

Simio API Note:

Creation: June2018 (Dhouck)

Revisions:

January 2022 (Dhouck) – Added to Github

November 2022 (Dhouck) – Updated to Simio 15.245.28983

November 2022 (Dhouck) – Added RunScheduleTest2

March 2025 (Glang) – Migrated to .NET Core

Contents

Simio API Note:	1
Overview	2
RunSimioSchedule	2
Architecture	3
Configuration	4
RunSimioSchedule2	5
Architecture	5
Configuration	6
Some Background Information on the Simio Engine	7
Building A Headless Executable	8
Creating a Service from the Executable	12

Overview

This API Note describes two common Simio Engine (SimEngine) API configurations. Both use unattended or “headless” operation of the SimEngine to achieve results, and both use “file-drop” techniques as the action that begins the processing.

These are both .NET Core console applications that can be run interactively or installed as a Windows Service. In both cases, the employed SimEngine files are referenced directly from where they were installed for a Simio Desktop application under the standard `c:\program files\Simio LLC\Simio` folder. You must alter the project references if your installation or DLLs are elsewhere.

This note describes some complex programming topics. It assumes that the reader is familiar with C# and .NET technologies such as locking and DLL dependencies.

RunSimioSchedule

The first scenario is in the RunSimioSchedule project. Here the same Simio project file is used. This project runs schedules that depend upon an external file that indicates outages. When the outage file changes (via file-drop) the project is run again to produce new results.

1. Load a Simio Project Schedule in a “headless” or unattended mode.
2. Set up folders to received data for a Schedule simulation and automatically run the Plan with new downtime data.
3. Export the results from the schedule

This project also demonstrates a configuration where a model is set up to be run either interactively or as a Windows service. Note that Program inherits from ServiceBase, and in the code’s Main method it checks against `Environment.UserInteractive` to determine how to Run.

This configuration has a Simio Project file (name defined in settings) being loaded at the very beginning and then uses a file containing downtime information as the triggering mechanism for the running of a Plan. A popular variation of this is to use the Simio project file itself as the trigger (and then of course the new project must be loaded). This variation is discussed below in Project RunSimioSchedule2.

Architecture

The diagram below illustrates the overall architecture of this API example:

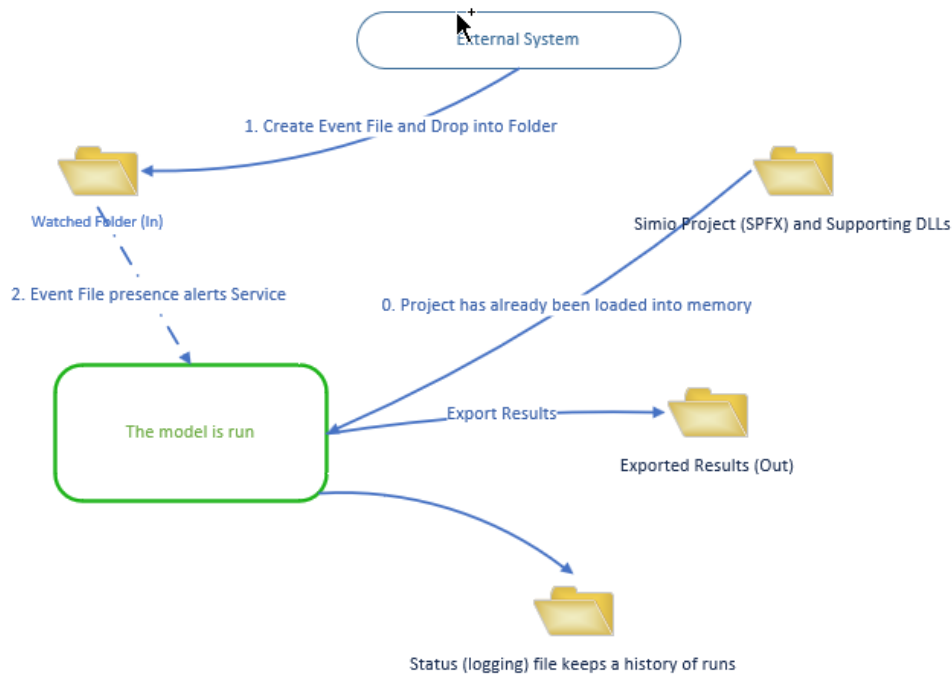


Figure 1- RunSimioSchedule Architecture

RunSimioSchedule is a windows process that runs as a service and waits for an event file to be dropped into a specified folder. The event file indicates that a model server has an exception, such as downtime being scheduled.

As the program begins the method Main is called which creates a RunContext which will exist throughout the life of the service. During the RunContext constructor, the following occurs:

- The API SimioProjectFactory.SetExtensionsPath is called to indicate where all the Simio Extension DLLs can be found when referenced.
- Some setup occurs, such as making sure of the existence of the folders (e.g. c:\temp\RunSimioSchedule\In, which is the folder for the Event files.
- The Simio project file (e.g. SchedulingDiscretePartProductionWithOnEndingExport.spfx) is located and loaded. It resides in memory for the duration of the run so project re-loads are unnecessary.

- Calls OnStart which creates a system FileWatcher which monitors the Events folder and has events for whenever an event file is created or changed within the folder. It also starts a timer that polls for the event file in case the FileWatcher ever misses a file.

Then the program goes into an infinite loop, leaving the Event file event checking to do all the work.

Whenever an Event file is found, the method CheckAndRun is called, which subsequently calls the method RunScheduleExportResultsAndSaveProject, which is the central logic to this example.

RunScheduleExportResultsAndSaveProject does the following.

- Converts the Event File - which is a CSV (Comma Separated Value) file to a DataTable and then deletes the Event File.
- References the model from the already loaded Simio project.
- If the DataTable exists, then it is assumed to be downtime data and is Imported into the model's Resource Table by the method ImportDowntime. If the model doesn't have a table called "Resources", then the data is simply ignored.
- If configured, the Model is saved prior to the run.
- If configured, the model's Plan is run, which now incorporates the downtime data.
- If configured, risk analysis is run on the model.
- If configured, the plan's schedule is exported.
- If configured, the model is saved

Configuration

The folder for configuration is found under Source > RunSimioSchedule2 Configuration.

The folder found there should be placed under the root folder, as defined in the project settings (e.g. c:\temp)

Start the executable and then drop a file found directly under the root into the In folder.

RunSimioSchedule2

The second scenario is in the RunSimioSchedule2 project. Here a popular scheme is employed where each time a Simio project file (*.spfx) is dropped into the In folder, the program will load the project and perform the actions specified in the settings file. RunSimioSchedule2 has been designed to run experiments as well as schedules.

Architecture

Many of the settings and particulars are like RunSimioSchedule, but the flow is slightly different:

1. An external system drops an .SPFX file into the In folder.
2. The SystemFileWatcher kicks off RunSimioSchedule2's CheckAndRun method.
3. CheckAndRun validates the file and calls RunScheduleAndSaveProject to take actions according to the Settings.
4. If errors are found, the project file is moved to Errors
5. If successful, then the project file is moved to Success

The architecture is represented here:

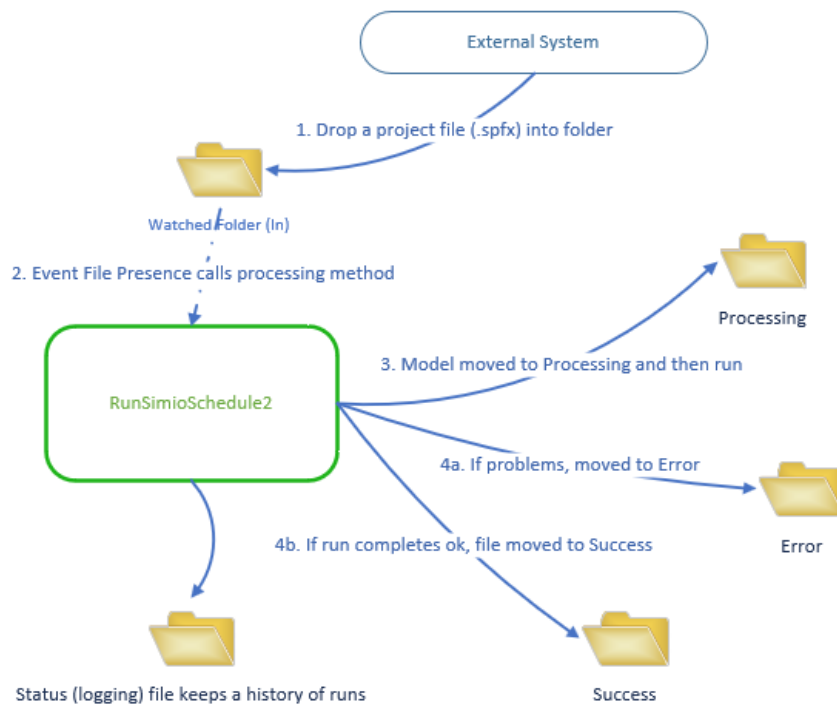


Figure 2 - RunSimioSchedule2 Architecture

Configuration

The folder for configuration is found under Source > RunSimioSchedule2 > Configuration.

The folder found there should be placed under the root folder, as defined in the project settings (e.g. c:\temp)

Start the executable and then drop a file found directly under the root into the In folder.

Some Background Information on the Simio Engine

The Simio Engine (or SimEngine or simply Engine) is the logic that implements the simulation and planning logic of Simio. When used as a desktop application, it is generally paired with a UI, but in the “headless” mode it isn’t. Only the DLL containing the engine (SimioDLL.DLL) along with support files for API and some file operations will be used.

When running within the UI, specific folders are searched by Simio when looking for DLLs, such as User Extensions and other add-ins.

When running headless this is not done, and instead all DLLs that are used must be in the location specified by the call `SetExtensionsPath` of `SimioProjectFactory`.

This can be a tricky problem, as often DLLs depend on other DLLs, which depend on even more and so on. There are a few free tools that might help you solve this puzzle:

One is DotPeek, which is made available for free by JetBrains. With this tool you can inspect a DLL for its dependencies.

Another is Process Explorer, which part of Microsoft’s SysInternal toolset. When run in Administrator mode this tool permits you to examine a running process (such as Simio) and determine what DLLs are loaded.

In this document are included the instruction for loading your headless executable as a Windows Service..

Building A Headless Executable

The overall structure of the program is simple: a path to the extensions is set, and then a series of actions on the model, such as running the model plan or running risk analysis are issued.

The difficulty most users have – as was mentioned before – is determining the correct DLLs to include. This will be covered later in a separate section.

Separate but closely allied with the calling of the model is the need for a mechanism to initiate the model. This example employs a file-drop mechanism. When a file is dropped in a special folder, a method is triggered by the System FileWatcher. This method (called CheckAndRun) runs the method RunScheduleExportResultsAndSaveProject, which does most of the work.

Note: because of a well-known FileWatcher deficiency (under rare circumstances it won't detect a file event) there is also a timer that checks every so often for a file and runs the same model processing method.

There is also a file that logs information about the running of the model. This is specified in the configuration file.

The utility methods (such as those that are used to read and write information to the model are included in a utility class called HeadlessHelpers.

Configuration settings are included as Application Settings, which – during the build of this application are output into a configuration file (RunSimioSchedule.exe.config) that is placed in the same folder as the executable (e.g. RunSimioSchedule.exe)

Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application retrieves them the next time it runs. [Learn more about application settings...](#)

Name	Type	Scope	Value
EventFile	string	User	Event.csv
StatusFile	string	User	RunStatus.txt
SaveProject	bool	User	True
ClearStatusBeforeEachRun	bool	User	False
DeleteStatusBeforeEachRun	bool	User	False
SimioProjectFile	string	User	SchedulingDiscretePartProductionWithOnRunEndingExport.spfx
ExportScheduleFile	string	User	ExportSchedule.xml
ModelName	string	User	Model
RunPlan	bool	User	True
RunRiskAnalysis	bool	User	False
ExportSchedule	bool	User	False
RootFolder	string	User	c:\temp\RunSimioScheduleTest
*			

When the program is built from Visual Studio the results are placed (as always) in a folder such as Source > RunSimioSchedule > bin > release, such as:

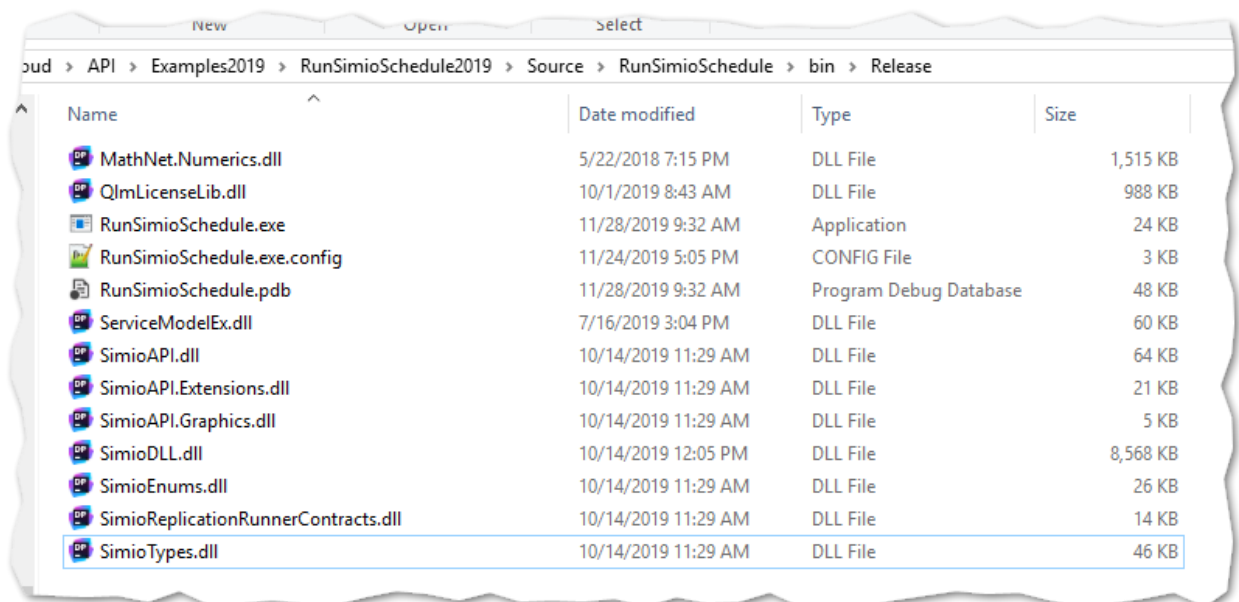


Figure 3 - Assemblies From Visual Studio - Very Likely Incomplete!

Now, you might think that all the DLLs you need would be found here. But this is likely wrong, since these are only the assemblies that Visual Studio could find through its process of compiling and linking and explicit references. The RunSchedulePlan program is going to – at run-time - launch a Simio Model, which may include many other DLLs (such as licensing files, user extensions, etc.)

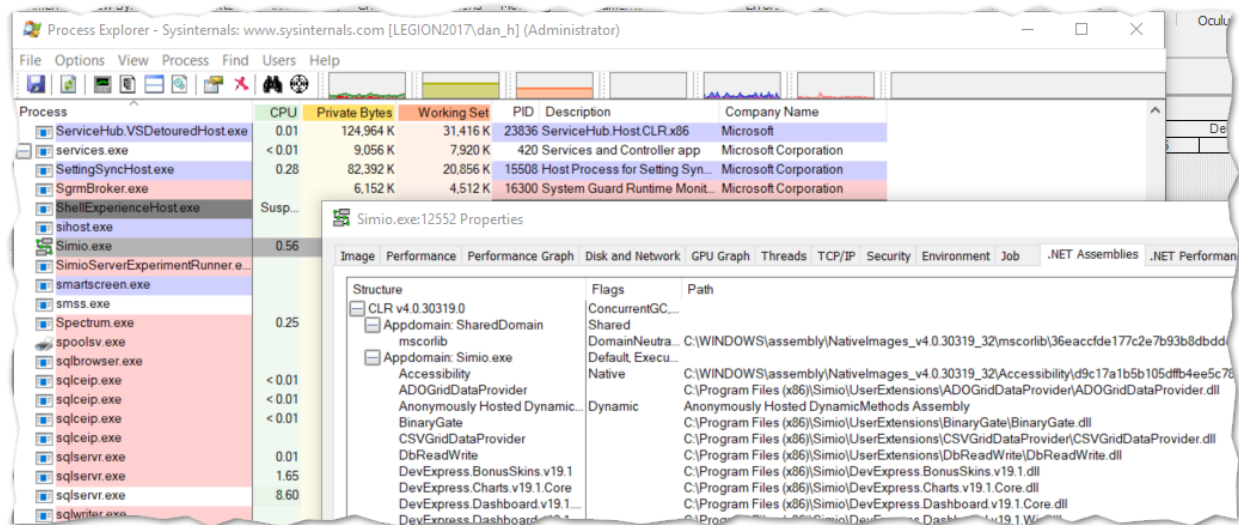
So, what to?

The easiest path is to have a desktop version of Simio loaded on the same machine and then have the program reference the needed DLLs that Simio installs at Program Files > Simio LLC > Simio.

Alternatively, you could explicitly locate the DLLs you need and place them with the EXE. If this is the path you wish, then continue reading...

A good starting point to locate the actual DLLs is to run Simio with your model and then run Microsoft's Process Explorer (as Administrator) to examine what DLL's are being used.

After starting Simio desktop and doing a RunPlan, Process Explorer is started. Look for Simio.exe and double-click to bring up its DLL view:



Looking through this we can see a lot of DevExpress DLLs, but many are DevExpress UI DLLs. There is a reference to DevExpress.Office.v19.1.Core, which is often included when Excel operations are employed.

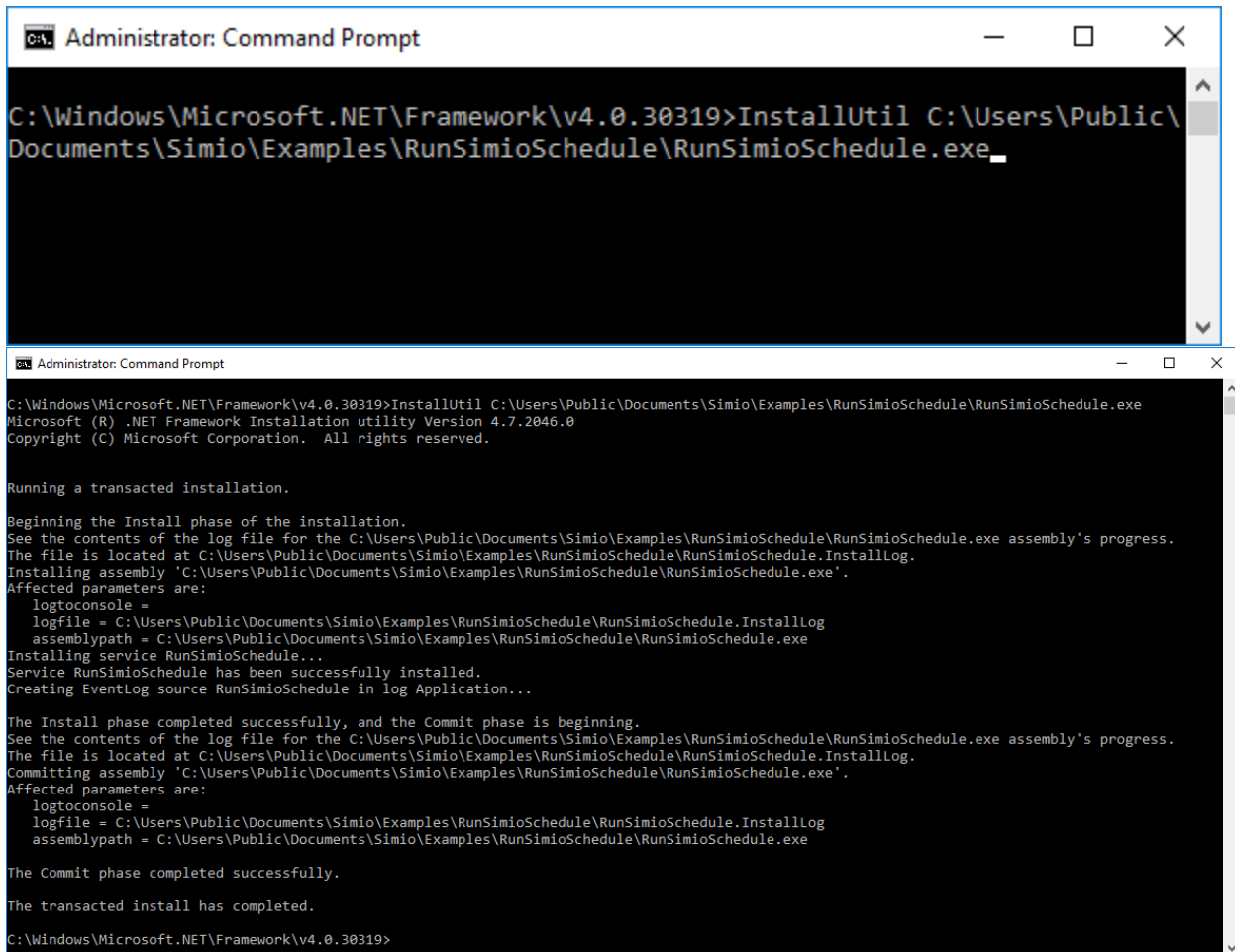
Of great interest to us are the ones within \SimioUserExtensions:

- ExcelGridDataProvider
- ExcelReadWrite
- GoodSelectionProcedure
- SimioRelocateObject

- SelectBestScenario
- SimioReplenishmentPolicies
- SimioSelectionRules
- SimioTravelSteeringBehaviors
- (no) SourceServerSink
- (no) WonderwareMES
- XMLGridDataProvider

Creating a Service from the Executable

- 1) Open a command prompt as Administrator
- 2) Navigate to your .NET installation folder to give access to utilities. For example:
"C:\Windows\Microsoft.NET\Framework\v4.0.30319> "
- 3) Enter "InstallUtil followed by the path to your executable. For example:
C:\Github\RunSimioSchedule\bin\Release\ RunSimioSchedule.exe". This will install the RunSimioSchedule as a windows service.



```
C:\Windows\Microsoft.NET\Framework\v4.0.30319>InstallUtil C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe

Microsoft (R) .NET Framework Installation utility Version 4.7.2046.0
Copyright (C) Microsoft Corporation. All rights reserved.

Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe assembly's progress.
The file is located at C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.Installlog.
Installing assembly 'C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe'.
Affected parameters are:
  logtoconsole =
  logfile = C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.Installlog
  assemblypath = C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe
Installing service RunSimioSchedule...
Service RunSimioSchedule has been successfully installed.
Creating EventLog source RunSimioSchedule in log Application...

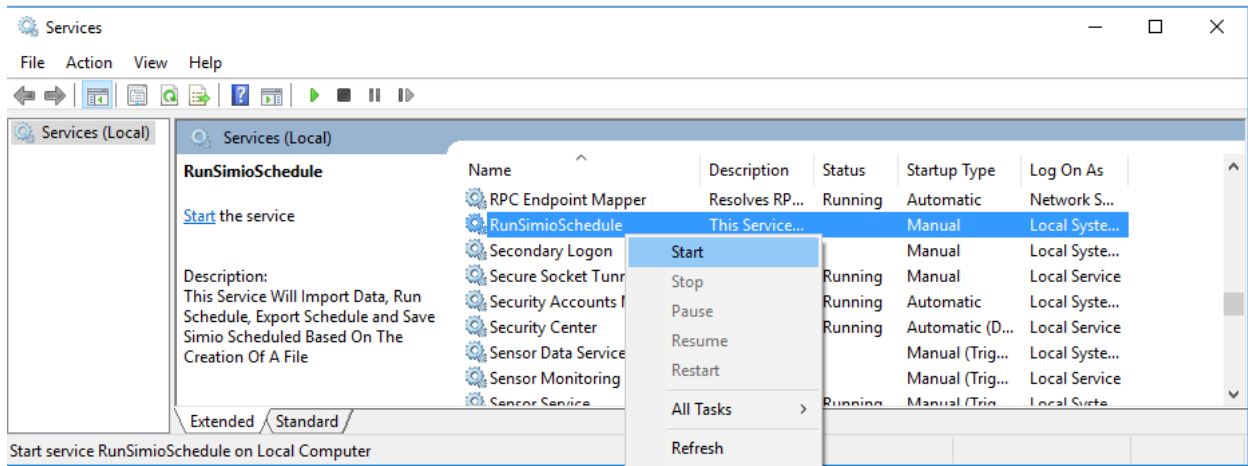
The Install phase completed successfully, and the Commit phase is beginning.
See the contents of the log file for the C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe assembly's progress.
The file is located at C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.Installlog.
Committing assembly 'C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe'.
Affected parameters are:
  logtoconsole =
  logfile = C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.Installlog
  assemblypath = C:\Users\Public\Documents\Simio\Examples\RunSimioSchedule\RunSimioSchedule.exe

The Commit phase completed successfully.

The transacted install has completed.

C:\Windows\Microsoft.NET\Framework\v4.0.30319>
```

- 4) Next, start the service from Services.



And then the service should be running, and you can test by dropping a file in your “In” sub-folder.