the command is the path and the name of the AppX file you want to sign.

If you did everything correctly, the signing operation will complete successfully and, this time, you’ll be able to install the updated **FlightTracker.appx** since:

1. We have increased the version number, so it will be detected as an update, and it won’t conflict with the version we have installed before.
2. The package is signed with a certificate that we have already installed in one of the previous exercises, so it’s already trusted.

Now that the new version is installed, you can easily test how the new alias that we have added thanks to the special extension is working. In the same command prompt we have just used to create and sign the package, just type:

flighttracker

and you’ll see how, despite you have launched the command in a folder completely different from the one where the app has been installed, the converted version of Flight Tracker will start.

**Note**: in this case, to keep the exercise simple, we have reused the same certificate that the Desktop App Converter has generated during the creation of the app package. However, you have the option also to create your own custom certificate, if you prefer, and use it for signing the app. In this case, you’ll need to use another set of tools (called **makecert** and **pvk2pfx**), which can generate for you a pfx file (the one you need to sign the package) and a cer file (the one you need to install on your machine or distribute to every computer where you want to manually install the converted application). [The complete procedure is described in the official documentation](https://docs.microsoft.com/en-us/windows/uwp/porting/desktop-to-uwp-signing).

## Publish a converted desktop application on the Store

If you try to publish a converted desktop application on the public Window Store, you will get an error in the Packages section of the submission, as soon as you try to upload the AppX file you have generated. The reason can be found if you analyze the **AppxManifest.xml** file: you will discover, in the **Capabilities** section, that a converted desktop app uses a special capability called **runFullTrust**, which is the one that allows it to still act as a regular Win32 application, despite it’s running as a Universal Windows Package.

For security and legal reasons, it’s a restricted capability; as such, a regular developer account isn’t allowed to submit applications that make use of it. If you want to publish your converted desktop app on the Store you will need [to fill the following form](https://developer.microsoft.com/en-us/windows/projects/campaigns/desktop-bridge): you will be reached by a Microsoft engineer, who will help you to solve any technical blocker and will work with you to give to your account the required permissions.

# Exercise 2 – Enhance

The Enhance phase of the Desktop Bridge is helpful to start modernizing your application without having to rewrite it as a full Universal Windows Platform app. From a technical point of view, it means that, even if your application is based on a different platform (like Flight Tracker, which is developed with Windows Forms and it’s based on the .NET Framework), you’ll be able to invoke some of the APIs that are part of the Universal Windows Platform, despite being a completely different development runtime.

In this exercise, we’re going to improve the user experience of our application by offering a better integration with Windows 10: when the user generates a boarding pass, instead of just displaying a message in the form like in the original version of the app, we’re going to display a toast notification, which will be much more visible and, additionally, it will be stored in the Action Center. This way, even if the user was distracted when the export operation was completed, he will have the chance to retrieve the information.

## Task 1 – Set up Visual Studio 2017

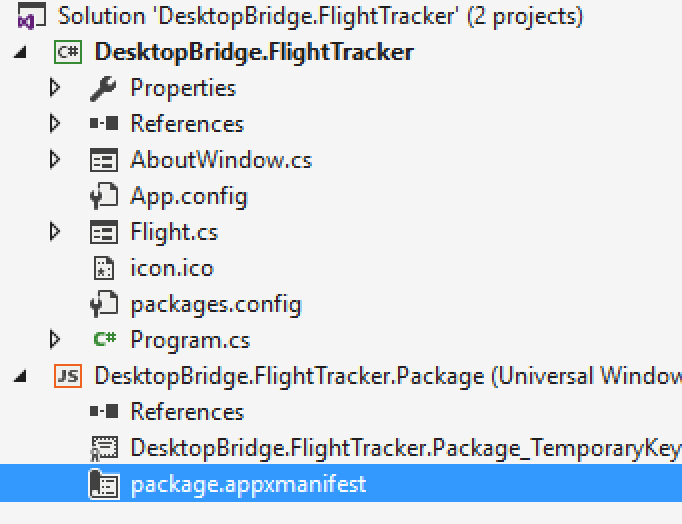
The first step to make our life easier in working with a desktop app that needs to be enhanced is to setup the solution using some of the tools provided by Visual Studio 2017 so that, other than doing the required code changes to integrate with the Universal Windows Platform, we can easily debug the code, generate an app package, edit the manifest, etc. These are all operations that, in Exercise 1, we made manually using command line tools and that now, instead, we’ll be able to do in a visual way.

Let’s start by opening the solution that contains the Windows Forms project: it’s the file called **DesktopBridge.FlightTracker.sln** and it’s stored in the path *02-Enhance\01-Start* of the lab.

Once we have opened Visual Studio, the next step is to add a WinJS project to our solution. If you’re a C# developer, you may be tempted to close the document and immediately quit the workshop 😊 Don’t worry, you won’t have to write a single line of HTML / CSS / Javascript code. The reason why we’re using a WinJS project is that it acts as an intermediate between the original desktop version and the converted version: unlike a Universal Windows Platform app built with C# and XAML, in fact, there is no compilation involved in a WinJS project. It will simply be a gateway to make easier to edit the manifest, to generate app packages for publishing on the Store or sideloading, etc.

Let’s move on by right clicking on the solution and by choosing **Add -> New Project**. You’ll find the WinJS project under the section **Javascript -> Windows Universal -> Blank App (Universal Windows)**. Call it **DesktopBridge.FlightTracker.Package,** thenchoose the target and minimum SDK (you’re free to keep using the Anniversary Update one, identified by the build number 14393, since during this lab we won’t use any new feature added in the Creators Update) and then we can start to do some clean up. As already mentioned, we need the WinJS project only as a gateway for our converted desktop app, it won’t be a real Universal Windows App built with WinJS, so we can delete all the files and folders that are added by default to the project: **images**, **lib**, **css**, **js** and the **index.html** file. You should leave only the temporary certificate generated by Visual Studio and the manifest, which is the **package.appxmanifest** file.

Here is how your solution should look like after the cleanup:



The second step is to change the manifest: if you right click on it and you choose the option **View code**, you’ll notice that it’s quite different from the one we’ve seen in Exercise 1 since it refers to a Universal Windows Platform app and not to a Desktop Bridge one. The easiest way to make our manifest compliant, since we have already converted our application in the Exercise 1, is to take the manifest file the Desktop App Converter has produced and use it to replace the whole content of the **package.appxmanifest** file. The only exception is that we need to keep the following namespace:

xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"

and the following entry:

<mp:PhoneIdentity PhoneProductId="ad8f83e7-c68a-43c7-90f3-a66cdde6b623" PhonePublisherId="00000000-0000-0000-0000-000000000000" />

even if a converted desktop app will never run on a phone. The reason is that, without this entry, the manifest won’t be recognized as valid and we would get a compilation error trying to build and deploy the WinJS project.

By reusing the same manifest of the exercise 1, this is how the final **package.appxmanifest** should look like:

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

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

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

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

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

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

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

xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"

IgnorableNamespaces="uap uap2 uap3 rescap desktop mp">

<Identity Name="FlightTracker" ProcessorArchitecture="x64" Publisher="CN=AppConsult" Version="1.1.0.0" />

<mp:PhoneIdentity PhoneProductId="ad8f83e7-c68a-43c7-90f3-a66cdde6b623" PhonePublisherId="00000000-0000-0000-0000-000000000000" />

<Properties>

<DisplayName>FlightTracker</DisplayName>

<PublisherDisplayName>AppConsult</PublisherDisplayName>

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

</Properties>

<Resources>

<Resource Language="en-us" />

<Resource uap:Scale="100" />

<Resource uap:Scale="125" />

<Resource uap:Scale="150" />

<Resource uap:Scale="200" />

<Resource uap:Scale="400" />

</Resources>

<Dependencies>

<TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.14393.0" MaxVersionTested="10.0.15063.0" />

</Dependencies>

<Capabilities>

<rescap:Capability Name="runFullTrust" />

</Capabilities>

<Applications>

<Application Id="FlightTracker" Executable="DesktopBridge.FlightTracker.exe" EntryPoint="Windows.FullTrustApplication">

<uap:VisualElements DisplayName="FlightTracker" Description="FlightTracker" BackgroundColor="transparent" Square150x150Logo="Assets\AppMedTile.png" Square44x44Logo="Assets\AppList.png">

<uap:DefaultTile Wide310x150Logo="Assets\AppWideTile.png" Square310x310Logo="Assets\AppLargeTile.png" Square71x71Logo="Assets\AppSmallTile.png">

<uap:ShowNameOnTiles>

<uap:ShowOn Tile="square150x150Logo" />

<uap:ShowOn Tile="wide310x150Logo" />

<uap:ShowOn Tile="square310x310Logo" />

</uap:ShowNameOnTiles>

</uap:DefaultTile>

</uap:VisualElements>

<Extensions>

<uap3:Extension

Category="windows.appExecutionAlias"

Executable="DesktopBridge.FlightTracker.exe"

EntryPoint="Windows.FullTrustApplication">

<uap3:AppExecutionAlias>

<desktop:ExecutionAlias Alias="FlightTracker.exe" />

</uap3:AppExecutionAlias>

</uap3:Extension>

</Extensions>

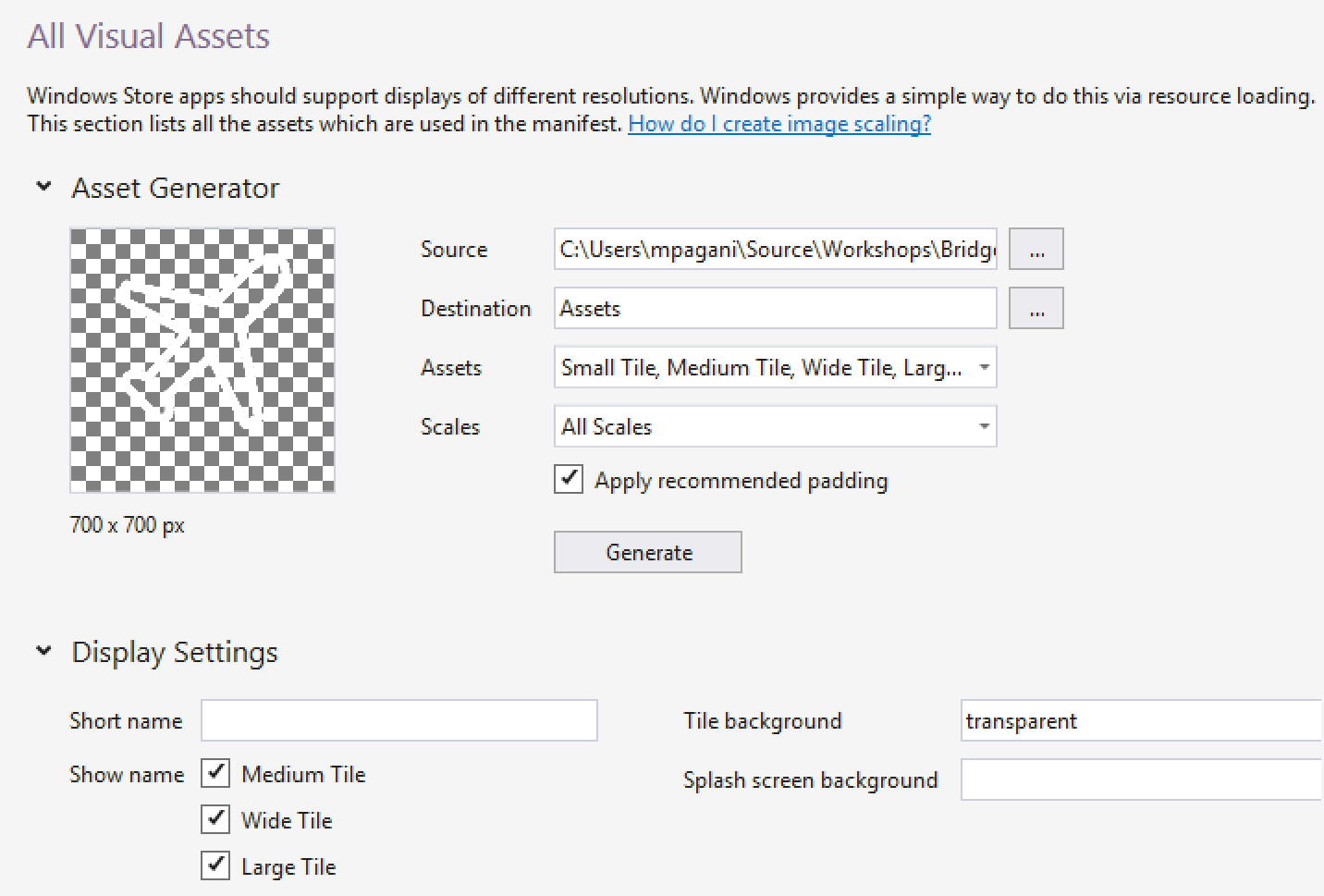
</Application>

</Applications>

</Package>

Now that we have a proper manifest, we can take care of generating the visual assets that, previously, were automatically extracted from the application’s icon by the Desktop App Converter. To make our life easier, we can use a new feature offered by Visual Studio 2017. Double click on the **package.appxmanifest** file, so that it gets opened with the visual manifest editor, and move to the **Visual Assets** section.

Visual Studio 2017 has added a new option to pick up an image (possibly, at the highest possible resolution you have) and to automatically generate, starting from it, all the required assets.

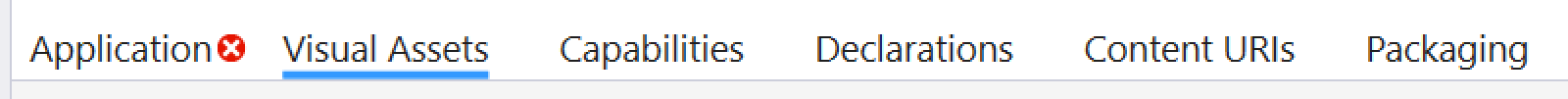


For your convenience, you can find an image that you can use as a logo in the *02-Enhance\01-Start* folder of the lab and it’s called **logo.png.** To generate the assets, follow these steps:

1. Press the button with the three dots near the **Source** field and choose the **logo.png** image.
2. In the **Assets** dropdown:
   1. Uncheck the **Splash Screen** option, since converted apps can’t handle the splash screen image automatically like a regular Universal Windows Platform app.
   2. Check the **Package Logo** option, since it’s the image that will be displayed in the window that is displayed when you manually install the AppX file.
3. Leave all the other options untouched.

Now press the **Generate** button: after a few seconds you will find, inside the WinJS project, a new folder called **Assets**, with all the different versions of the image at the various scaling factors supported by Windows 10.

**Note:** when you will open the manifest with the visual editor, you will notice that it will notify an error in the Application section.



The reason is that the **Start page** field (which should contain the path to the entry page of the UWP app) will be empty because the visual editor doesn’t officially support the Desktop Bridge, so it isn’t able to recognize that the entry point declared in the manifest isn’t a HTML or a XAML page, but a Win32 executable. You can safely ignore it, it won’t cause any problem.

Now we need to include the desktop application inside the WinJS project, so that we can use it to generate a converted version through the Desktop Bridge. To achieve this goal, we can use a special post build task that we can include in the definition of the desktop project. The task will take care, every time we build the desktop application, to copy the output of the build in a subfolder (that we’re going to call **win32**) of the WinJS project. But let’s do it one step at a time.

First, right click on the **DesktopBridge.FlightTracker** project and choose **Unload project**. Once it’s unloaded, if you right click again on it, you will have access to the **Edit DesktopBridge.FlightTracker.csproj** option, which will open the XML file that describes the structure of the project. Move at the end of the file and, before the closing **</Project>** tag, add the following XML code:

<Target Name="AfterBuild">

<PropertyGroup>

<TargetUWP>..\DesktopBridge.FlightTracker.Package\win32\</TargetUWP>

</PropertyGroup>

<ItemGroup>

<Win32Binaries Include="$(TargetDir)\\*" />

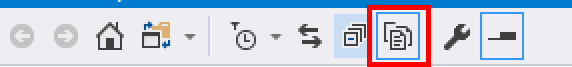
</ItemGroup>

<Copy SourceFiles="@(Win32Binaries)" DestinationFolder="$(TargetUWP)" />

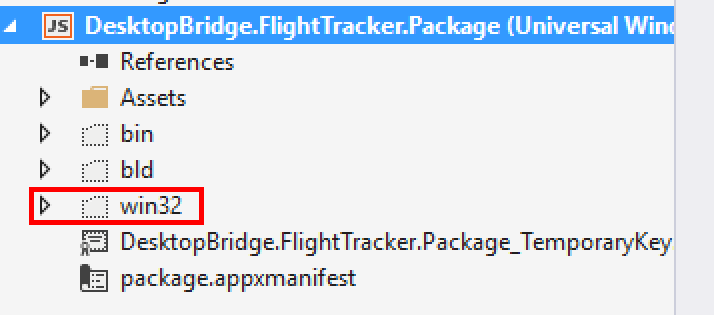
</Target>

Most of the code you see is fixed, the only information that changes based on the project is thevalue of the **<TargetUWP>** node: it must point to the path of a subfolder (the **win32** one we have mentioned before) of the WinJS project. In our scenario, since the WinJS project is placed at the same level of the desktop one, we use the relative path **..\DesktopBridge.FlightTracker.Package\win32\**.

Now right click again on the **DesktopBridge.FlightTracker** project and choose **Reload Project**, then right click again and choose **Build**. Now select the **DesktopBridge.FlightTracker.Package** project and, in Solution Explorer, click on the button to display all the files, even the ones that aren’t part of the project, as highlighted in the below image.

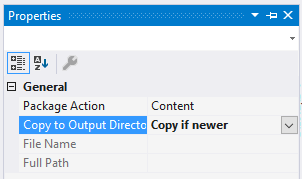


You will find, inside the WinJS project, a new folder called **win32** with the output of the build of the Windows Forms project:



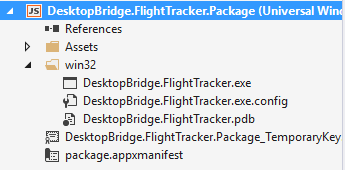
Right click on it and choose **Include in project**. Now you can press again the button in the toolbar, so that you can hide again the files and folders that aren’t part of the project: we have already added the only one we need.

Now the desktop app is included in the app package and, since it’s automatically declared as **Content**, Visual Studio won’t try to compile it, but it will include the various files as they are inside the package. However, there’s one last step to do: by default, these files aren’t recognized as standard files that should belong to a Universal Windows Platform project and, as such, they won’t be included in the output of the build. To change this behavior, we need to expand the **win32** folder, select all the files inside it and open the **Properties** panel of Visual Studio.



The **Package Action**, as you can see, is already set to **Content**, but you need to change the **Copy to Output Directory** parameter to **Copy if newer**. This way every time you’re going to recreate and deploy the app package, the WinJS project will make sure to update also the files that are part of the Windows Forms application, in case you have applied any change to the original source code.

This is how your WinJS project should look like at the end:



Now you can try to build the WinJS project, by right clicking on the **DesktopBridge.FlightTracker.Package** one and choosing **Build**. However, you will get the following exception:

*Manifest references file 'DesktopBridge.FlightTracker.exe' which is not part of the payload.*

The reason is that the original manifest that we have copied from the output of Exercise 1 includes a reference, as main process to launch, to the **DesktopBridge.FlightTracker.exe** file and it assumes that it’s included in the root of the app package. With this new approach, instead, the file is stored in another position: the **win32** folder. As such, we need to right click on the **package.appxmanifest** file, choose **View code** and look for the line which declares the main application entry:

<Application Id="FlightTracker" Executable="DesktopBridge.FlightTracker.exe" EntryPoint="Windows.FullTrustApplication">

We simply need to change the value of the **Executable** attribute, so that it points to the right path, as you can see in the following sample:

<Application Id="FlightTracker" Executable="win32\DesktopBridge.FlightTracker.exe" EntryPoint="Windows.FullTrustApplication">

Remember that, if we have used the manifest that we have created in Exercise 1, you will need to change the path also in the alias we have declared using the special extension:

<Extensions>

<uap3:Extension Category="windows.appExecutionAlias" Executable="win32\DesktopBridge.FlightTracker.exe" EntryPoint="Windows.FullTrustApplication">

<uap3:AppExecutionAlias>

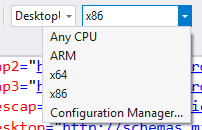
<desktop:ExecutionAlias Alias="FlightTracker.exe" />

</uap3:AppExecutionAlias>

</uap3:Extension>

</Extensions>

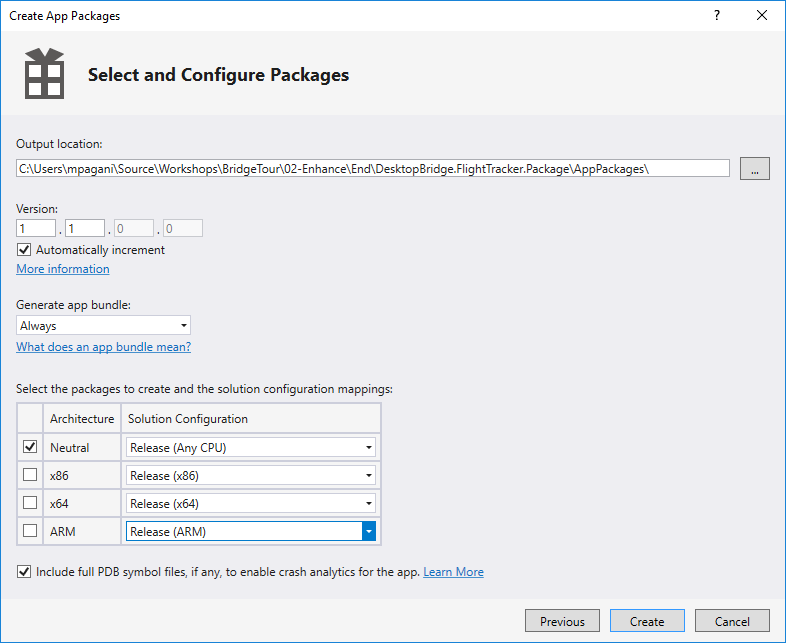
Now we are ready to put the WinJS project at test: first, right click on the **DesktopBridge.FlightTracker.Package** project and choose **Set as startup project**, then make sure that in the **Configuration Manager** dropdown you choose a specific architecture (since Universal Windows Platform apps can’t run with a cross-architecture configuration), like x86 or x64.



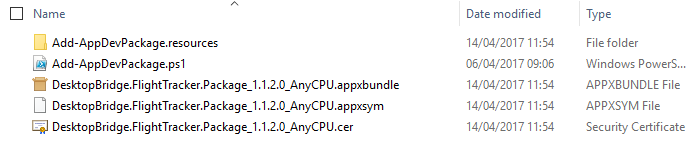
Now right click again on the WinJS project and choose **Build**, then **Deploy.** If you did everything correctly, you will find a Flight Tracker icon in your Start Menu, which will launch the converted version of your desktop app.

Thanks to the WinJS project, many operations that we had to do with a manual command line tool in the first exercise (like generating the app package or signing it), can be executed using visual wizards. For example, let’s say that you want to create an app package to share with your colleagues or to install on other computers. Now it’s enough to:

1. Right click on the **DesktopBridge.FlightTracker.Package** project.
2. Choose **Store -> Create app packages**.
3. Since, at this point, we’re assuming that we want to create a package just for internal distribution and not for the Store, choose **No** as answer of the first question.
4. In the next window, you’ll be able to choose multiple options for the package that you want to create, like the CPU architecture, the configuration mode or if you want to generate a bundle. The bundle is a special package that contains an app package with the base app, with multiple small additional packages for the various architectures and scaling factors. This way, when the user will sideload / download the app from the Store, he will get only the base package plus the specific small package related to the configuration of his device. For example, if you’re using a computer with a x64 CPU, the package compiled for the x86 CPU won’t be acquired. The following screen shows a typical sample configuration:



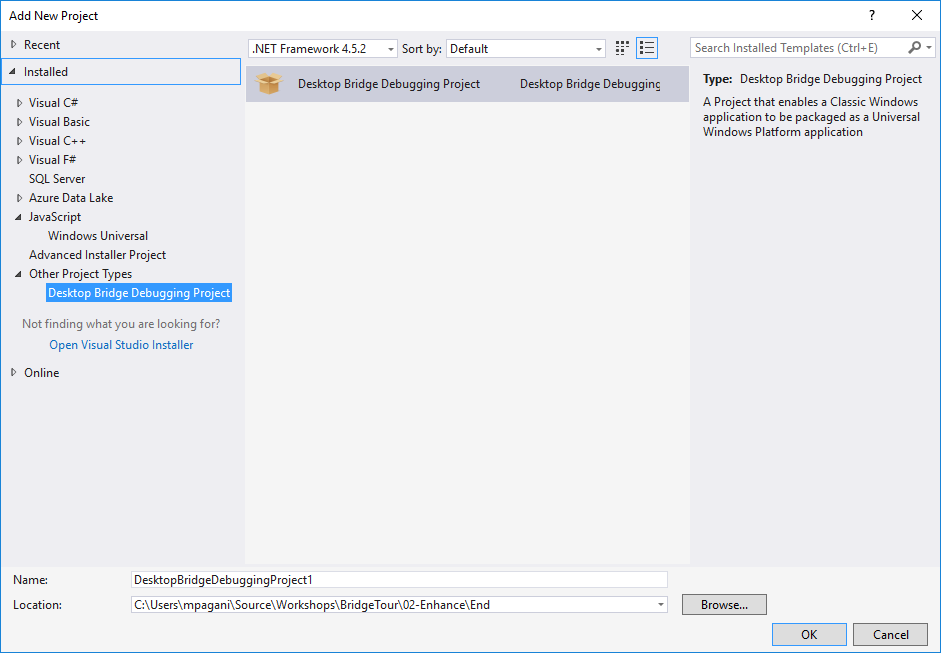
1. Now press **Create** and, at the end of the process, you will get a folder (the one that, in the previous section, was defined in the **Output location** field) with a content like the one you got with the Desktop App Converter in the first exercise: an app package to share, with the related certificate to install as trusted on the target machine.



However, there’s one thing that the WinJS project can’t help us to do: debugging the code. We can use it to generate the packages, deploy the converted version of our app, but we can’t launch a debugging session.

To solve this problem, the Visual Studio team has created a special extension for Visual Studio 2017 called **Desktop Bridge Debugging Project**, that you can download and install from <https://marketplace.visualstudio.com/items?itemName=VisualStudioProductTeam.DesktoptoUWPPackagingProject>

Once you have installed it, right click on the Visual Studio solution, choose **Add -> New project** and you will find a new template in the **Other Project Types** section called **Desktop Bridge Debugging Project.**



Give to it a meaningful name (like **DesktopBridge.FlightTracker.Debug**) and press OK to create it. There are two steps to properly configure it. The first one is to edit the file included in the project called **AppXPackageFileList.xml**, which is used to configure which is the main executable to launch as part of the debug session.

Here is how it should look like in your scenario:

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

<Project ToolsVersion="14.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<PropertyGroup>

<MyProjectOutputPath>$(PackageLayout)</MyProjectOutputPath>

</PropertyGroup>

<ItemGroup>

<LayoutFile Include="$(MyProjectOutputPath)\win32\DesktopBridge.FlightTracker.exe">

<PackagePath>$(PackageLayout)\win32\DesktopBridge.FlightTracker.exe</PackagePath>

</LayoutFile>

</ItemGroup>

</Project>

Most of the XAML code is fixed, except for the **<LayoutFile>** entry: as you can see, it contains an attribute called **Include** and a node called **<PackagePath>** which points to the main executable of our desktop app inside the WinJS project.

The second step is to right click on this new project (**DesktopBridge.FlightTracker.Debug**) and configure the two fields called **PackageLayout** and **Startup Tile**. They are the ones that define the path to the folder that contains the app package, and which is the main process to start.

The path of the folder to specify in the **Package Layout** field is the following one:

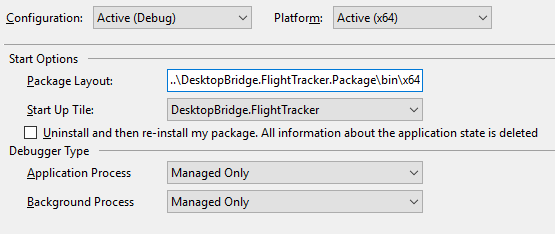
..\DesktopBridge.FlightTracker.Package\bin\x64\Debug\AppX

It’s like the **PackageFiles** folder that we obtained as output of the Desktop App Converter in the first exercise and that, thanks to the **makeappx** tool, can be turned into an AppX file. The visual wizard we have previously used to generate the AppX file, under the hood, leveraged in fact the same makeappx tool we have manually used in Exercise 1 on this folder.

**Note:** the AppX folder isn’t generated until you have deployed the app at least once. As such, before setting this path in the **Package Layout** field, make sure that you have deployed the WinJS project at least once.

As soon as you have set this path, the **Startup Tile** dropdown should automatically enable itself, allowing you to choose the only entry point that it’s specified in the manifest file, which is the **DesktopBridge.FlightTracker** app.

This is how the configuration of the Desktop Bridge Debugging Project should look like at the end of the process:



Now you’re ready to use the project for debugging purposes. To test it, just try to place a breakpoint in the source code of the original desktop project. For example, right click on the file called **Flight.cs** in the **DesktopBridge.FlightTracker** project and put a breakpoint in the first line of the **Form1\_Load()** event handler (which gets triggered when the main form of the app is displayed).

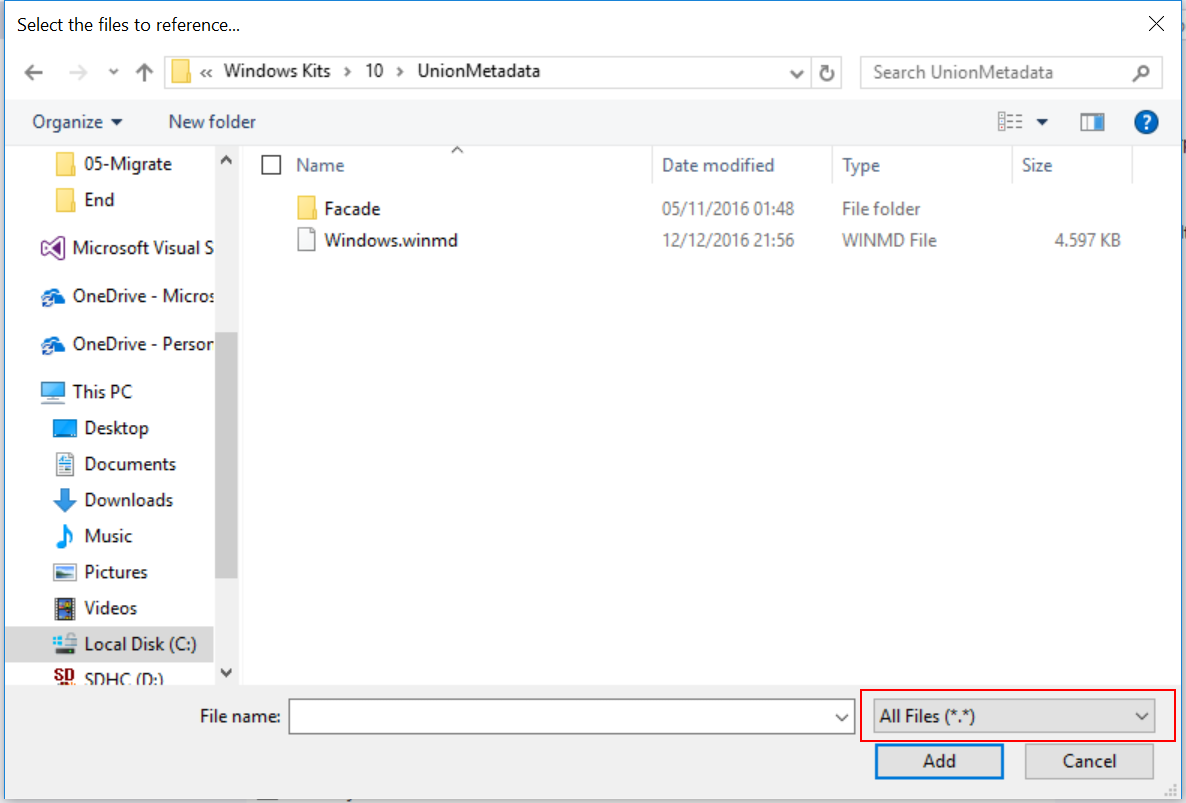
Now right click on the **DesktopBridge.FlightTracker.Debug** project, choose **Set as Startup project** and press F5: if you did everything correctly, you will notice that, despite the fact you are running the converted desktop version of the app, the breakpoint you have placed in the source code of the original desktop app will be triggered when the app starts, giving you a full debug experience (so you can add watches, perform step by step debugging, check the content of variables, see exception details, etc.).

We’re done: now we have a converted desktop app fully managed by Visual Studio 2017 and we are ready to start enhancing it.

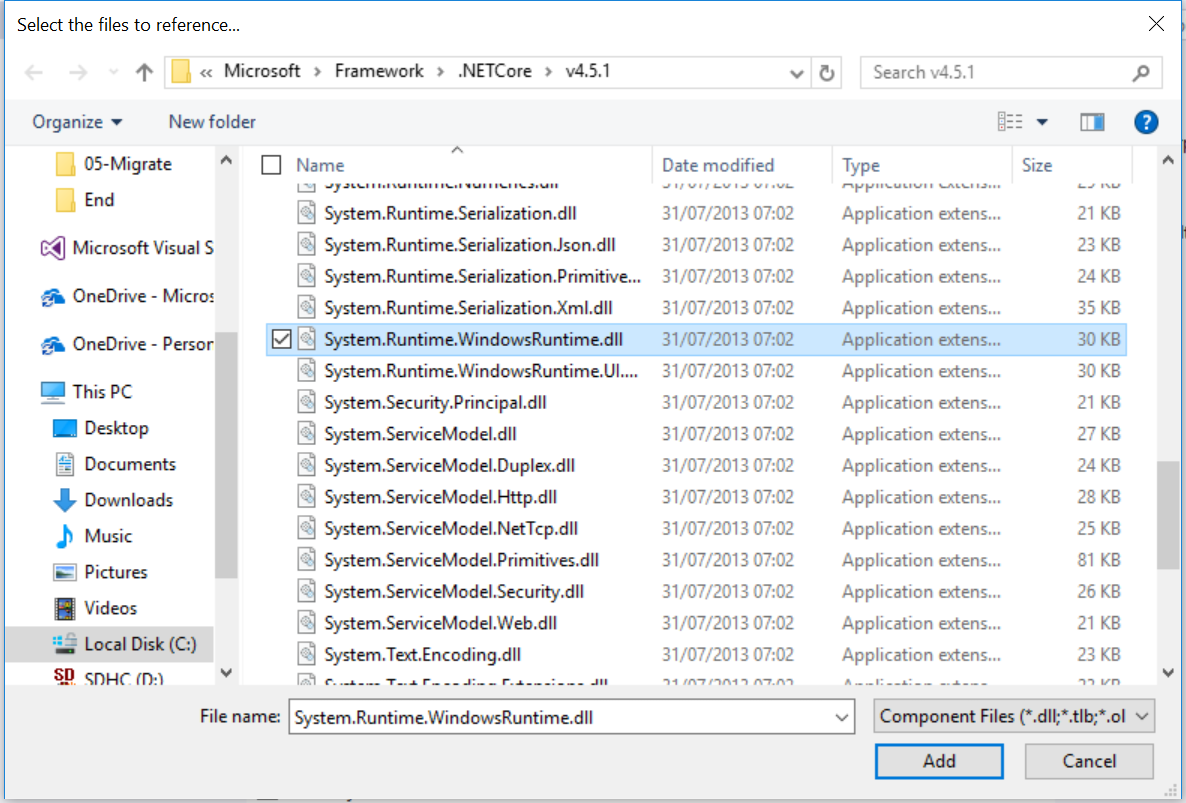
## Task 2 – Prepare your desktop to use the Universal Windows Platform

If the desktop application is based on the .NET Framework, like in our scenario, accessing to the Universal Windows Platform is very easy.

The first step is to add a reference to the Universal Windows Platform to the Windows Forms project: right click on the **DesktopBridge.FlightTracker** project, choose **Add -> Reference** and press the **Browse** button. You will have to look for a file called **Windows.md** placed in the path **C:\Program Files (x86)\Windows Kits\10\UnionMetadata.** Pay attention, in the dropdown menu near the **File name** field, to choose the option **All files (\*.\*).** The file we need to add, in fact, isn’t a standard DLL, but it’s a metadata file, which is a specific component of the Universal Windows Platform ecosystem and it’s identified by the **.winmd** extension. As such, it won’t be displayed by default until we change the option in the dropdown menu: after that, you’ll be able to see the **Windows.md** file. Choose it and press the **Add** button.



Even if, for the sake of this exercise, we don’t need it now, there’s another reference that will be useful for the future exercises. As you’ll probably know, one of the biggest features added in C# 5 is the support to the **async** and **await** keywords, which is a way to write asynchronous code in a simpler way. By default, they aren’t supported by a Windows Forms app: if you would try to use an asynchronous method, you would get a compilation error. This can become a serious limitation when you are in the Enhance phase, since all the APIs of the Universal Windows Platform that can take more than 50 ms to complete a task are asynchronous. To properly add support to the **async** and **await** keywords**,** you need to right click again on the **DesktopBridge.FlightTracker** project, choose **Add -> Reference** and, this time, look for a file called **System.Runtime.WindowsRuntime.dll** in the folder **C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5.1.**



## Task 2 – Display the notification

Now that we have added a reference to the Universal Windows Platform, we can start using some of the APIs that it offers. First, we need to add in the header of the **Flight.cs** class two namespaces, that contain the classes we’re going to use:

using Windows.Data.Xml.Dom;

using Windows.UI.Notifications;

Now we can add a method to display a toast notification on the screen:

private void ShowNotification()

{

string xml = $@"<toast>

<visual>

<binding template='ToastGeneric'>

<text>Flight Tracker</text>

<text>The boarding pass for flight {codeTextbox.Text} from {departureTextbox.Text} to {arrivalTextbox.Text} has been exported on your desktop</text>

</binding>

</visual>

</toast>";

XmlDocument doc = new XmlDocument();

doc.LoadXml(xml);

ToastNotification toast = new ToastNotification(doc);

ToastNotificationManager.CreateToastNotifier().Show(toast);

}

A toast notification is identified by a XML payload, which describes the content that will be displayed. Windows 10 has greatly expanded the opportunities offered by notifications, which can now include multiple lines of text, images, custom icons, etc. You can refer to the following documentation if you want to know more: <https://msdn.microsoft.com/en-us/windows/uwp/controls-and-patterns/tiles-and-notifications-adaptive-interactive-toasts>

The notification we are going to create is very simple and it’s made just by two lines of text:

1. The first one contains the name of the application, which is **Flight Tracker**.
2. The second one contains a message to alert the user that the boarding pass has been exported with success. To make the message more meaningful, we’re including some of the information about the flight that the user has added in the form, like the code, the departure and the arrival city.

A toast notification is represented by a class of the Universal Windows Platform called **ToastNotification**, which belongs to the **Windows.UI.Notifications** namespace we have previously added. As parameter, when you create a new instance, you need to pass a **XmlDocument** object with the XML that defines its content. Pay attention that there are multiple versions of the **XmlDocument** class: the one we need to use belongs to the Universal Windows Platform and, as such, it’s part of the **Windows.Data.Xml.Dom** namespace.

To finally display the notification, we create a notifier object (by calling the **CreateToastNotifier()** method of the **ToastNotificationManager** class) and we call the **Show()** method, passing as parameter the **ToastNotification** object we have just created.

That’s all: now we need to replace, in the Windows Forms app, the confirm message that was displayed inside the form with a call to the **ShowNotification()** method. As such, right click on the **Flight.cs** file of the **DesktopBridge.FlightTracker** project, choose **View code** and look for the event handler called **exportButton\_Click**. The last line of code will be the following one:

operationStatusLabel.Text = "Export completed";

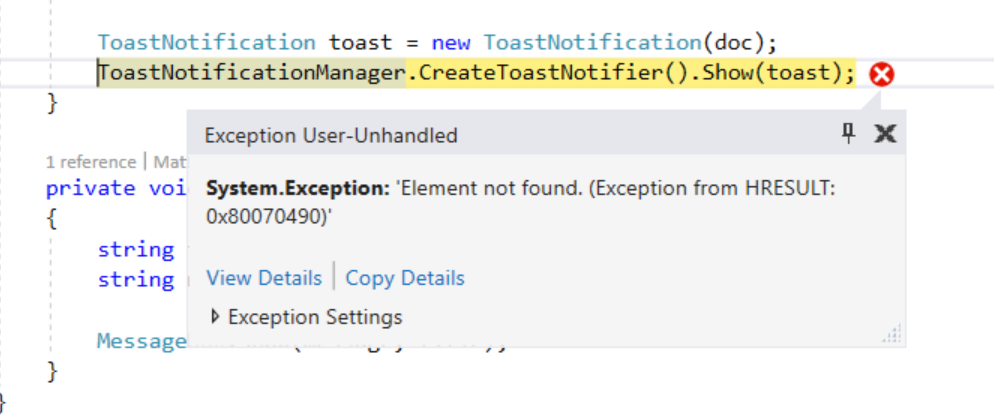
Simply remove this line and replace it with the following lines of code:

operationStatusLabel.Visible = false;

ShowNotification();

We’re hiding the label that, in the original version, was displaying the confirm message and we are invoking the **ShowNotification()** method as replacement.

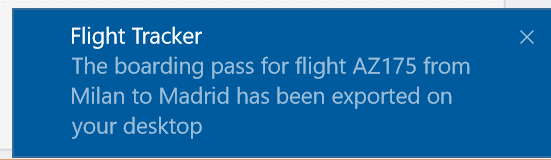
We can now test the work we’ve done. Let’s try to set, as startup project, the Windows Forms one: right click on the **DesktopBridge.FlightTracker** project, choose **Set as StartUp project** and then press F5 on your keyboard to launch the debug. Fill the form and press the Export button: a progress bar will be displayed to show that the operation is in progress and, after 5 seconds, you will get… an exception!



The reason is that the app is running as a native desktop app and not as a Universal Windows Package. This is an important lesson to learn: it isn’t enough for a desktop app to run on Windows 10 to be able to leverage the Universal Windows Platform. This opportunity is offered by the Desktop Bridge and, as such, it needs to run as an app package.

Since we have already a Desktop Bridge Debugging Project configured, it’s easy to repeat the test in the proper way: first, right click on the **DesktopBridge.FlightTracker** project and choose **Rebuild**, to make sure that all the latest changes we’ve made to the Windows Forms app are copied inside the **win32** folderof the WinJS project. Now, right click on the **DesktopBridge.FlightTracker.Package** project, choose **Deploy** and, in the end, right click on the **DesktopBridge.FlightTracker.Debug** project, choose **Set as StartUp project** and press F5 again: this time, the converted version of the app will be launched.

Consequently, when the boarding pass export procedure will be completed, you will properly see the toast notification appearing on the bottom right corner of the screen of your computer:



## Task 3 – Maintain retro compatibility

The code we have just introduced has a downside, as you may have noticed from the test we did at the end of Task 2 of this exercise: the code is specific of the Universal Windows Platform and, by adding it, we have made our application not compatible with previous versions of Windows or with a standard desktop installer of Flight Tracker, in addition to the one we will publish on the Store as converted.

One solution would be to create multiple branches of our project: one for the converted version and one for the traditional desktop of the app. However, maintaining multiple branches has always a cost for the developer. For example, if we decide to add a new feature to the Flight Tracker application which isn’t specific of the Universal Windows Platform (like adding the capability of searching for a flight) we need to find a way to maintain the two branches in sync.

A solution to avoid this problem and maintain a single code base is to leverage **runtime checks:** we will detect if the app is running as native or converted. In our case, the **ShowNotification()** method will be called only if the app is running as converted, otherwise it will be ignored.

To achieve this goal, we can leverage a method called **GetCurrentPackageFullName()**, which has been introduced in the native Windows API **kernel32.dll** starting from Windows 8, which was the first edition of Windows to support the Windows Runtime and the Windows Store apps (the predecessors of the Universal Windows Platform). This method can be used to retrieve the package’s identity of the running application: however, this is a concept that belongs only to the Universal Windows Platform world. A Win32 application doesn’t have a package identity. As such, we can leverage this API to understand if the app is running as converted or not and act accordingly in our application. However, there are two challenges to solve:

1. The API is available only in C++, as such it requires a bit of work to be used inside an application based on the .NET Framework like Flight Tracker.
2. The API has been introduced only starting from Windows 8: if you try to invoke it in a desktop application running on Windows 7, you will get an exception because it doesn’t exist.

To make the developer’s life easier, we’re going to use a library called **DesktopBridge.Helpers** which is available:

1. As an open source project, with instructions and samples, on GitHub: <https://github.com/qmatteoq/DesktopBridgeHelpers>
2. As a NuGet package, that can be easily installed on any .NET based application: <https://www.nuget.org/packages/DesktopBridge.Helpers/>

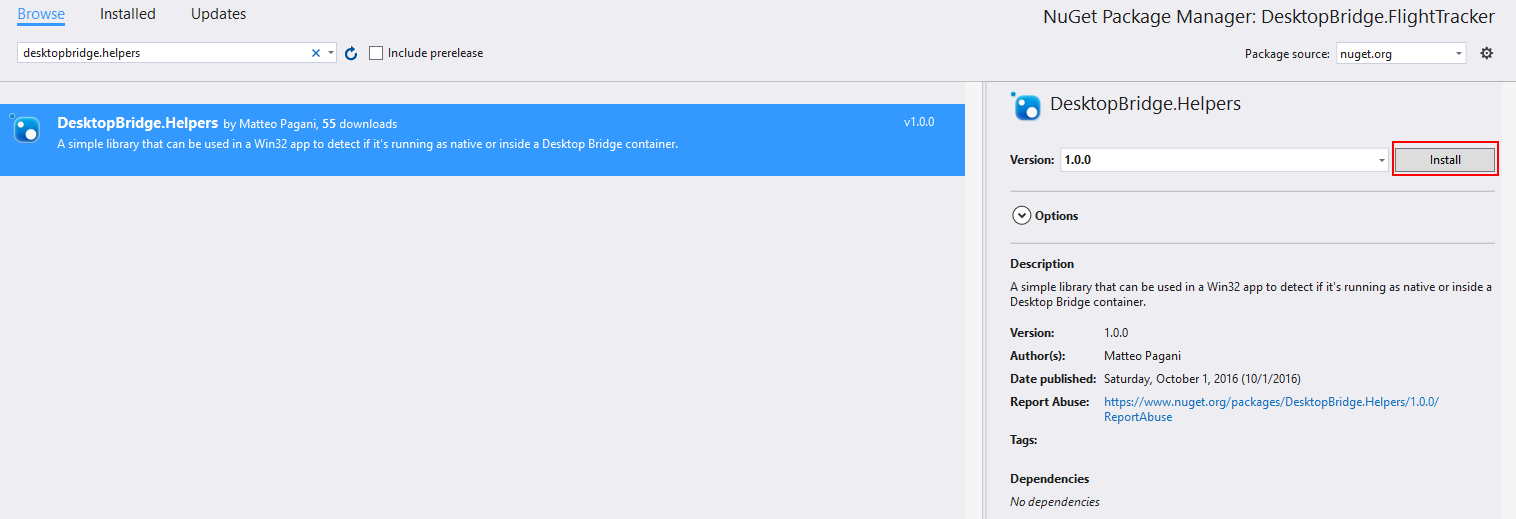
The library exposes a simple method, called **IsRunningAsUwp()**, that will return a Boolean value, based on the fact that the app is running as converted or not. Under the hood, the library takes care of:

1. Using the P/Invoke mechanism offered by the .NET Framework to invoke the native C++ API.
2. Return immediately **false** if the app is running on a Windows version that doesn’t support converted apps, like Windows 7.

We’re going to use it for two purposes in the desktop application:

1. To show the toast notification we have previously created in case the app is running as converted, instead of the displaying a message in the app’s window.
2. To hide the **Check for updates** button if the app is running as converted.

The first step is to add it to our project. The easiest way is using NuGet: right click on the **DesktopBridge.FlightTracker** project, choose **Manage NuGet packages** and, in the **Browse** section, look for a package called **DesktopBridge.Helpers**. Once you’ve found it, press the **Install** button on the right side of the window, as highlighted in the following image:



As first step, let’s use this library to determine if we can display the notification or not. Here is how the final **exportButton\_Click()** event handler looks like:

private async void exportButton\_Click(object sender, EventArgs e)

{

operationStatusLabel.Text = "Exporting file...";

progressBar.Visible = true;

// Simulate a high load process

await Task.Delay(5 \* 1000);

string userPath = Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);

string fileName = $"{userPath}\\BoardingPass.txt";

var builder = new StringBuilder();

builder.AppendLine("Boarding pass generated by FlightTracker");

builder.AppendLine("-----------------------------------------");

builder.AppendLine($"Flight code: {codeTextbox.Text}");

builder.AppendLine($"Flight date: {dateTextbox.Text}");

builder.AppendLine($"Departure city: {departureTextbox.Text}");

builder.AppendLine($"Arrival city: {arrivalTextbox.Text}");

builder.AppendLine("-----------------------------------------");

builder.AppendLine("Thank you for using FlightTracker");

File.WriteAllText(fileName, builder.ToString());

progressBar.Visible = false;

operationStatusLabel.Visible = false;

DesktopBridge.Helpers bridgeHelpers = new DesktopBridge.Helpers();

if (bridgeHelpers.IsRunningAsUwp())

{

ShowNotification();

}

else

{

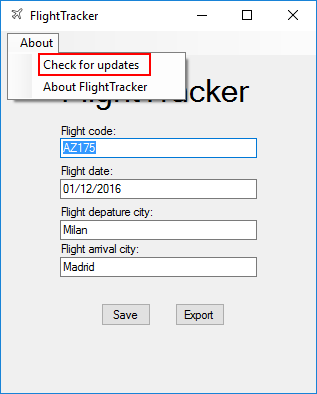
operationStatusLabel.Text = "Export completed";

}

}

Highlighted in yellow you can see the difference compared to the original code. We create a new instance of the **DesktopBridge.Helpers** class and we call the **IsRunningAsUwp()** method. If the returned value is **true**, the app is running as converted and we show the notification using the Universal Windows Platform APIs; otherwise, if the app is running as a regular desktop app, we continue to do what we were doing in the original version, which is displaying a message in the main window.

The second step is to use the same library to hide the **Check for updates** button in the **About** menu if the app is running as converted.



As we have already mentioned in the introduction of the lab, this option must be removed from the converted version of the app: a Universal Windows Platform app can’t update itself, but it needs to be updated using the Store or one of the many enterprise distribution systems that are available, like Intune, SSCM or Windows Store for Business.

To handle this scenario, we need to open the **Flight.cs** file in the **DesktopBridge.FlightTracker** project and, at the end of the **Form1\_Load()** event handler (right after the initialization of the flight data from the registry), add the following code:

DesktopBridge.Helpers bridgeHelpers = new DesktopBridge.Helpers();

if (bridgeHelpers.IsRunningAsUwp())

{

updateStripMenuItem.Visible = false;

}

This is how the final definition of the event handler should look like, with highlighted in yellow the code we’ve added:

private void Form1\_Load(object sender, EventArgs e)

{

\_regKey = Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Microsoft\DesktopBridgeWorkshop\Demo", true);

if (\_regKey == null)

{

\_regKey = Registry.CurrentUser.CreateSubKey(@"SOFTWARE\Microsoft\DesktopBridgeWorkshop\Demo", RegistryKeyPermissionCheck.ReadWriteSubTree);

}

codeTextbox.Text = (string)\_regKey.GetValue("Code");

dateTextbox.Text = (string)\_regKey.GetValue("Date");

departureTextbox.Text = (string)\_regKey.GetValue("Departure");

arrivalTextbox.Text = (string)\_regKey.GetValue("Arrival");

DesktopBridge.Helpers bridgeHelpers = new DesktopBridge.Helpers();

if (bridgeHelpers.IsRunningAsUwp())

{

updateStripMenuItem.Visible = false;

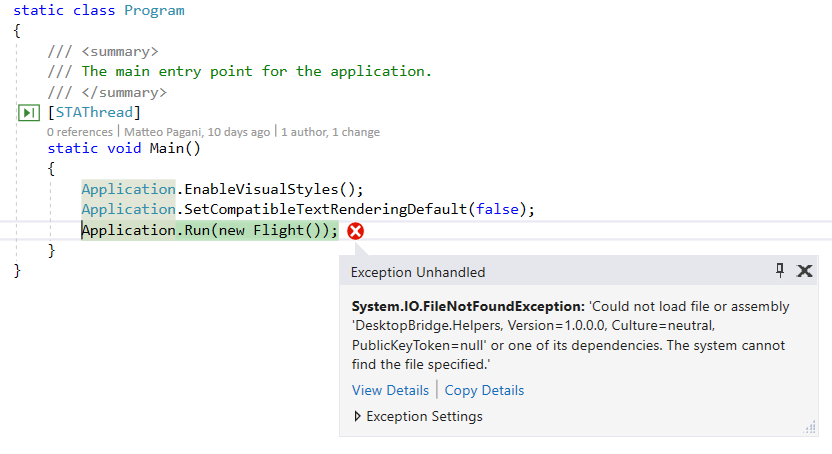
}

}

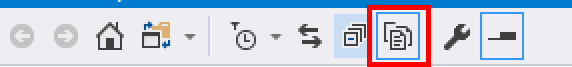
Also in this case we create a new instance of the **DesktopBridge.Helpers** object and we call the **IsRunningAsUwp()** method. If it returns **true**, it means that the application is running as converted, so we don’t need to display the **Check for updates** button: consequently, we set the **Visible** property of the **updateStripMenuItem** control to **false**.

Let’s test our code: make sure that the **DesktopBridge.FlightTracker.Package** project is built and deployed, then set the **DesktopBridge.FlightTracker.Debug** project as the startup one, do a **Clean** and **Rebuild** to make sure that the updated executable is properly synchronized and then press F5.

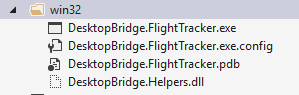
Well, the result isn’t exactly the one we were expecting:



By adding the **DesktopBridge.Helpers** NuGet package, we have added a new dependency in our application: the **DesktopBridge.FlightTracker.exe** executable can’t work anymore by itself, but it needs also to have access to the **DesktopBridge.Helpers.dll** file. However, if this new file has been automatically copied in the **win32** folder of the WinJS project thanks to the post build task we have added when we have configured the project to use Visual Studio 2017, it isn’t part of the app package yet: if you expand the **win32** folder in the **DesktopBridge.FlightTracker.Package** project, you’ll notice that you won’t see any file called **DesktopBridge.Helpers.dll.** As such, we need to repeat the procedure we did in the beginning, when we included the **win32** folder inside the project. Expand the **win32** folder and press again the button to show all the files that are stored inside the folder:



You will see that the **DesktopBridge.Helpers.dll** file will be there, but with a transparent icon, which means that isn’t part of the project:



Right click on the **DesktopBridge.Helpers.dll** file and choose **Include in project**. Then, by keeping it selected, open the **Properties** panel in Visual Studio and make sure that, also for this file, the **Copy to output directory** property is set to **Copy if newer**.

Now try again to compile and deploy the **DesktopBridge.FlightTracker.Package** project and to launch for debugging the **DesktopBridge.FlightTracker.Debug** one. This time, you shouldn’t encounter any exception, and everything should work as expected.

Testing if the work we have done is correct is easy. First, try to set up, as startup project, the Windows Forms application (so the **DesktopBridge.FlightTracker** one) and press F5 to start the debugging. In this case, since the **IsRunningAsUwp()** method will return **false**, you should notice that:

1. In the **About** menu, the **Check for updates option** is visible.
2. If you try to export the boarding pass, you will see a message at the end of the operation directly in the main window.

Now, instead, set as startup project the **DesktopBridge.FlightTracker.Debug** project and press F5 to start the debugging of the converted version. This time, since the **IsRunningAsUwp()** method will return **true**, you should notice that:

1. In the **About** menu, the **Check for updates** option will be hidden.
2. If you try to export the boarding pass, at the end of the operation you will see a toast notification appearing on the bottom right corner of the screen.

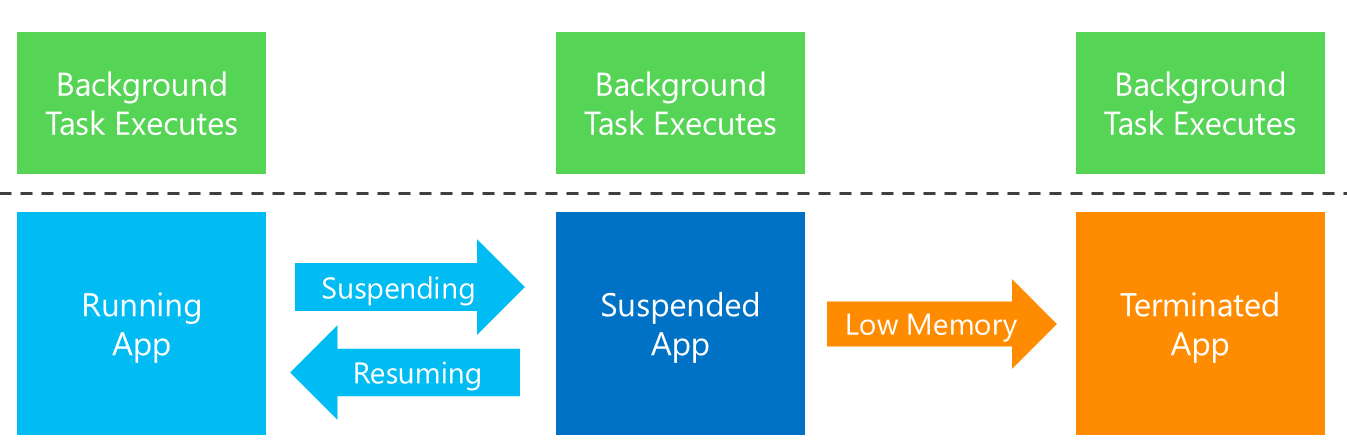
# Exercise 3 – Extend

In the Extend phase we do another step forward in the integration between the Win32 world and the Universal Windows Platform ecosystem. In Exercise 2, we have seen how to leverage some APIs. In this exercise, instead, we’re going to see how to use real components that belong just to the Universal Windows Platform, like background tasks, app services, etc.

In this exercise, we’re going to focus on adding a background task as a component. Background tasks are connected to the application’s lifecycle of a Universal Windows Platform app, which is quite different from the one of a standard desktop application.

In a Win32 application, in fact, the user is the only actor in control of the application: he can open and close it but, even if he minimizes it, the app continues to run in background. A Universal Windows Platform app, instead, adopts a more conservative approach: after a while that the app isn’t used (because it’s minimized on the taskbar or because, on another device like a phone or the Xbox One, the user has switched to another app), it’s suspended, which means that it’s kept in memory, but every running process is terminated. This approach helps to save battery and resources, which can be a critical factor on some devices like tablets or phones.

However, not allowing developers to perform any background operation at all would be very limiting, so the Universal Windows Platform has introduced the concept of background tasks: separate components from the main app, which contain pieces of code that can be executed no matter which is the status of the app (running, not running, suspended, etc.).



However, a converted app, even if it runs inside a UWP container, still runs as a regular Win32 process: as such, the lifecycle I’ve previously described doesn’t apply to applications converted using the Desktop Bridge. Consequently, why should a desktop developer be interested anyway in integrating a background task in his Win32 application?

Background tasks are connected to the concept of **trigger**, which is the event that invokes the execution of the task. Consequently, they have some advantages:

1. They are entirely handled by Windows, so once a background task is registered it will always be executed, no matter if the trigger occurs even when the converted app is completely closed. In the standard Win32 world, we would need to rely on more complex approaches to achieve the same goal (like creating a Windows Service or having a process always in background).
2. Some triggers (like **TimeTrigger**, which can be used to periodically repeat an operation in background) can be easily implemented also in a standard desktop application (for example, using a timer). The same can’t be said for other kind of triggers (like the ones connected to the communication with Bluetooth devices or to other system events).
3. Triggers are respectful of the resources and of the battery of the device: Windows will always make sure that the performance impact of a background task doesn’t degrade the user experience of the device where the application is running.

In this exercise, we’re going to create a background task connected to a trigger called **TimeZoneChange**: every time the time zone of the device where the app is running will change, we will display a notification to the user, to alert him that he has moved to a different time zone so he needs to double check that he’s still in time to take his flight.

## Task 1: Create the background task

A background task is a Windows Runtime Component, which is a separate project from the main application, and contains the code that will be executed in background even when the app is not running.

Let’s start by opening the solution that you can find in the folder **\03-Extend\01-Start** folder. If you’re doing the exercises one after the other, there’s no need: the starting point of Exercise 3 will be the application we have built at the end of Exercise 2.

The first step is to create the task: right click on the solution in Visual Studio, choose **Add** **-> New project** and, in the **Visual C# -> Windows Universal** category, choose **Windows Runtime Component (Universal Windows)** and create a new project, by calling it **DesktopBridge.FlightTracker.Notification**.

It will contain a default file, called **Class1.cs**: rename it with a more meaningful name (like **ToastTask.cs**). Remember also to rename the name of the class from **Class1** to **ToastTask** (Visual Studio should automatically ask you if you want to do that when you rename the file, in this case simply choose **Yes**).

To act as a background task, the class needs to implement the **IBackgroundTask** interface, which is part of the **Windows.ApplicationModel.Background** namespace. Additionally, since we’re going to display also a notification, we will need to add a reference also to the **Windows.Data.Xml.Dom** and **Windows.UI.Notifications** namespaces. Consequently, make sure to add the proper definitions to the header of the class:

using Windows.ApplicationModel.Background;

using Windows.Data.Xml.Dom;

using Windows.Storage;

using Windows.UI.Notifications;

When you implement the **IBackgroundTask** interface, you are required to define the **Run()** method, like in the following sample:

public sealed class ToastTask: IBackgroundTask

{

public void Run(IBackgroundTaskInstance taskInstance)

{

}

}

Inside the **Run()** method, we’re going to add the code to show the toast notification, which will be like to the one we have added in Exercise 2 when we have added the **ShowNotification()** method in the Windows Forms app.

public void Run(IBackgroundTaskInstance taskInstance)

{

string xml = $@"<toast>

<visual>

<binding template='ToastGeneric'>

<text>Flight Tracker</text>

<text>Pay attention, your time zone has changed! Make sure your flight is still in time!</text>

</binding>

</visual>

</toast>";

XmlDocument doc = new XmlDocument();

doc.LoadXml(xml);

ToastNotification toast = new ToastNotification(doc);

ToastNotificationManager.CreateToastNotifier().Show(toast);

}

The code simply takes care of defining a string with the XML payload of the notification (which contains a warning message for the user), embed it into a **XmlDocument** and use it to create a new **ToastNotification** object. In the end, thanks to the **ToastNotificationManager** and the notifier object, we display the notification by passing it to the **Show()** method.

## Task 2: Register the background task in the application

As you have seen from the previous code snippet, a background task is independent from the trigger that will handle it: it doesn’t contain any reference about how it will be used and under which conditions. It’s up to the main application to define this information.

Open the code of the **Flight.cs** file of the **DesktopBridge.FlightTracker** project and, in the event handler that is triggered when the form is loaded (called **Form1\_Load()**), add the following code at the end:

string triggerName = "FlightTimeZoneTrigger";

// Check if the task is already registered

foreach (var cur in BackgroundTaskRegistration.AllTasks)

{

if (cur.Value.Name == triggerName)

{

// The task is already registered.

return;

}

}

BackgroundTaskBuilder builder = new BackgroundTaskBuilder();

builder.Name = triggerName;

builder.SetTrigger(new SystemTrigger(SystemTriggerType.TimeZoneChange, false));

builder.TaskEntryPoint ="DesktopBridge.FlightTracker.Notification.ToastTask";

builder.Register();

We need to include this code in the condition that we’re testing thanks to the **DesktopBridgeHelpers** library, to determine if the app is running as converted of not. In fact, we don’t have to register the background task if the app is running as a regular desktop app, otherwise we would get multiple errors.

First, we give a name to the trigger, which needs to be unique: this way, we can avoid registering the same task multiple times. This is the purpose of the first part of the code: thanks to the **BackgroundTaskRegistration.AllTasks** collection we have access to all the tasks registered by the current app: if the task already exists, we can simply return and avoid continuing the registration.

Otherwise, we move on and we register it using the **BackgroundTaskBuilder** class, which belongs to the **Windows.ApplicationModel.Background** namespace. As such, makes sure to add the following namespace in the header of the **Flight.cs** class, otherwise you will get some compilation errors:

using Windows.ApplicationModel.Background;

A task is defined by:

1. A name, which is assigned to the **Name** property.
2. A trigger, which is the even that is going to invoke the task. We can define it by calling the **SetTrigger()** method. In this case, we use a **SystemTrigger** (which means that it’s a trigger connected to an event handled directly by the operating system) and, specifically, the **TimeZoneChange** one: it means that the background task will be invoked every time the time zone of the devices changes. By setting the second parameter as **false**, we’re saying that it isn’t a one-shot background task: we want it to be invoked every time the time zone changes, not just the first time.
3. A **TaskEntryPoint**, which is the full signature of the class we have created in the Windows Runtime Component and that implements the **Run()** method. In our scenario, it’s **DesktopBridge.FlightTracker.Notification.ToastTask.**

After we have defined the task, we can register it by calling the **Register()** method.

## Task 3: Configure the manifest

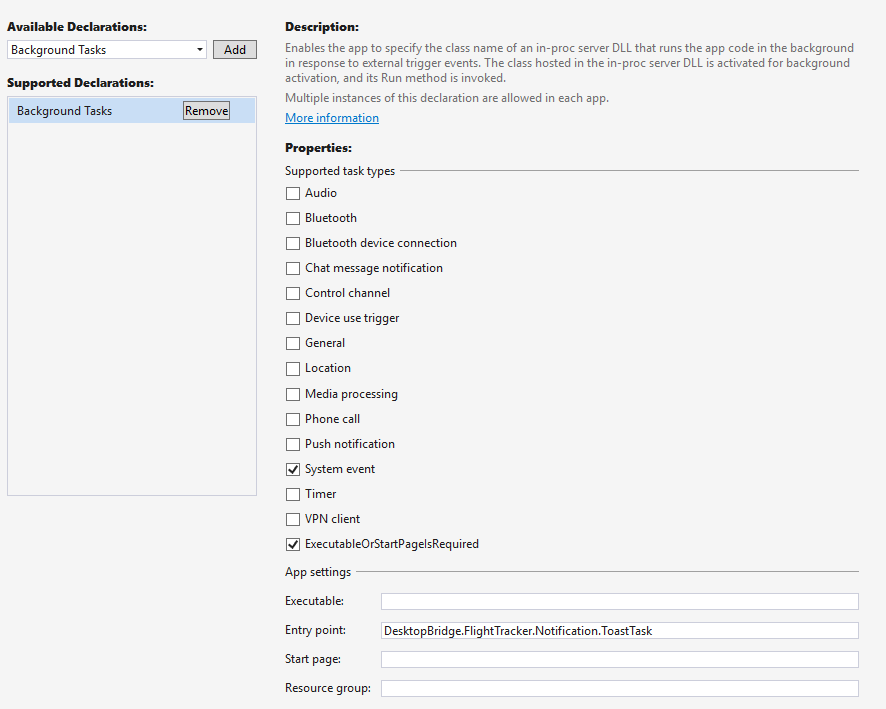
Background tasks, other than in the application, needs to be registered also in the manifest file. This is another one of the benefits of using Visual Studio 2017 and the WinJS project to handle our converted desktop app. We can leverage, in fact, the visual manifest editor integrated in Visual Studio to perform this operation, instead of having to manually add the corresponding XML entries.

As such, double click on the **package.appxmanifest** in the **DesktopBridge.FlightTracker.Package** project and move to the **Declarations** section. In the **Available declarations** dropdown choose the option **Background Tasks** and press **Add**.

In the section that will appear in the middle of the editor, make sure to:

1. Check the **System event** value in the **Supported task types** section.
2. Specify as **EntryPoint** in the **App settings** section the full signature of the class that implements the task. It’s the same signature we have set in the **TaskEntryPoint** property when we have registered the task in the code.

This is how the editor should like:



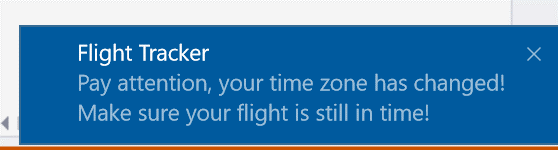
## Task 4: Deploying the background task in the converted app

We can consider the background task like a dependency, as we’ve seen in the Exercise 2 when we have added the **DesktopBridge.Helpers** library. When we build the **DesktopBridge.FlightTracker.Notification** project, we get an output inside the **bin/x86/Debug** folder. The only difference is that, since it’s a Windows Runtime Component and not a Class Library, we won’t get a DLL as outcome, but two files: **DesktopBridge.FlightTracker.Notification.pri** and **DesktopBridge.FlightTracker.Notification.winmd**.

Consequently, if we would try to test the application at this point of the exercise, the background task registration would simply fail, since the **BackgroundTaskBuilder** class won’t find the entry point **DesktopBridge.FlightTracker.Notification.ToastTask**. We need, also in this case, to make sure that the output of the Visual Studio build is copied inside the app package. Again, also this scenario is simplified by the WinJS project: it’s enough to right click on the **DesktopBridge.FlightTracker.Package** project, choose **Add -> Reference** and, in the **Project -> Solution** section, check the **DesktopBridge.FlightTracker.Notification** project.

Now we can test our work. Make sure to build all the projects that are part of the solution and to deploy the **DesktopBridge.FlightTracker.Package** one**,** then to select, as startup project, the **DesktopBridge.FlightTracker.Debug** one, perform the usual **Clean** and **Rebuild** operation to make sure that all the changes we have applied to the Windows Forms app are included in the app package and then press F5.

Now open the Start menu of your PC, choose **Settings -> Time & Language -> Date & Time**. In case it’s enabled, disable the **Set time zone automatically** option and, from the **Time zone** dropdown, choose a different time zone than the current one. If you did everything correctly, you will see a toast notification appearing on the bottom right corner of your screen.

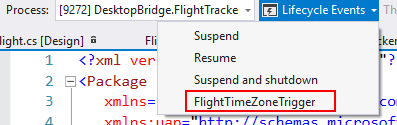


If you stop the debugging so that the application is closed and you repeat the test, you will see again the notification appearing. The reason is that the background task is now registered into the system and, consequently, Windows will take care of handling it, no matter if the app that registered it is running or not.

*Is there a better way to test the background task?*

Some background task can be complicated to test. In our scenario, the **TimeZoneChange** one wasn’t so hard to trigger: it has been enough to change the current time zone of our computer. However, there are some other triggers (like the ones connected to communication with Bluetooth devices or geolocation services) that can be harder to test. For this reason, the Visual Studio toolbar that is displayed at the top when a Universal Windows Platform application is in debugging mode, offers a dropdown menu called **Lifecycle events** to simulate the different stages of the lifecycle, which includes any background task that has been registered.

In case of the Flight Tracker app, this is the experience you should see in your Visual Studio window while you’re debugging the converted version of your app:



By clicking on the name of the task (in this case, **FlightTimeZoneTrigger**) the background task will be immediately executed, without requiring you to manually change the time zone of your computer.

## Task 5: Display the info about the flight in the notification

The notification we have generated in the previous step is useful, but there’s space for improvement: we’re generically warning the user to check his flight, but we are not specifying its details. It would be great if, other than the previous message, we could add also the precise information about the flight that the user is tracking. However, we hit a blocker if we want to implement this feature: the Windows Forms app stores its data in the registry while a background task, since belongs to the Universal Windows Platform ecosystem, doesn’t have access to it. Consequently, we need to find a different way to store the information about the flight, so that they can be read both by the Windows Forms app and by the background task.

The best approach to leverage for our scenario is the local storage, which offers a specific area where to easily store settings. Consequently, instead of storing the info about the flights in the registry, we’re going to use the **ApplicationData.Current.LocalSettings** class, which offers a dictionary where we can store key / value pairs. Each info about the flight will be stored into the **Values** collection, with a unique key.

The first step is to add, in the header of the **Flight.cs** file of the **DesktopBridge.FlightTracker** project, the namespace **Windows.Storage**, which give you access to the storage APIs of the Universal Windows Platform:

using Windows.Storage;

Here is, instead, the new code that is going to replace the old definition of the **saveButton\_Click** event handler, which is invoked when the user presses the Save button:

private void saveButton\_Click(object sender, EventArgs e)

{

ApplicationData.Current.LocalSettings.Values["Code"] = codeTextbox.Text;

ApplicationData.Current.LocalSettings.Values["Date"] = dateTextbox.Text;

ApplicationData.Current.LocalSettings.Values["Departure"] = departureTextbox.Text;

ApplicationData.Current.LocalSettings.Values["Arrival"] = arrivalTextbox.Text;

operationStatusLabel.Text = "The flight has been saved";

}

Due to this change, we need to modify also the code in the **Load** event of the form: we won’t need to read any more the stored data from the registry, but from the local settings. To achieve this goal, we’re going to do the opposite operation compared to the save one: we’re going to retrieve each value and display it in the proper **TextBox** control. The difference is that, this time, we will have to call the **ToString()** method on each retrieved value, since the **Values** collection stores generic objects. Additionally, before retrieving the value, we use the **ContainsKey()** method to check if it exists: the application, in fact, can be in a state (like during the first run) in which no data has been saved yet.

As such, here is the new definition of the **Form1\_Load()** event handler of the **Flight** class:

private void Form1\_Load(object sender, EventArgs e)

{

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Code"))

{

codeTextbox.Text = ApplicationData.Current.LocalSettings.Values["Code"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Date"))

{

dateTextbox.Text = ApplicationData.Current.LocalSettings.Values["Date"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Departure"))

{

departureTextbox.Text = ApplicationData.Current.LocalSettings.Values["Departure"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Arrival"))

{

arrivalTextbox.Text = ApplicationData.Current.LocalSettings.Values["Arrival"].ToString();

}

DesktopBridge.Helpers bridgeHelpers = new DesktopBridge.Helpers();

if (bridgeHelpers.IsRunningAsUwp())

{

updateStripMenuItem.Visible = false;

string triggerName = "FlightTimeZoneTrigger";

// Check if the task is already registered

foreach (var cur in BackgroundTaskRegistration.AllTasks)

{

if (cur.Value.Name == triggerName)

{

// The task is already registered.

return;

}

}

BackgroundTaskBuilder builder = new BackgroundTaskBuilder();

builder.Name = triggerName;

builder.SetTrigger(new SystemTrigger(SystemTriggerType.TimeZoneChange, false));

builder.TaskEntryPoint = "DesktopBridge.FlightTracker.Notification.ToastTask";

builder.Register();

}

}

Highlighted in yellow you can see the new lines of code that we have added, in replacement of the old ones that were loading the data from the system registry.

Now that the info about the flight are stored in the local settings, we can retrieve them also in the background task. As such, in the **ToastTask** class of the **DesktopBridge.FlightTracker.Notification** project we’re going to add a new method that, by leveraging the **ApplicationData.Current.LocalSettings** API, will retrieve these info and will prepare a message to add to the notification.

private string GenerateFlightInfo()

{

string code = string.Empty;

string date = string.Empty;

string departure = string.Empty;

string arrival = string.Empty;

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Code"))

{

code = ApplicationData.Current.LocalSettings.Values["Code"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Date"))

{

date = ApplicationData.Current.LocalSettings.Values["Date"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Departure"))

{

departure = ApplicationData.Current.LocalSettings.Values["Departure"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Arrival"))

{

arrival = ApplicationData.Current.LocalSettings.Values["Arrival"].ToString();

}

string message = $"Flight code: {code} - Date: {date} - From: {departure} - To: {arrival}";

return message;

}

Now that we have a way to generate the message, we can call it in the **Run()** method of the background task and add it as a new **<text>** node inside the toast notification definition. Here is the final version of the **Run()** method, with highlighted in yellow the new lines we have added compared to the previous definition:

*Snippet 3.5.5*

public void Run(IBackgroundTaskInstance taskInstance)

{

string flightInfo = GenerateFlightInfo();

string xml = $@"<toast>

<visual>

<binding template='ToastGeneric'>

<text>Flight Tracker</text>

<text>Pay attention, your time zone has changed! Make sure your flight is still in time!</text>

<text>{flightInfo}</text>

</binding>

</visual>

</toast>";

XmlDocument doc = new XmlDocument();

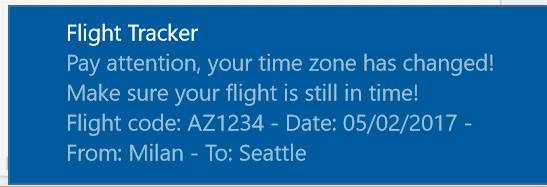
doc.LoadXml(xml);

ToastNotification toast = new ToastNotification(doc);

ToastNotificationManager.CreateToastNotifier().Show(toast);

}

If you want to test the changes we’ve made, just make sure to rebuild all the projects and deploy the **DesktopBridge.FlightTracker.Package** one, that set the **DesktopBridge.FlightTracker.Debug** as startup project and press F5: at first, all the fields will be empty, because we are not reading anymore the flight info from the registry but from the local settings, which don’t contain any value. As such, before performing our test, add again the info about the flight and press the **Save** button. Now invoke the background task (either by changing the current time zone in the Settings or by leveraging the **Lifecycle Events** dropdown): this time, the notification should contain, other than the warning message, also the info about the flight that you have previously saved in the Windows Forms app.



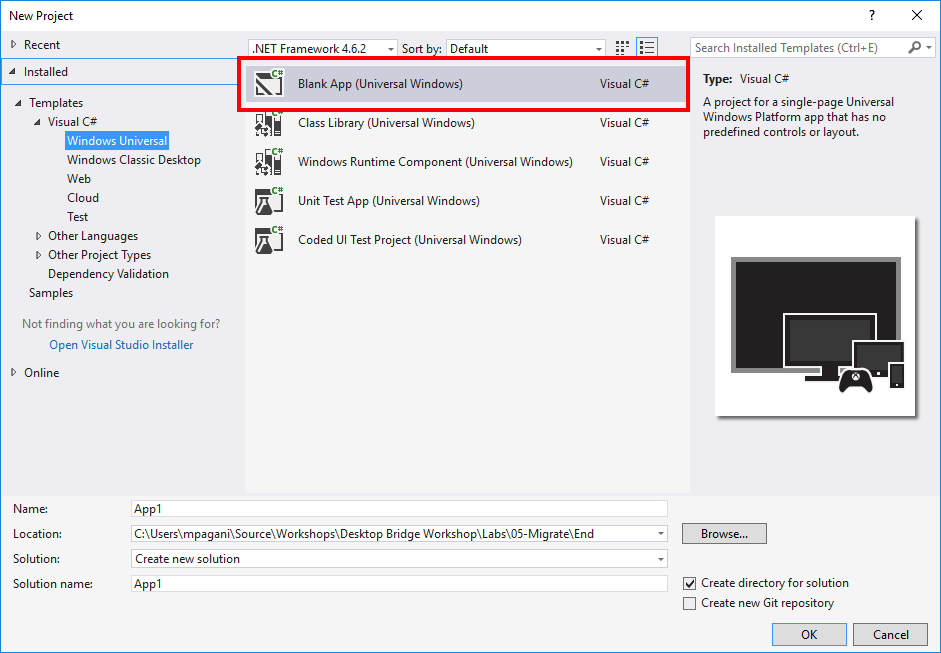
# Exercise 4 – Migrate

In this exercise, we’ll learn a new step of the Desktop Bridge journey, called Migrate. The main difference with the previous ones is that, in this scenario, the Win32 process isn’t anymore the protagonist, but the sidekick. The main actor will be a real Universal Windows Platform app, capable of running on any Windows 10 device. However, only when it’s running on the desktop, it will be able to invoke a Win32 process to perform some operations that the Universal Windows Platform doesn’t allow yet or to make easier to perform a gradual migration of the various components of our application.

In our scenario, we’re going to convert Flight Tracker from a Windows Forms application to a Universal Windows Platform one. However, for the moment we want to leave the boarding pass export feature unchanged: when the user presses the Export button, the boarding pass will be automatically created on the desktop’s user. As we have highlighted at the beginning of the lab, this scenario can’t be implemented using UWP APIs, since the desktop isn’t one of the folders where a Universal Windows Platform app can directly write content without user interaction.

## Task 1: Moving the main app from Windows Forms to UWP

Since now we’re moving to a new model, we need to create a new app based on the Universal Windows Platform. As such, the starting point of the exercise is a solution with a blank Universal Windows Platform application, which you will already find in the **\04-Migrate\Start** folder. This kind of project can be found in the **Visual C# -> Windows Universal** category: it’s the **Blank App (Universal Windows)** template.



A Universal Windows Platform project is made by many files. By default, like a Windows Forms project contains a main form, a UWP app contains a default page called **MainPage.xaml** with a connected filenamed **MainPage.xaml.cs**. The XAML file is the one that contains the user interface, created using the XAML markup language, while the .cs file is called code-behind and it contains all the code required to interact with the user interface.

The first step is to move the user interface of the old desktop apps from Windows Forms to XAML. There isn’t a way to automatically convert it, so we’ll have to create our user interface from scratch. The **MainPage.xaml** page contains, by default, the definition of an empty page with just a **Grid**, which acts a container for all the content. Here is the code we’re going to add inside the **Grid**:

*Snippet 4.1.1*

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"

HorizontalAlignment="Center" VerticalAlignment="Top">

<StackPanel>

<TextBlock Text="FlightTracker" Style="{StaticResource HeaderTextBlockStyle}" Margin="0, 0, 0, 30" />

<TextBox x:Name="codeTextbox" Header="Flight code" />

<TextBox x:Name="dateTextbox" Header="Flight date" />

<TextBox x:Name="departureTextbox" Header="Flight departure city" />

<TextBox x:Name="arrivalTextbox" Header="Flight arrival city" />

<StackPanel Orientation="Horizontal" HorizontalAlignment="Center" Margin="0, 20, 0, 20">

<Button Content="Save" Click="OnSaveFlight" />

<Button Content="Export" Click="OnExportBoardingPass" Margin="5, 0, 0, 0"

Visibility="Collapsed" x:Name="exportButton" />

</StackPanel>

<TextBlock x:Name="operationStatusLabel" Style="{StaticResource SubtitleTextBlockStyle}" />

<ProgressRing Background="Transparent" HorizontalAlignment="Center" VerticalAlignment="Center"

Width="50" Height="50" x:Name="progressBar" />

</StackPanel>

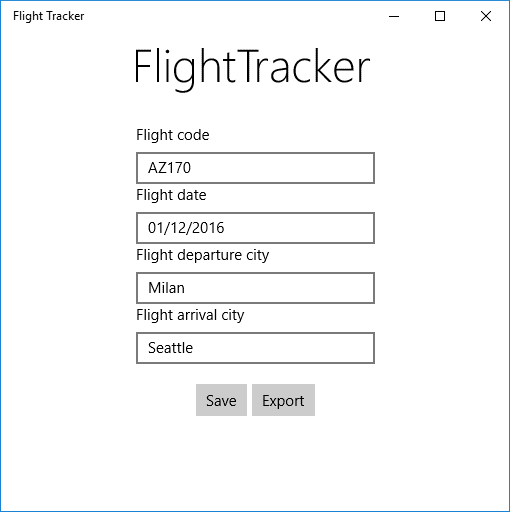
</Grid>

Inside the **Grid** we place a **StackPanel** control, which allows to stack all the children controls one below the other. Inside this container, we leverage the **TextBox** control, which has the same purpose of the Windows Forms one: asking for an input to the user. Additionally, the control offers a property called **Header**, where we can specify the purpose of the field. This text will be automatically displayed above the **TextBox**.

In the bottom part of the page we have added a **TextBlock** control, which will display a message with the output of the operation, and a **ProgressRing** control, which acts as a loading indicator: it will replace the progress bar that, in the Windows Forms app, was displayed during the boarding pass export procedure.

If you pay attention at the code, you may notice a peculiar thing: the **Visibility** property (which determines if a control is visible or not) of the button to export the boarding pass is set to **Collapsed**, which means that, by default, it’s hidden. The reason is that, in this phase, we’re going to integrate the UWP app with a Win32 process that will take care of generating the boarding pass. Since this feature is supported only by the desktop version of Windows 10 (the Windows APIs aren’t available on a phone or an Xbox One), the Export button will be hidden by default. Later we’re going to see that, in code, we’re going to make it visible only if we detect that the app is running on a device with the full version of Windows (like a desktop, a notebook or a tablet).

Here is the final look of our form:



## Task 2: The About page

In the Windows Forms app, we had also a menu called **About**, with some options to know more about the application and to check for updates. In the Universal Windows Platform ecosystem, menus can be replaced with an application bar, which is a bar that can contain one or more buttons and that can be displayed at the top or at the bottom of the page, thanks to a control called **CommandBar**. In this case, we’re placing it at the bottom and we’re adding an **AppBarButton** control, which is a special button that will display the About page with a proper icon. We won’t need the **Check for updates** button anymore, since UWP apps are updated automatically through the Store. The button is added as a primary command: it means that it will always be visible. Here is the code you need to past outside the main **Grid** but before the end of the **Page** node.

<Page.BottomAppBar>

<CommandBar>

<CommandBar.PrimaryCommands>

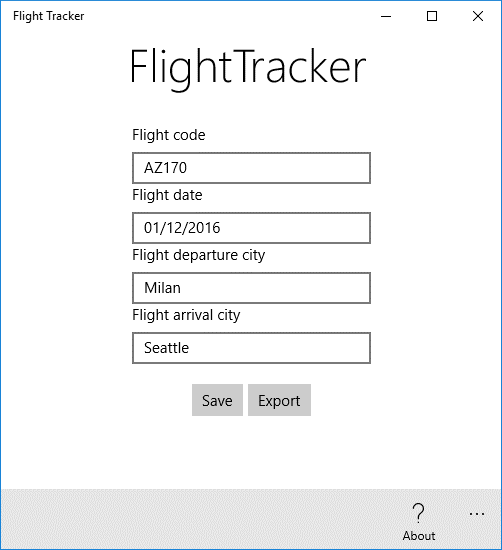
<AppBarButton Label="About" Icon="Help" Click="OnShowAbout" />

</CommandBar.PrimaryCommands>

</CommandBar>

</Page.BottomAppBar>

The previous code will update the user interface of the form like in the following image:



In the Windows Forms version of the application, the About option opened a new form with some info about the application. In the Universal Windows Platform, instead of creating a new page, which would force us to perform a navigation that would lead the user to lose the focus on the most important part of the app (the form to add the flight), we leverage a control called **ContentDialog**, which can be defined using XAML (like a normal page), but it can be displayed in overlay as a dialog. The dialog will contain the same information we were displaying in the forms app. To create it, right click on the UWP project, choose **Add -> New item** -> **Content Dialog** and call it **AboutPage.xaml**. Replace the content of the dialog with the following XAML code:

<ContentDialog

x:Class="DesktopBridge.FlightTracker.AboutPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="using:DesktopBridge.FlightTracker"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

mc:Ignorable="d"

Title="About FlightTracker"

PrimaryButtonText="Ok">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<TextBlock Text="Developers" Style="{StaticResource TitleTextBlockStyle}"

Grid.Row="0" Grid.ColumnSpan="2" Margin="0, 0, 0, 15" />

<StackPanel Grid.Row="1" Grid.Column="0" Margin="0, 0, 40, 0">

<TextBlock Text="Matteo Pagani" Style="{StaticResource SubtitleTextBlockStyle}" />

<TextBlock Text="Windows AppConsult Engineer" />

<TextBlock Text="matteo.pagani@microsoft.com" />

<TextBlock Text="@qmatteoq" />

</StackPanel>

<StackPanel Grid.Row="1" Grid.Column="1" >

<TextBlock Text="Adrian Fernandez" Style="{StaticResource SubtitleTextBlockStyle}" />

<TextBlock Text="Windows AppConsult Engineer" />

<TextBlock Text="adrifer@microsoft.com" />

<TextBlock Text="@tracker086" />

</StackPanel>

<TextBlock Grid.Row="2" Grid.ColumnSpan="2" Text="Plane by Icons fest from the Noun Project"

Margin="0, 30, 0, 0" />

</Grid>

</ContentDialog>

The dialog is made by a **Grid** control with two columns and three rows:

* The first row, which is spanned across the two columns, contains the header.
* The second row contains, split in the two columns, the information about the developers
* The third row, which again is spanned across the two columns, contains some additional copyright info

To display the **ContentDialog** we have just created we need to handle the **Click** event of the **AppBarButton** we have added before to the **CommandBar** control. Here is the definition of the event handler that we need to add in the **MainPage.xaml.cs** file:

private async void OnShowAbout(object sender, RoutedEventArgs e)

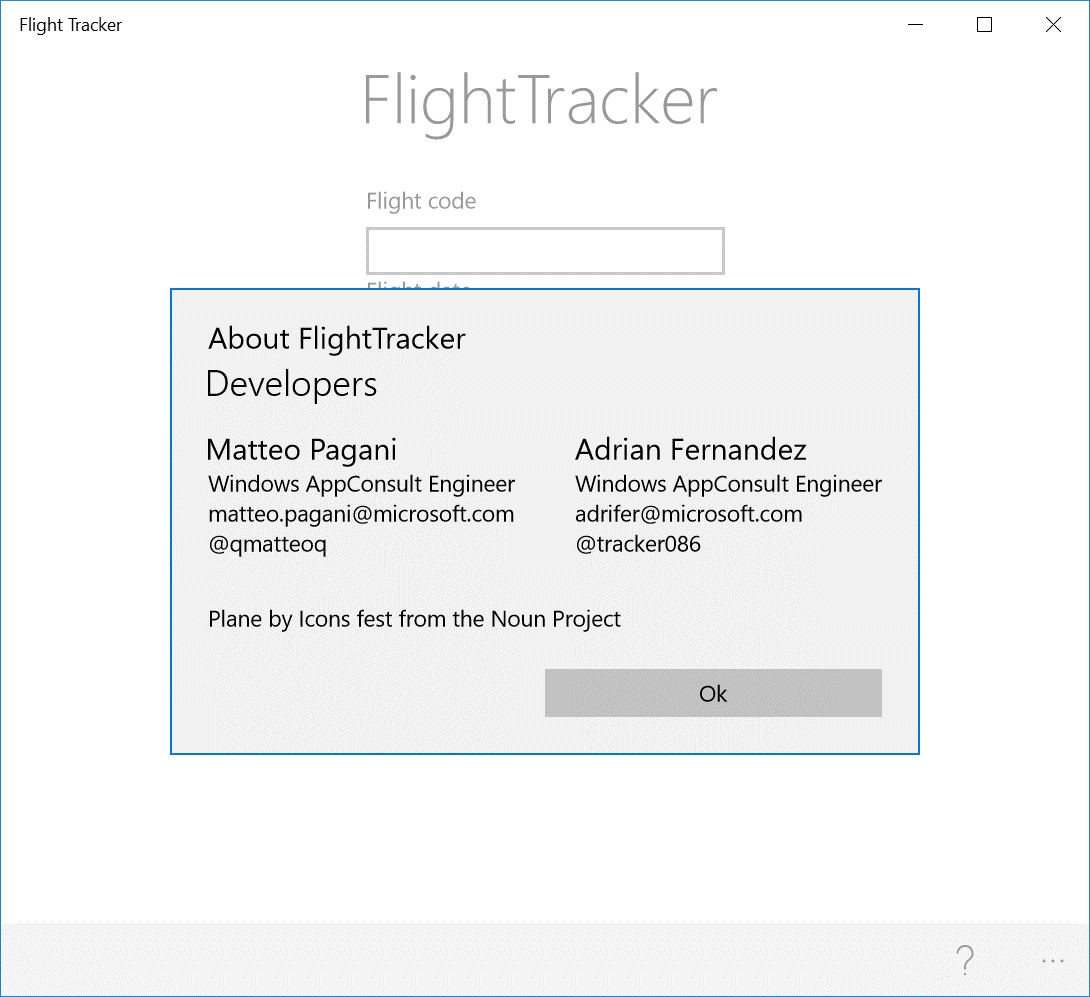
{

AboutPage aboutPage = new AboutPage();

await aboutPage.ShowAsync();

}

We just create a new instance of the control we have just created and we call the **ShowAsync()** method. The dialog will be displayed in overlay in the middle of the page, like in the following image:



## Task 3: Saving and loading the data about the flight

From a code point of view, since in the Exercise 3 (Extend) we have already changed the way we store the info about the flight (the local storage instead of the registry), we can reuse the same code. Consequently, we can define the handler of the **Click** event of the **Save** button in the **MainPage.xaml.cs** file with the following code, which is the same we had in the Windows Forms apps:

private void OnSaveFlight(object sender, RoutedEventArgs e)

{

ApplicationData.Current.LocalSettings.Values["Code"] = codeTextbox.Text;

ApplicationData.Current.LocalSettings.Values["Date"] = dateTextbox.Text;

ApplicationData.Current.LocalSettings.Values["Departure"] = departureTextbox.Text;

ApplicationData.Current.LocalSettings.Values["Arrival"] = arrivalTextbox.Text;

operationStatusLabel.Text = "The flight has been saved";

}

To avoid compilation errors, make sure to add the proper namespace in the header of the class:

using Windows.Storage;

Again, the operation to retrieve the data stored in the local storage, if exists, is the same we defined in the Windows Forms app. The difference is that, in the Windows Forms app, we were using the **Load** event offered by the form. In the Universal Windows Platform, instead, we can leverage the **OnNavigatedTo()** method exposed by the page in the code behind, which is triggered when the user navigates to the current page. As such, we can add the following code in the **MainPage.xaml.cs** file:

protected override void OnNavigatedTo(NavigationEventArgs e)

{

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Code"))

{

codeTextbox.Text = ApplicationData.Current.LocalSettings.Values["Code"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Date"))

{

dateTextbox.Text = ApplicationData.Current.LocalSettings.Values["Date"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Departure"))

{

departureTextbox.Text = ApplicationData.Current.LocalSettings.Values["Departure"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Arrival"))

{

arrivalTextbox.Text = ApplicationData.Current.LocalSettings.Values["Arrival"].ToString();

}

}

## Task 4: Handling the background task

Now we need to reconnect the background task we have created in the Exercise 3, which you will already find included in the starting solution. The code to register the task is the same we have used in the Windows Forms app in the Exercise 3. The difference is that, this time, we can leverage the same **OnNavigatedTo()** method we have just defined in the previous step, right after the code we have just included to populate the form with the info stored in the local storage. The code we have added compared to the previous snippet is highlighted in yellow:

protected override async void OnNavigatedTo(NavigationEventArgs e)

{

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Code"))

{

codeTextbox.Text = ApplicationData.Current.LocalSettings.Values["Code"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Date"))

{

dateTextbox.Text = ApplicationData.Current.LocalSettings.Values["Date"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Departure"))

{

departureTextbox.Text = ApplicationData.Current.LocalSettings.Values["Departure"].ToString();

}

if (ApplicationData.Current.LocalSettings.Values.ContainsKey("Arrival"))

{

arrivalTextbox.Text = ApplicationData.Current.LocalSettings.Values["Arrival"].ToString();

}

//we register the background task that is triggered every time the time zone of the device changes

string triggerName = "FlightTimeZoneTrigger";

// Check if the task is already registered

foreach (var cur in BackgroundTaskRegistration.AllTasks)

{

if (cur.Value.Name == triggerName)

{

// The task is already registered.

return;

}

}

BackgroundTaskBuilder builder = new BackgroundTaskBuilder();

builder.Name = triggerName;

builder.TaskEntryPoint = "DesktopBridge.FlightTracker.Notification.ToastTask";

builder.SetTrigger(new SystemTrigger(SystemTriggerType.TimeZoneChange, false));

builder.Register();

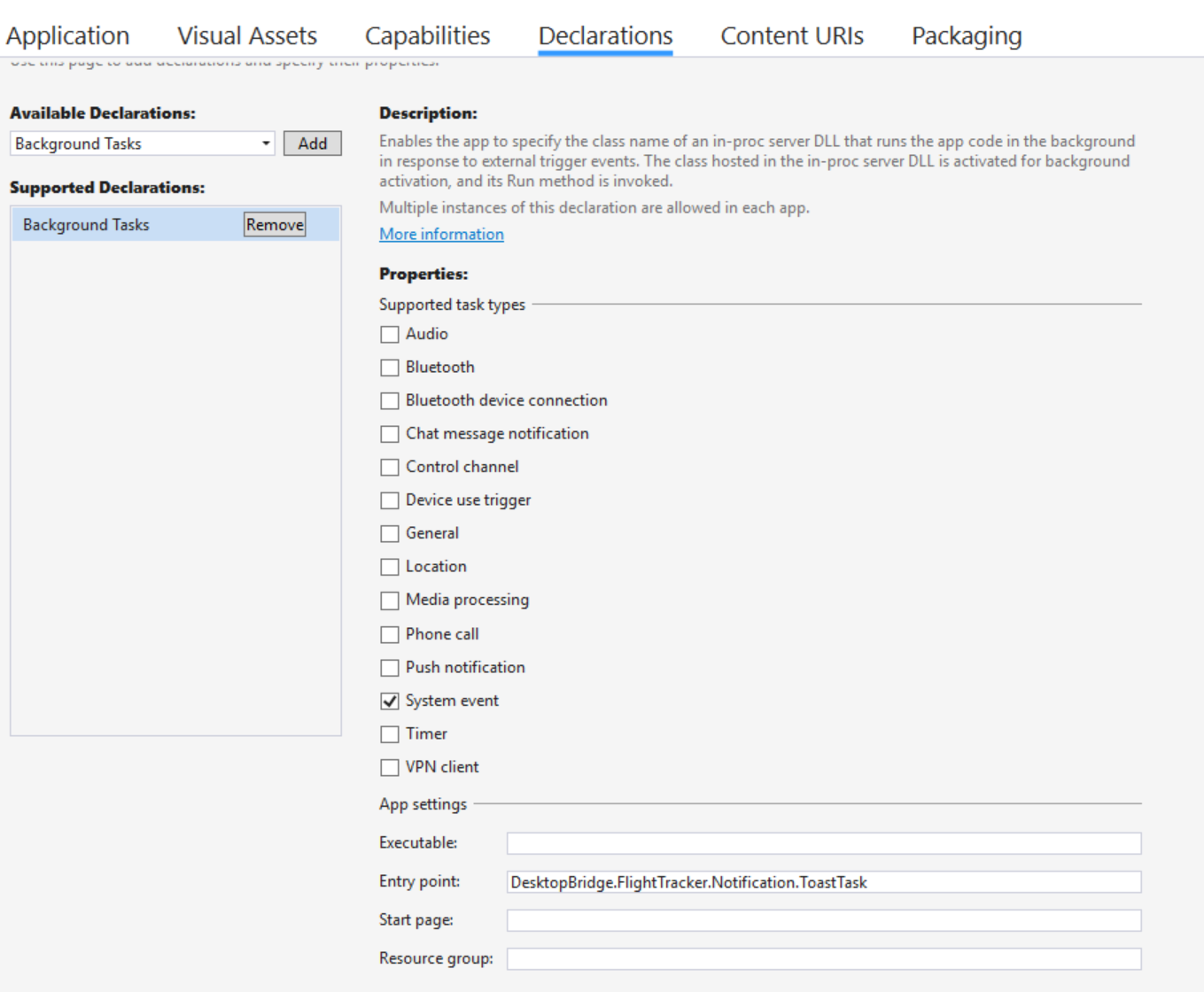
}

Also in this case, to avoid compilation errors, we need to add the proper namespace in the class header:

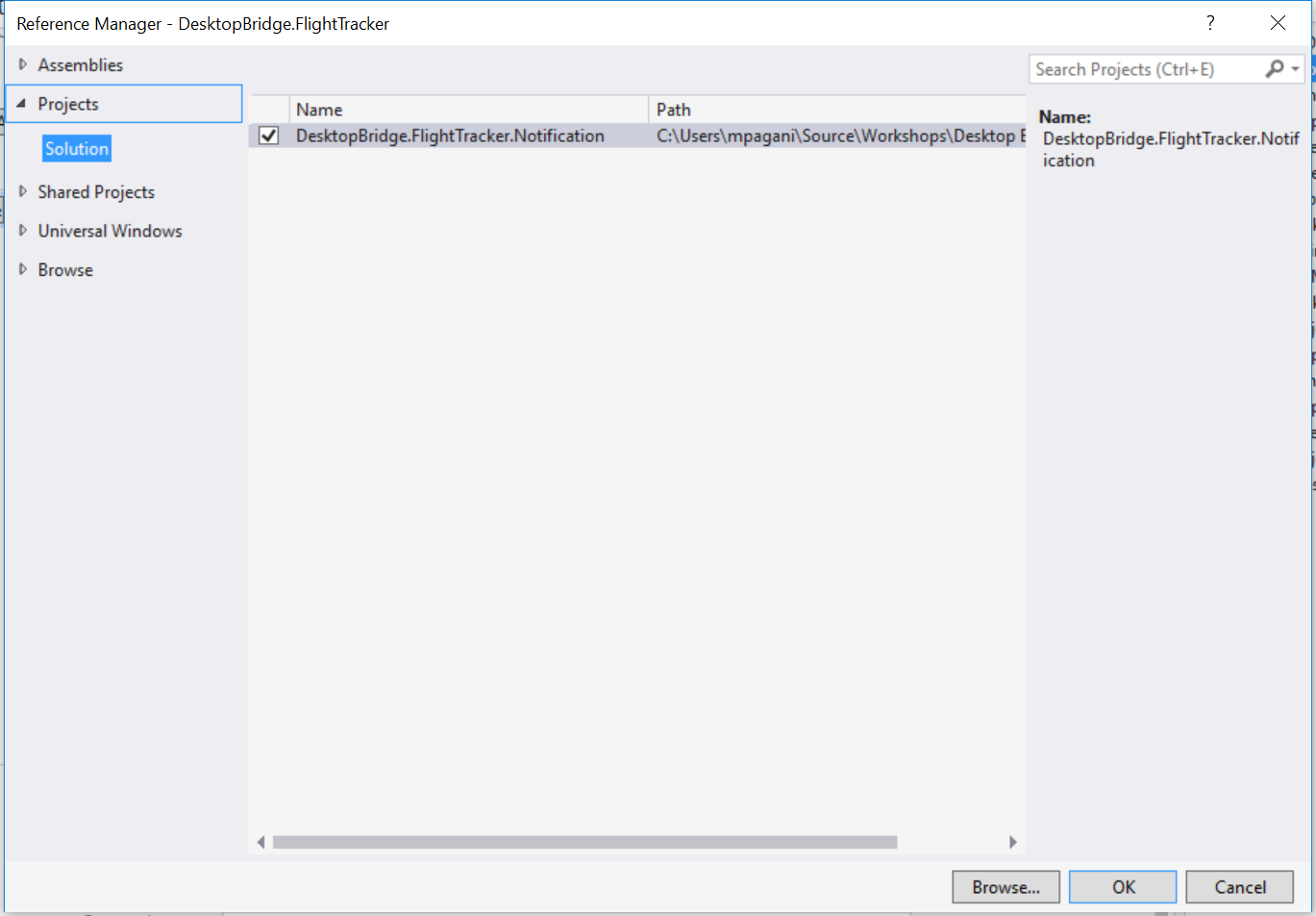
using Windows.ApplicationModel.Background;

If you remember what we did in Exercise 3, we had also to register the task in the manifest. We can repeat the same exact steps we have followed with the visual editor offered by the WinJS project. Double click on the file called **Package.appxmanifest** and move to the section titled **Declarations**. You will find a dropdown menu titled **Available Declarations**, with all the supported extensions by the Universal Windows Platform: the one we need to add is called **Background Tasks**. In the page, we must configure two options:

1. We need to choose which is the task type. In our case, it’s a **System event** one.
2. In the **Entry point** we need to specify the full signature of the class that implements the task, which is **DesktopBridge.FlightTracker.Notification.ToastTask.**



The last step to register the task is to add a reference to the background task in the main application. Consequently, right click on the Universal Windows App project, choose **Add reference** and, in the **Projects -> Solution**, choose the project that contains the definition of the task, which is called **DesktopBridge.FlightTracker.Notification**.



## Task 5: Integrating the boarding pass export feature

If you want to keep the boarding pass export feature as it was in the original desktop version of the app, you’ll hit a blocker: a UWP app can’t write a file directly on the desktop. This is where the Desktop Bridge comes in: the UWP application has the chance to launch a Win32 process and to use a new feature introduced in the Universal Windows Platform called App Services. This technology allows a UWP app to expose a background task that isn’t connected to a specific trigger (like the one we’ve seen in Exercise 3, which was invoked every time the time zone of the device changed), but it offers a set of features or operations that can be leveraged by the other applications installed on the same device. Any third-party application can establish a communication channel with the background task, send some data, wait for the task to perform some operations on them and then receive the result back. In the context of the Migrate scenario we’re learning in this exercise, an App Service is the way we can handle the communication between the UWP app and the Win 32 app. Messages between the two apps are exchanged using a class called **ValueSet**, which is a collection of key / value pairs.

Consequently, in our exercise we’re going to:

1. Create a Win32 app which will act a background process.
2. Create in the UWP app an App Service, which will be used as a channel to send to the Win32 app the info about the flight.
3. The Win32 app running in background will establish the communication with the app service, retrieve the data about the flight, generate the boarding pass on the desktop and then send a confirm to the UWP that the operation has been completed.

To complete the exercise, we’re going to use also a new feature introduced in the Anniversary Update, called **Background Single Process Model**. In the past, when we wanted to implement a background task, the only option was to use the same approach we’ve learned in Exercise 3: add a separate project to our solution of type Windows Runtime Component, create a new class which implements the **IBackgroundTask** interface and then define the **Run()** method, which contains the actual code that is executed when the task is triggered.

With this new approach, instead, we aren’t forced anymore to create a separate Windows Runtime Component, but we can define the method to execute in background directly in the UWP application. In this scenario, the **Run()** method is replaced by another one called **OnBackgroundActivated()** that we can define in the **App** class. We’re going to see the details later, when we’ll have to implement it in our new UWP app.

Let’s see the required steps to implement this scenario.

## Task 6: Create the Win32 process

First right click on the Solution, choose **Add -> New project** and, from the **Visual C# -> Windows Classic Desktop** section, choose **Console App (.NET Framework)**. We’re going to call it **DesktopBridge.FlightTracker.Export.**

As first thing, right click on the project you have just created and, in the **Application** section, change the **Output Type** from **Console Application** to **Windows Application**. This will allow us to turn the console project into a process capable of running in background, without displaying any user interface.

The next step is the same we did in all the previous exercises (starting from the Exercise 2 – Enhance) to leverage the Universal Windows Platform in a desktop application. Right click on the project, choose **Add reference**, press the **Browse** button and add a reference to the following libraries:

1. **Windows.md**, stored in **C:\Program Files (x86)\Windows Kits\10\UnionMetadata.**
2. **System**.**Runtime.WindowsRuntime.dll**, stored in **C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5.1**

Now we can start working on the **Program.cs** file, which is the class that will contain all the code that will be executed when the process will be launched. First, we need to declare two static variables in the class:

static AppServiceConnection connection = null;

static AutoResetEvent appServiceExit;

These two classes require the definition of two specific namespaces in the header of the **Program.cs** class:

using System.Threading;

using Windows.ApplicationModel.AppService;

**AppServiceConnection** is part of the Universal Windows Platform and it’s the API needed to establish a connection using an App Service. **AutoResetEvent**, instead, is part of the thread management APIs available in the .NET Framework. We’re going to use it to keep the application running until the connection to the App Service is alive: without it, the Win32 process would start and immediately be terminated.

Now we need to define the **Main()** method, which is the one executed when the Win32 process is launched. Here is the code to include:

static void Main(string[] args)

{

//we use an AutoResetEvent to keep to process alive until the communication channel established by the App Service is open

appServiceExit = new AutoResetEvent(false);

Thread appServiceThread = new Thread(new ThreadStart(ThreadProc));

appServiceThread.Start();

appServiceExit.WaitOne();

}

First we initialize the **AutoResetEvent** object and then we call the **WaitOne()** method: this way, the process will keep running until we call the **Set()** method, which instead will release the thread, allowing the process to be terminated by Windows. Then we start a new thread by initializing a **Thread** object passing, as parameter, a **ThreadStart** object with the name of the method we want to execute inside it (in this case, it’s called **ThreadProc**). As we’re going to see in the next step, **ThreadProc** is the name of the method that will execute all the code required to establish the connection with the App Service. In the end, we start the thread by calling the **Start()** method.

As already anticipated, the **ThreadProc()** method is the one that takes care of establishing the connection with the App Service declared by the UWP app. Here is the definition that you need to include in the **Program.cs** file:

static async void ThreadProc()

{

//we create a connection with the App Service defined by the UWP app

connection = new AppServiceConnection();

connection.AppServiceName = "BoardingPassService";

connection.PackageFamilyName = Windows.ApplicationModel.Package.Current.Id.FamilyName;

connection.RequestReceived += Connection\_RequestReceived;

connection.ServiceClosed += Connection\_ServiceClosed;

//we open the connection

AppServiceConnectionStatus status = await connection.OpenAsync();

if (status != AppServiceConnectionStatus.Success)

{

//if the connection fails, we terminate the Win32 process

appServiceExit.Set();

}

else

{

//if the connection is successful, we communicate to the UWP app that the channel has been established

ValueSet initialStatus = new ValueSet();

initialStatus.Add("Status", "Ready");

await connection.SendMessageAsync(initialStatus);

}

}

The **ValueSet** class isn’t part of the default UWP namespace, so you’ll have to add the following the definition in the header of the class to use it:

using Windows.Foundation.Collections;

The previous code will take care to:

1. Establish a new **AppServiceConnection** object, by setting the two properties which are required to identify the App Service we want to use: the name (**AppServiceName**) and the package family name (**PackageFamilyName**), which is the unique identifier of the UWP app that has registered the App Service. Since the Win32 process will be embedded in the same AppX package of the UWP app, they will both share the same identity: this is why, instead of setting the **PackageFamilyName** property with a fixed string, we can use the **Windows.ApplicationModel.Package.Current.Id.FamilyName**, which is a UWP API that contains the **PackageFamilyName** of the current running app.
2. Subscribe to two events offered by the connection:
   1. **RequestReceived**, which is invoked when another application is requesting to establish a connection with the App Service.
   2. **ServiceClosed**, which is invoked when the communication channel that has been established with another application is closed.
3. Open a connection channel with the App Service, by calling the **OpenAsync()** method.

When we try to establish the connection, we will get in return a value of the **AppServiceConnectionStatus** enumerator with the outcome of the operation: if we get a value different than **Success**, it means that the connection has failed, so we call the **Set()** method on the **AutoResetEvent** object. This operation will allow Windows to terminate the process. Otherwise, it means that the connection with the UWP app has been established and we can send a message to alert it that we are ready to communicate: as previously mentioned, messages are exchanged using the **ValueSet** class, so we create a new instance and we add a new item, with value **Ready**, identified by the key **Status**.

Let’s now, in details, how to handle the two events exposed by the **AppServiceConnection** class we have subscribed to: **RequestReceived** and **ServiceClosed.**

### Receiving a message from the UWP app

We already mentioned that the **RequestReceived** event is trigged when the UWP app has sent a message to the Win32 process: in our scenario, this will happen when the user will press the Export button, which will open a connection to the process to share the data about the flight.

This is how the method that handles the **RequestReceived** event looks like, which you need to include in the **Program.cs** file:

private static async void Connection\_RequestReceived(AppServiceConnection sender, AppServiceRequestReceivedEventArgs args)

{

//we receive the data about the flight from the UWP app and we generate the boarding pass on the desktop

string flightCode = args.Request.Message["Code"].ToString();

string flightDate = args.Request.Message["Date"].ToString();

string departureCity = args.Request.Message["Departure"].ToString();

string arrivalCity = args.Request.Message["Arrival"].ToString();

GenerateBoardingPass(flightCode, flightDate, departureCity, arrivalCity);

//we send a message back to the UWP app to communicate that the operation has been completed with success

ValueSet set = new ValueSet();

set.Add("Status", "Success");

await args.Request.SendResponseAsync(set);

}

The message created by the UWP app is stored inside the parameter of the method: specifically, we can retrieve it in the **Message** property of the **Request** object, which is a **ValueSet**. As such, we can extract the info that the UWP has sent us about the flight using the various keys, like **Code, Date, Departure** and **Arrival**. Then we pass all these info to a method called **GenerateBoardingPass()**, which will take care of generating the boarding pass and saving into the user’s desktop. In the end, we send back a new message to the UWP app to say that the operation has been completed.

The **GenerateBoardingPass()** method shouldn’t pose any surprise: it’s the part of code that we can reuse from the original Windows Forms app we have used as a starting point for our journey. In fact, it does the same operation that, in the original app, it was performed by the Export button: it generates a string with all the info about the flight and it saves it into a text file on the desktop. We must include also this method in the **Program.cs** file:

*Snippet 4.6.7*

private static void GenerateBoardingPass(string flightCode, string flightDate, string departureCity, string arrivalCity)

{

//we generate the boarding pass on the desktop of the user

string userPath = Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory);

string fileName = $"{userPath}\\BoardingPass.txt";

var builder = new StringBuilder();

builder.AppendLine("Boarding pass generated by FlightTracker");

builder.AppendLine("-----------------------------------------");

builder.AppendLine($"Flight code: {flightCode}");

builder.AppendLine($"Flight date: {flightDate}");

builder.AppendLine($"Departure city: {departureCity}");

builder.AppendLine($"Arrival city: {arrivalCity}");

builder.AppendLine("-----------------------------------------");

builder.AppendLine("Thank you for using FlightTracker");

File.WriteAllText(fileName, builder.ToString());

}

To use the **File** class to create the text file we need to add its namespace in the class header:

using System.IO;

### Closing the connection

When the user will close the UWP app, it will automatically close also the connection to the App Service. In this case, the **ServiceClosed** event of the **AppServiceConnection** object will be triggered. We can use it to call the **Set()** method on the **AutoResetEvent**, so that Windows can terminate the process. If we don’t perform this operation, the Win32 process will keep running in background indefinitely, even if the UWP app has been closed.

Here is the definition of the event handler that you will need to include in the **Program.cs** file:

private static void Connection\_ServiceClosed(AppServiceConnection sender, AppServiceClosedEventArgs args)

{

//when the connection with the App Service is closed, we terminate the Win32 process

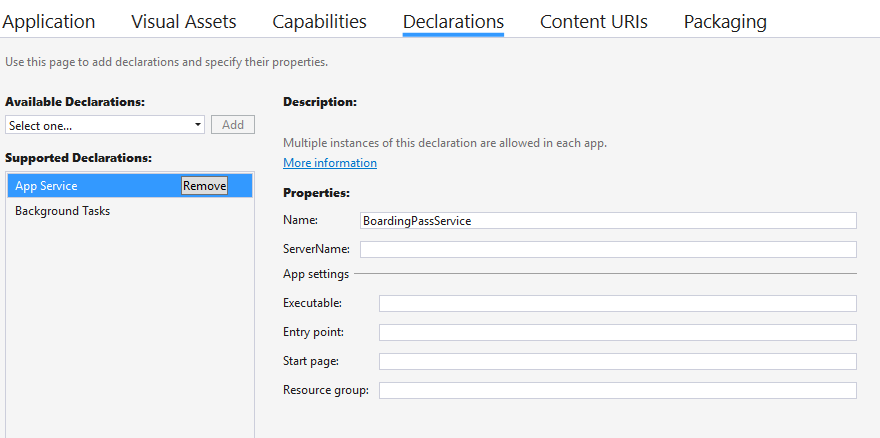
appServiceExit.Set();

}

## Task 7: Create the App Service in the UWP app

The next task is to configure the App Service in the UWP app and to handle the communication with the Win32 process we have created in the previous task.

The first step to create an App Service is to define it in the manifest of the application, so double click on the **Package.appxmanifest** and move again to the **Declarations** tab. This time, from the dropdown menu, we must choose the **App Service** option and configure it. The only field we must fill is the **Name** one, which contains the name of the service (do you remember the **AppServiceName** property of the **AppServiceConnection** class that we have configured in the Win32 process?). Unlike we did with the background task connected to the **TimeZoneTrigger**, in this case we don’t have to specify an entry point, since we are going to leverage the **Background Single Process Model** we explained in the beginning. The task definition won’t be included in a separate project of the solution, but in the app itself, se we don’t have to specify which is the class that implements the **Run()** method.



We’re going to handle the App Service configuration in the **App** class, so double click on the **App.xaml.cs** file and start adding the following namespaces in the header of the class:

using Windows.ApplicationModel.AppService;

using Windows.ApplicationModel.Background;

Thanks to these namespaces, we’ll be able to add the following properties in the **App** class:

public event EventHandler<string> StatusUpdated;

public static AppServiceConnection Connection = null;

private BackgroundTaskDeferral appServiceDeferral = null;

* The first property is an **EventHandler**, which we’re going to invoke when the connection with the service has been established. We’re going to leverage it in the **MainPage** of the app, to get notified when the App Service is ready to be used.
* The second property is an **AppServiceConnection**: we’ve already learned its purpose in the Win32 process. It’s the class that we can use to handle the connection with the App Service.
* The third property is a **BackgroundTaskDeferral**, which we will use to avoid that Windows terminates the App Service while the communication is still in progress.

The next step is to define the code to invoke when the App Service is invoked: as we mentioned in the beginning of the exercise, since we’re leveraging the **Background Single Process Model**, we can do it directly in the **App** class by overwriting a method called **OnBackgroundActivated().** Here is the definition to include in your **App.xaml.cs** file:

protected override void OnBackgroundActivated(BackgroundActivatedEventArgs args)

{

//this method is invoked when the Win32 process requests to open the communication channel with the App Service

base.OnBackgroundActivated(args);

if (args.TaskInstance.TriggerDetails is AppServiceTriggerDetails)

{

appServiceDeferral = args.TaskInstance.GetDeferral();

AppServiceTriggerDetails details = args.TaskInstance.TriggerDetails as AppServiceTriggerDetails;

Connection = details.AppServiceConnection;

//we subscribe to the RequestReceived event, which is triggered when the Win32 app sends some data through the channel

Connection.RequestReceived += Connection\_RequestReceived;

}

}

The first step is to check if the background activation has really been triggered by an App Service, by checking the type of the **TriggerDetails** property of the **TaskInstance** object: it must be **AppServiceTriggerDetails**. If that’s the case, we can use this property to retrieve the **AppServiceConnection** object and store it in the **Connection** property we’ve previously created, so that we can reuse it later. Then, we perform an operation that we did also in the Win32 process: we subscribe to the **RequestReceived** event. The communication between the UWP app and the process, in fact, will be bidirectional, so both applications needs a way to be notified when one side of the channel has sent some data to the other.

In our scenario, since the App Service usage will be connected to the Export button in the main page, we will leverage this event to send a notification that the App Service is ready to be used, thanks to the event handler called **StatusUpdated** we’ve created before. Here is the definition of the event handler you need to include in your **App.xaml.cs** file:

private void Connection\_RequestReceived(AppServiceConnection sender, AppServiceRequestReceivedEventArgs args)

{

//the Win32 app has sent us a message. If it's the message to confirm that the channel has been open, we raise the StatusUpdated event

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

{

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

StatusUpdated?.Invoke(this, currentStatus);

}

}

This is the scenario where the Win32 process we’ve created in the Task 6 has started, has established the connection with the App Service and has sent us a message using a **ValueSet** collection to say that it’s ready to receive the information about the flight. Consequently, we check if the **Message** collection of the **Request** object contains the item identified by the **Status** key we’re expecting: if that’s the case, we raise the **StatusUpdated** event, so that we can send a notification to every other class that has subscribe to it (in our case, it will be the main page).

We’ve completed the work in the **App** class. Let’s take look now at the changes to apply in the **MainPage.xaml.cs** file.

The first step is to display the **ProgressRing** indicator until the Win32 process has been initialized: as such, we add the following line of code at the beginning of the **OnNavigatedTo()** method:

progressBar.IsActive = true;

Then we can leverage the **StatusUpdated** event we have defined in the **App** class to get notified when the initialization has been completed and, as such, we can hide the **ProgressRing** control. We subscribe to this event in the constructor of the **MainPage** class, like in the following snippet (highlighted in yellow you can see the line we have added compared to the original code):

public MainPage()

{

this.InitializeComponent();

(Application.Current as App).StatusUpdated += MainPage\_StatusUpdated;

}

Here is, instead, the definition of the handler of the event:

private async void MainPage\_StatusUpdated(object sender, string e)

{

//the Win32 app has initialized the channel with the App Service, so we hide the ProgressRing

await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>

{

progressBar.IsActive = false;

});

}

We simply stop the indicator by setting its **IsActive** property to **false**. The only peculiar aspect of the previous code is that we perform this operation inside the **RunAsync()** method of the **Dispatcher**. The reason is that the event is raised by a background thread and, as such, it doesn’t have access to the UI thread: interacting with a control in the user interface would lead to an exception. The **Dispatcher**, instead, can dispatch an operation from a background thread to the UI thread, solving our problem.

Now we can return to the **OnNavigatedTo()** method and start adding some other changes: in our solution we have created a Win32 process and we have configured an App Service, but it isn’t enough because the UWP app automatically launches the process. We explicitly need to start it so, right after the initialization of the flight data from the local storage and before the initialization of the background task connected to the **TimeZoneTrigger**, we need to add the following block of code:

//we check if the app is running on the desktop: only if that's the case, we leverage the Desktop Bridge specific features

if (AnalyticsInfo.VersionInfo.DeviceFamily == "Windows.Desktop")

{

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

await FullTrustProcessLauncher.LaunchFullTrustProcessForCurrentAppAsync();

//we display the export button if the app is running on the desktop, since it's the only Windows 10 platform which supports running Win32 applications

exportButton.Visibility = Visibility.Visible;

}

To avoid compilation errors, we need to add the following namespace in the header:

using Windows.ApplicationModel;  
using Windows.System.Profile;

We need also to mark the **OnNavigatedTo()** method with the **async** keyword, since now we’re invoking an asynchronous method:

protected override async void OnNavigatedTo(NavigationEventArgs e)

{

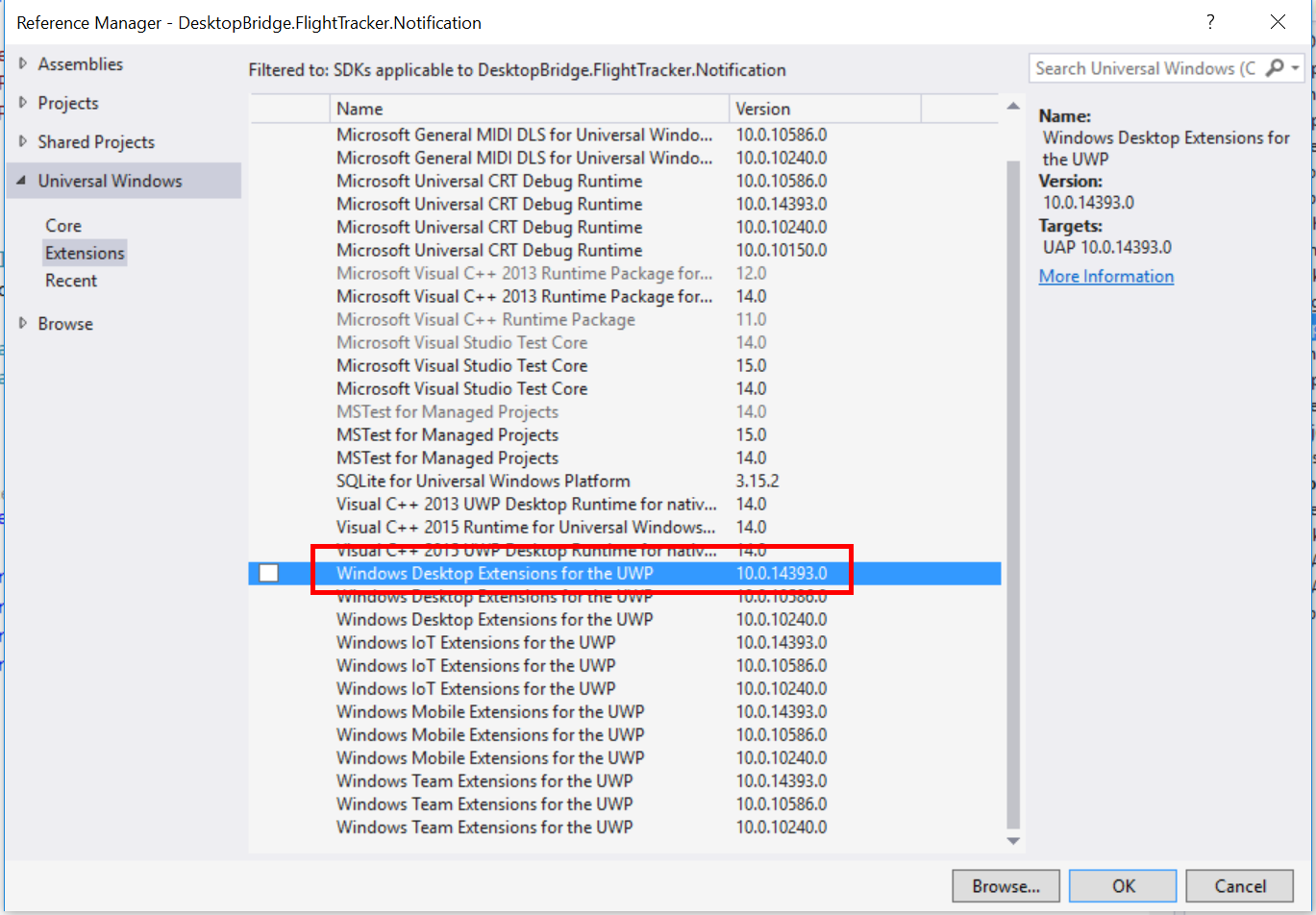
//content of the method

}

We need to remember that the Desktop Bridge is supported only by the full Windows 10 version, since it’s the only platform that, in addition to the Universal Windows Platform, it includes the standard Windows APIs that are leverage by every other development technology (like the .NET Framework, Java, Delphi, etc.). Consequently, we’re going to enable the Export feature only if the app is running on a desktop, a tablet or a notebook: we can get this information thanks to the **AnalyticsInfo.VersionInfo.DeviceFamily** property, which is part of the **Windows.System.Profile** namespace. If the app is running on a platform supported by the Desktop Bridge, we will get the string **Windows.Desktop** as a return value. If that’s the case, we perform two operations:

1. We invoke the **LaunchFullTrustProcessForCurrentAppAsync()** method of the **FullTrustProcessLauncher** class (which belongs to the **Windows.ApplicationModel** namespace). This is the method that is going to launch the **DesktopBridge.FlightTracker.Export.exe** executable we have created in Task 6.
2. We make the Export button visible, by changing its **Visibility** property to **Visible**.

However, if you copy and paste the previous code snippet in your application you will notice an error when you will try to use the **FullTrustProcessLauncher** class: Visual Studio will complain about the lack of this class in the **Windows.ApplicationModel** namespace. The reason is that the **FullTrustProcessLauncher** class can be leveraged only on a desktop and, as such, it isn’t part of the standard Universal Windows Platform, but it’s included in the specific Extension SDK for the desktop family. Consequently, to use it, you first need to right click on the UWP app project and choose **Add reference**: you will find a section called **Universal Windows -> Extensions**. Inside this category, you will find all the Extension SDKs for all the platforms supported by Windows 10, like mobile, IoT, desktop, etc. The one we need to use is called **Windows Desktop Extensions for the UWP**. In case you have multiple versions of the Windows 10 SDK, make sure to add the one identified by version **10.0.14393.0** or **10.0.15063.0**. The **FullTrustProcessLauncher** class, in fact, is part of the new features added starting from Anniversary Update and, as such, you won’t find it in the previous SDK versions.



What happens when we click on the Export button? We need to communicate to the App Service through the channel we have opened in the **App** class and send to the Win32 process the data about the flight, so that it can generate the boarding pass for us. Here is the definition of the event handler connected to the **Click** event of the Export button:

private async void OnExportBoardingPass(object sender, RoutedEventArgs e)

{

progressBar.IsActive = true;

//we simulate that the exporting operation takes a while to be completed

await Task.Delay(5000);

//if the connection with the App Service has been established, we send the info about the flight to the Win32 process

if (App.Connection != null)

{

ValueSet set = new ValueSet();

set.Add("Code", codeTextbox.Text);

set.Add("Date", dateTextbox.Text);

set.Add("Departure", departureTextbox.Text);

set.Add("Arrival", arrivalTextbox.Text);

AppServiceResponse response = await App.Connection.SendMessageAsync(set);

//if the Win32 process has received the data and it has successfully generated the boarding pass, we show a notification to the user

if (response.Status == AppServiceResponseStatus.Success)

{

string status = response.Message["Status"].ToString();

if (status == "Success")

{

ShowNotification();

}

}

}

progressBar.IsActive = false;

}

The beginning and the end of the method are the same of the original Windows Forms app: when the operation starts, we display a progress indicator, that then it’s hidden when the operation is completed. Also in this case, since we want to simulate that the export process may take a while to be performed, we simulate a 5 seconds’ delay, by calling the **Task.Delay()** method.

The rest of the code is the core of the method, since it’s the one that handles the communication with the Win32 process through the App Service. As first thing, we check if the connection has been successfully established, by checking that the **Connection** property we have previously defined in the **App** class is not **null**. If everything is ok, we can send to the Win32 process the info about the flight: we create a **ValueSet** collection and we add multiple items, one for each of the fields. Then we send the message to the process by passing the collection to the parameter of the **SendMessageAsync()** method exposed by the connection.

That’s all: if everything has worked as expected, the **RequestReceived** event in the Win32 process will be raised, it will receive the data about the flight and it will generate the boarding pass on the desktop. If you remember what we did in Task 6, at the end of the boarding pass generation we were sending a message back to the UWP with the key **Status** and the value **Success**. If that’s the case, it means that the operation has been performed successfully: we can call the **ShowNotification()** method to notify to the user about the outcome of the operation.

The definition of the **ShowNotification()** method is exactly the same we have added in Exercise 2, when we have started to enhance our Windows Forms app with UWP APIs. The only difference is that, this time, since it’s a true UWP app, we won’t need any more to use conditional compilation, so you can just add the method to the **MainPage.xaml.cs** file as it is:

private void ShowNotification()

{

string xml = $@"<toast>

<visual>

<binding template='ToastGeneric'>

<text>Flight Tracker</text>

<text>The boarding pass for flight {codeTextbox.Text} from {departureTextbox.Text} to {arrivalTextbox.Text} has been exported on your desktop</text>

</binding>

</visual>

</toast>";

XmlDocument doc = new XmlDocument();

doc.LoadXml(xml);

ToastNotification toast = new ToastNotification(doc);

ToastNotificationManager.CreateToastNotifier().Show(toast);

}

To avoid compilation errors, we need to add a few namespaces, since we have added a set of new UWP API’s we haven’t used so far:

using Windows.ApplicationModel.AppService;  
using System.Threading.Tasks;  
using Windows.Data.Xml.Dom;  
using Windows.UI.Notifications;

## Task 8: Configure the manifest

There are a couple of steps to perform in the exercise to make the whole solution working: the first one is to make some changes in the manifest file. Do you remember that, in *snippet 4.7.8*, we have added the code required to launch the Win32 process? However, so far, we haven’t specified which is the executable that we want to launch. We must do it in the manifest, thanks to a special extension offered by the Desktop Bridge.

To leverage it, we need to open the **Package.appxmanifest** file with a code editor, since it isn’t supported by the visual editor. The easiest way is to right click on the file in Visual Studio and choose **View code**: you will see the original XML of the manifest.

The first operation to do is to add a set of additional namespaces to the root **<Package>** node, since the extensions we’re going to add aren’t included in the default ones. Here is how the **<Package>** node should look like after your changes:

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

xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"

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

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

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

IgnorableNamespaces="uap mp rescap desktop"

<!--content of the manifest -->

</Package>

Highlighted in yellow you can find the parts that have been changed compared to the original definition**.**

The next change to apply is inside the **<Extensions>** section, which you will find inside the **<Application>** node. You will already find two extensions declared: they’re the background task and the App Service that we have previously defined using the visual editor. However, we need to add a new one, that you’ll find highlighted in yellow in the snippet below:

<Extensions>

<uap:Extension Category="windows.appService">

<uap:AppService Name="BoardingPassService" />

</uap:Extension>

<Extension Category="windows.backgroundTasks" EntryPoint="DesktopBridge.FlightTracker.Notification.ToastTask">

<BackgroundTasks>

<Task Type="systemEvent" />

</BackgroundTasks>

</Extension>

<desktop:Extension Category="windows.fullTrustProcess" Executable="DesktopBridge.FlightTracker.Export.exe" />

</Extensions>

The first thing to highlight is that, since it isn’t a standard extension, we need to declare it using the **desktop** prefix**,** which we have previously defined as a new namespace. The extension has two attributes:

1. **Category** is the extension’s type and it’s a fixed string, which is **windows.fullTrustProcess**.
2. **Executable**, instead, is the name of the Win32 process, which is **DesktopBridge.FlightTracker.Export.Exe**.

When, in the code of the **OnNavigatedTo()** method of the **MainPage**, we’re calling the **LaunchFullTrustProcessForCurrentAppAsync()** method of the **FullTrustProcessLauncher** class, the UWP app is going to launch this executable we have just defined in the manifest.

The other change to apply in the manifest is about capabilities: since we’re leveraging the Desktop Bridge, we still need the **runFullTrust** capability, so we need to add it in the **<Capabilities>** section in addition to the already existing ones.

<Capabilities>

<Capability Name="internetClient" />

<rescap:Capability Name="runFullTrust" />

</Capabilities>

Also in this case, we need to use a special prefix called **rescap**, since the capability isn’t part of the standard ones.

## Task 9: Deploy the Win32 process in the AppX package

The last step to do is to include the Win32 process we have created in the folder which is created when we compile the UWP app and that it will become an AppX package: this means that the output of the UWP project build, other than the specific files of a UWP app, must contain our **DesktopBridge.FlightTracker.Export.exe** executable.

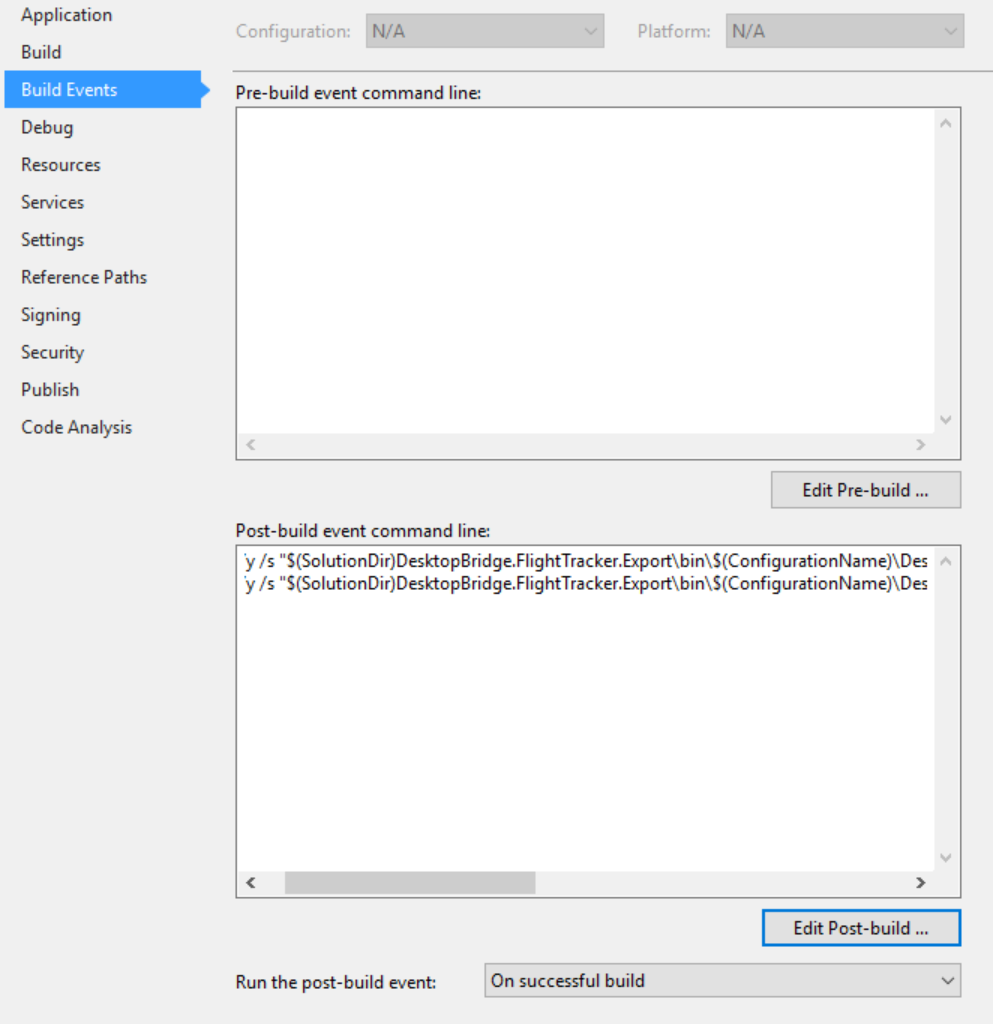
However, we can’t include it directly into the project of the UWP app, otherwise Visual Studio will try to compile it, generating all sort of errors. We need to copy it manually: the easiest way to make sure that, every time we make some changes to the Win32 process, the output is copied into the proper directory, is to define a post-build command, like in Exercise 3 we did for the background task. This way, the new executable will be copied every time we build the Win32 process.

To define the command, we need to right click on the **DesktopBridge.FlightTracker.Export** project, choose **Properties** and move to the **Build events** section. In the area titled **Post-build event command line** we need to specify the following commands:

xcopy /y /s "$(SolutionDir)DesktopBridge.FlightTracker.Export\bin\$(ConfigurationName)\DesktopBridge.FlightTracker.Export.exe" "$(SolutionDir)DesktopBridge.FlightTracker\bin\x64\$(ConfigurationName)\AppX\"

xcopy /y /s "$(SolutionDir)DesktopBridge.FlightTracker.Export\bin\$(ConfigurationName)\DesktopBridge.FlightTracker.Export.exe" "$(SolutionDir)DesktopBridge.FlightTracker\bin\x86\$(ConfigurationName)\AppX\"

The two commands leverage the **xcopy** utility to copy the output of the build of the Win32 project (the **DesktopBridge.FlightTracker.Export.exe** file) inside the folder which contains the compiled version of the UWP project.



There’s only one caveat to pay attention to: the **AppX** folder inside the *\bin\x86\$(ConfigurationName)\* path is generated only when you have deployed your UWP app at least once, it isn’t enough just to build it. As such, before building the Win32 process, make sure that you have at least deployed and executed at least once the **DesktopBridge.FlightTracker** project, otherwise the post-build command will fail, since the AppX folder won’t exist.

That’s all: now, if you have properly followed the tutorial, you should try to launch the UWP app, insert the info about the flight and Save them. Then press the Export button, which will:

1. Establish a communication with the Win32 process thanks to the App Service.
2. Send the info about the flight to the Win32 process.
3. The Win32 process will activate itself, thanks to an even offered by the App Service, and it will take of receiving the info about the flight, generate the boarding pass and send a message back to the UWP to share that the operation has completed successfully.
4. The UWP will receive the message and will display a notification to the user, to alert him that his boarding pass is ready.

# Exercise 5 – Reach all

We have reached the end of our journey: thanks to this exercise, we’ll be able to remove the dependency from the Win32 process that we have created in the Exercise 4, making the app a true Universal Windows Platform one, which doesn’t require anymore any special capability and that can run exactly in the same way on every Windows 10 device.

If you have performed Exercise 4, you will know that the only thing that is preventing us to create a full UWP app is the requirement of exporting the boarding pass directly on the user’s desktop, since it’s one of the folders which we can’t directly access.

In this exercise, we’re going to remove this limitation by using a class offered by the Universal Windows Platform called **FileSavePicker**, which grants to the user the option to save a file on any location of the device. After this change, the boarding pass won’t be automatically exported anymore on the user’s desktop, but it will be up to the user to choose where he wants to export it.

The starting point of this exercise is the solution that you can find in the folder **C:\TR24\05-ReachAll\Start**. However, also in this case, if you’re following the exercises one after the other, you won’t need it: the starting point of this exercise is the application we have left at the end of Exercise 4.

## Task 1: Change the Export feature

The first task is to change the Export feature to leverage the **FileSavePicker** class instead of relying on a Win32 process. The first step is to add, in the header of the **MainPage.xaml.cs,** the proper namespace:

using Windows.Storage;

using Windows.Storage.Pickers;

Then we can completely replace the definition of the **OnExportBoardingPass()** event handler with the following code:

private async void OnExportBoardingPass(object sender, RoutedEventArgs e)

{

progressBar.IsActive = true;

var savePicker = new FileSavePicker();

savePicker.SuggestedStartLocation = PickerLocationId.Desktop;

// Dropdown of file types the user can save the file as

savePicker.FileTypeChoices.Add("Plain Text", new List<string>() { ".txt" });

// Default file name if the user does not type one in or select a file to replace

savePicker.SuggestedFileName = "BoardingPass";

var file = await savePicker.PickSaveFileAsync();

if (file != null)

{

//we simulate that the exporting operation takes a while to be completed

await Task.Delay(5000);

var builder = new StringBuilder();

builder.AppendLine("Boarding pass generated by FlightTracker");

builder.AppendLine("-----------------------------------------");

builder.AppendLine($"Flight code: {codeTextbox.Text}");

builder.AppendLine($"Flight date: {dateTextbox.Text}");

builder.AppendLine($"Departure city: {departureTextbox.Text}");

builder.AppendLine($"Arrival city: {arrivalTextbox.Text}");

builder.AppendLine("-----------------------------------------");

builder.AppendLine("Thank you for using FlightTracker");

await FileIO.WriteTextAsync(file, builder.ToString());

ShowNotification();

}

progressBar.IsActive = false;

}

The only missing namespace which we need to add in the header is the one required to create the string, which in the previous exercises was in the Win32 process:

using System.Text;

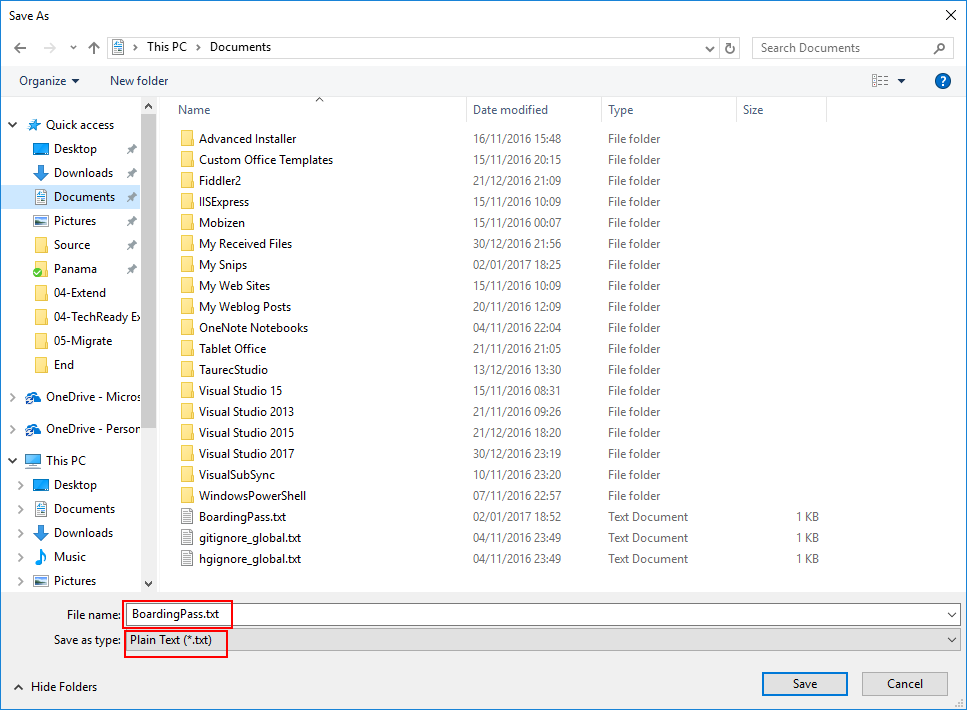
Using the **FileSavePicker** class is quite easy: after you have created a new instance, the only required property to configure is **FileTypeChoices**, which is a collection of all the options that the user has to create a file. In our case, we offer only the option to create a text file. As such, we add a unique element identified by:

* A label, which is **Plain Text**
* A set of supported extensions. In our case, we support only the **.txt** one.

However, the **FileSavePicker** class offers other properties to provide a better user experience. In our case, we’re leveraging:

* The **SuggestedStartLocation** property, to automatically open the picker at a specific location. In this case, since we want to recreate the experience of the original Windows Forms app, we’re using the **Desktop** value of the **PickerLocationId** enumerator, so that the picker will automatically point to the user’s desktop.
* The **SuggestedFilename** property, to give a default name to the file that will be created. In our scenario, it will be **BoardingPass.txt**, even if the user will have the chance to change it.

Once we have configured the picker, we can call the **PickSaveFileAsync()** method, which will simply open a window where the user can choose where to save the file, like in the following image:



Highlighted in a red box, you can see the options we have set when we have configured the **FileSavePicker** object. Once the user has pressed the **Save** button, we get in return a reference to the file, which is represented by the **StorageFile** class. The file, of course, will be empty, so we need to fill it with the boarding pass. The pass generation is made with the same code we used in the original Windows Forms app and it’s based on the **StringBuilder** class. The only difference is the way we write the content on the file selected by the user: we leverage a class of the Universal Windows Platform called **FileIO**, which offers a simple method to write a text into a file called **WriteTextAsync()**, which accept as parameter:

* A reference to the file where we want to save the content. In our case, it’s the **StorageFile** object that has been returned by the **FileSavePicker**.
* The text we want to save, which is the one generated using the **StringBuilder** class.

That’s all: the rest of the code is the same of the original Windows Forms app. We start displaying a progress indicator, we simulate a 5 seconds’ delay, we create the file with the boarding pass and then we stop the progress indicator.

## Task 2: Remove the dependency from the Desktop Bridge

Now that we have created a full Universal Windows Platform app and we have removed the dependency from the Win32 process, we can undo many of the changes we’ve made in Exercise 4, with the goal of removing all the dependencies from the Desktop Bridge. This way, we’ll get an application that will work exactly in the same way on any Windows 10 device.

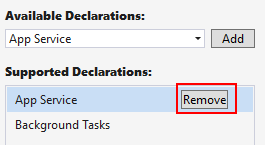
Here is an overview of all the changes we’re going to make

1. We’re going to remove the App Service, since we don’t need any more to handle the communication between the UWP app and the Win32 process.
2. We’re going to remove the project that contains the Win32 process.
3. We’re going to remove all the Desktop Bridge dependencies we have included in the manifest file.
4. We’re going to stop checking which is the device where the app is running: the Export feature will be available on every device now.
5. We’re going to remove the dependency from the **Windows Desktop Extensions for the UWP** library, since we don’t need any more to invoke the Win32 process.

As part of the exercise, try to do all of them on your own. However, in case you get stuck, you’ll find below the details on how to perform all these changes.

### Remove the App Service

The first step to remove the App Service is to delete the declaration in the manifest file. Double click on the **Package.appxmanifest** file, move to the **Declarations** section and click on the **Remove** button near the **App Service** declaration.



The removal of the App Service makes unnecessary many code blocks that we have added in Exercise 4:

1. In the **App.xaml.cs** file, we can remove the following properties:

public event EventHandler<string> StatusUpdated;

public static AppServiceConnection Connection = null;

private BackgroundTaskDeferral appServiceDeferral = null;

1. In the **App.xaml.cs** file, we can delete the:
   1. **OnBackgroundActivated()** method
   2. The **Connection\_RequestReceived()** event handler
2. In the **MainPage.xaml.cs** file, in the **MainPage()** constructor, we can remove the following line:

(Application.Current as App).StatusUpdated += MainPage\_StatusUpdated;

1. In the **MainPage.xaml.cs** file, in the first line of the **OnNavigatedTo()** method, we can remove the following line:

progressBar.IsActive = true;

1. In the **MainPage.xaml.cs** file, we can delete the **MainPage\_StatusUpdated()** event handler.

### Remove the Win32 process

We don’t need any longer the Win32 process, so we can safely delete the **DesktopBridge.FlightTracker.Export** project from the Visual Studio solution.

### Update the manifest file

In Exercise 4 we had to manually add some lines in the manifest file, since we leveraged some restricted capabilities that aren’t supported by the visual designer. As such, right click on the **Package.appxmanifest** file, choose **View code** and remove the below lines of code that are highlighted in yellow:

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

<Package xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10" xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest" xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10" xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities"

xmlns:desktop="http://schemas.microsoft.com/appx/manifest/desktop/windows10" IgnorableNamespaces="uap mp rescap desktop">

<Identity Name="2a72e6a6-5c14-4f98-a762-3511c6125f80" Publisher="CN=theappvengers" Version="1.0.0.0" />

<mp:PhoneIdentity PhoneProductId="2a72e6a6-5c14-4f98-a762-3511c6125f80" PhonePublisherId="00000000-0000-0000-0000-000000000000" />

<Properties>

<DisplayName>DesktopBridge.FlightTracker</DisplayName>

<PublisherDisplayName>The Appvengers</PublisherDisplayName>

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

</Properties>

<Dependencies>

<TargetDeviceFamily Name="Windows.Universal" MinVersion="10.0.0.0" MaxVersionTested="10.0.0.0" />

</Dependencies>

<Resources>

<Resource Language="x-generate" />

</Resources>

<Applications>

<Application Id="App" Executable="$targetnametoken$.exe" EntryPoint="DesktopBridge.FlightTracker.App">

<uap:VisualElements DisplayName="Flight Tracker" Square150x150Logo="Assets\Square150x150Logo.png" Square44x44Logo="Assets\Square44x44Logo.png" Description="Flight Tracker" BackgroundColor="transparent">

<uap:DefaultTile Wide310x150Logo="Assets\Wide310x150Logo.png" Square310x310Logo="Assets\LargeTile.png" Square71x71Logo="Assets\SmallTile.png">

</uap:DefaultTile>

<uap:SplashScreen Image="Assets\SplashScreen.png" BackgroundColor="#0078D7" />

</uap:VisualElements>

<Extensions>

<uap:Extension Category="windows.appService">

<uap:AppService Name="BoardingPassService" />

</uap:Extension>

<Extension Category="windows.backgroundTasks" EntryPoint="DesktopBridge.FlightTracker.Notification.ToastTask">

<BackgroundTasks>

<Task Type="systemEvent" />

</BackgroundTasks>

</Extension>

<desktop:Extension Category="windows.fullTrustProcess" Executable="DesktopBridge.FlightTracker.Export.exe" />

</Extensions>

</Application>

</Applications>

<Capabilities>

<Capability Name="internetClient" />

<rescap:Capability Name="runFullTrust" />

</Capabilities>

</Package>

This is one of the key changes on the project: since we have removed the **runFullTrust** capability, now we don’t need any more a special permission from Microsoft to publish our application on the Store.

### Make the export feature available on every device

The original definition of the Export button in the **MainPage.xaml** file was the following one:

<Button Content="Export" Click="OnExportBoardingPass" Margin="5, 0, 0, 0"

Visibility="Collapsed"

x:Name="exportButton" />

The reason was that, since we were leveraging a Win32 process to generate the boarding pass, it was available only on the desktop and, consequently, the button was hidden on every other platform.

Now we can safely remove the line highlighted in yellow, since we want the button to be always visible. The other change we need to make is in code. The **OnNavigatedTo()** of the **MainPage.xaml.cs** file contains the following block of code:

//we check if the app is running on the desktop: only if that's the case, we leverage the Desktop Bridge specific features

if (AnalyticsInfo.VersionInfo.DeviceFamily == "Windows.Desktop")

{

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

await FullTrustProcessLauncher.LaunchFullTrustProcessForCurrentAppAsync();

//we display the export button if the app is running on the desktop, since it's the only Windows 10 platform which supports running Win32 applications

exportButton.Visibility = Visibility.Visible;

}

We need to remove it since now the export feature is supported by every device. As such, we don’t need any more to launch the Win32 process when the app starts and we don’t need any more to make the export button visible only if the app is running on the desktop.

### Remove the dependency from the Windows Desktop Extensions library

Since now we don’t have any more to invoke the following line of code:

await FullTrustProcessLauncher.LaunchFullTrustProcessForCurrentAppAsync();

we can safely remove the reference to the Windows Desktop Extensions library we have added in Exercise 4. Expand the **References** section of the project, right click on the **Windows Desktop Extensions for the UWP** library and choose **Remove**.