# Simio API Note: Simio API Helper

Last Update: 3 March 2021

Contents

[Simio API Note: Simio API Helper 1](#_Toc65819407)

[Overview 2](#_Toc65819408)

[Using the API Helper. 3](#_Toc65819409)

[Tab DLL Helper 4](#_Toc65819410)

[Tab .Net Versions 6](#_Toc65819411)

[Tab: Find User Extensions 7](#_Toc65819412)

[Simio Automated SimEngine Workflow and Recommendations 8](#_Toc65819413)

[The Code for Automating SimEngine 11](#_Toc65819414)

[Tabs: SimEngine Harvester with direct SimEngine Run 12](#_Toc65819415)

[Tabs: Logs and Settings 13](#_Toc65819416)

[Running Simio API Helper 14](#_Toc65819417)

[Using the Simio API Helper Code 15](#_Toc65819418)

[Appendix: SimEngine Debugging 20](#_Toc65819419)

[Appendix – Simio Licensing (Server and Node-Locked) 24](#_Toc65819420)

[Server Licensing 24](#_Toc65819421)

[RLM Licensing 24](#_Toc65819422)

[Appendix – API Updates 28](#_Toc65819423)

[Appendix: Web API References 29](#_Toc65819424)

# SimioApiHelper GitHub Repository

The SimioApiHelper GitHub contains documents, utilities, and samples that can assist a developer in understanding how to automate the Simio Engine.

This repository includes several sample projects as well as a utility to help collect the files needed for automating the Simio Engine.

There are two ways to think about how to use this information:

1. If you are a developer, you can consider these samples as “guides” that you can take and modify as your own. There are several “helper” classes that should assist with this.
2. If you simply want to automate you can install and employ the ApiHelper to help you collect the DLLs from the version of Simio that you want to use, and then select the SimioEngineController and an Interface (such as SimioEngineInterfaceFileDrop) and start automating.

Note: this is not a supported product with full support and regular updates. It is a use-and-abuse collection of free software with all source included so you can understand the mechanisms and get you started with SimEngine automation.

## About Simio’s use of GitHub

This code and documentation are intended to be distributed via a GitHub repository using Simio’s GitHub conventions. The reasons for using GitHub include:

1. Employing a standard mechanism for distributing API notes
2. Allowing the API information to evolve
3. Allowing the Simio community to engage in the production of Simio API examples

If you have suggestions for improving this example you can either:

* Send your ideas to Simio Support ([support@Simio.com](mailto:support@Simio.com)). Make sure you reference which GitHub repository you are commenting on.
* Ask for permission to contribute directly to this GitHub repository.

The structure of the folders are this:

At the top level are usually these folders:

Models – Simio models (e.g. .SPFX files) containing models used by the site

Documentation – Documentation and support such as drawings

Executables – Fully built exe and DLL files (and sometimes an install kit or a test configuration.

Source – under this is source code. Sometimes the projects are simply at the top level.

Data – where data files are kept if needed

Examples – Examples of files, folder structures, etc.

# QuickStart – A FileDrop SimioEngine Example

Sometimes is it just good to get something working, so this section describes how you can get a SimEngine working with the filedrop interface.

Prerequisites:

To get this to work you will need a licensed version of Simio.

Here are the steps:

1. Create a folder structure that contains the needed files. We will use c:\TestSimioEngine as our root folder. If you want, you can simply copy the example from the Examples folder.
2. Under the root folder we’ll have:
   1. RequestFileDrop – Where our request files will be dropped
   2. Models – where models that will be used by the Requests have been stored.
   3. Extensions – Where the Simio extensions will be placed. You can use the SimioApiHelper to harvest your Simio DLL files here, or simply use the version that is already in the Examples folder.
   4. Logs – Where logging is stored for history and diagnostics
   5. Success – Where files are stored upon SimEngine run success
   6. Failure – Where files are stored upon SimEngine run failure
   7. Executables – This holds the SimioEngineController.dll and its supporting files. It also holds the SimioEngineInterfaceFileDrop.exe that calls the controller.
   8. RequestSamples – holds request files that can be copied and pasted into the RequestFileDrop folder.

So that’s it for the setup.

Now start a Command Prompt and navigate to the Executables folder mentioned above. Run SimioEngineInterfaceFiledrop.bat. This batch file simply contains a command to run the SimioEngineInterfaceFileDrop.exe executable file. It displays a banner showing what the arguments are and then waits for something to appear in the RequestFiledrop folder.

So, while this program is running, use FileExplorer to copy a file from the RequestSamples and drop it into the RequestFileDrop folder. With a second you will see the FileDrop program indicate that it has found the request and is processing it. When it is done it will place information into Logs, Success (or Failure) and (if indicated) will save he Model.

# SimioApiHelper Utility

## Overview

This section describes a utility that can:

1. Examine/test the DLLs that are used by the APIs.
2. Construct sample automated SimEngine folders
3. Test the operation of the SimEngine methods.

The organization of the build file is described in the section “Using the Simio API Helper Code”

## Using the API Helper.

The Simio API Tester is not part of the Simio product. Rather, it is a demonstration and test tool that was built to help debug problems related to using the Simio API, as well as to demonstrate how to use the Simio API.

How much benefit you derive will depend upon your programming knowledge, but anyone attempting to use the API should find it useful.

You can use this tool to verify that your DLLs are:

1. In the right place
2. Accessible
3. Implementing the correct Interfaces
4. Referencing other Assemblies correctly

And in the current version you can also use it to:

1. Prepare your environment for running Simio in a bare SimEngine (without the UI) mode.
2. Run a Simio model within SimEngine to
   1. Run experiments, and/or
   2. Run a Plan.

Note: You can only run a Plan if you have the Simio RPS edition (or the Personal Edition with a small demonstration model).

The UI architecture is a simple WinForms tabs, and the sections below are organized according to those tabs.

## Tab DLL Helper

This tab is used to examine and load DLL assemblies.

The top drop-down displays the locations where Simio DLLs can be found, and the next drop-down then shows the DLL files within that location. The Exclude filter can be used to reduce the number of DLL files displayed.

Once a file is chosen, general information about the contents of the DLL is shown, along with the definitions found within the file. If you wish to see only Simio information, check the “Simio Only” checkbox.

A screenshot of a computer

Description automatically generated

Using the “Load Assembly” you can get information about the Interfaces as well as DLL dependencies.

Using the “Add Dependency to Harvest” button, you can make sure that the DLLs upon which this DLL is dependent are in the target harvest folder.

As an example, let’s use this tab to find the dependencies for the Simio Engine (SimioDLL.DLL).

Select the Location c:\Program Files (x86)\Simio

And then select the DLL file SimioDLL.dll

Next press “Load Assembly” and then “Add Dependencies to Harvest”.

Scrolling to the bottom of the text to the right of the “Load Assembly” reveals the following dependences for Simio build 205. So, at a minimum, these are the dependent DLLs required to run a model directly in SimEngine. There may be (are likely are) others, which would be required by User Extensions.

\*\*\*\*\*\*\*\*\*\* Referenced Assemblies:

There are 9 unique dependencies:

0 SimioAPI.Extensions Full=SimioAPI.Extensions, Version=11.205.20185.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\SimioAPI.Extensions.dll

1 SimioAPI Full=SimioAPI, Version=11.205.20185.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\SimioAPI.dll

2 SimioAPI.Graphics Full=SimioAPI.Graphics, Version=11.205.20185.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\SimioAPI.Graphics.dll

3 SimioEnums Full=SimioEnums, Version=11.205.20185.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\SimioEnums.dll

4 SimioTypes Full=SimioTypes, Version=11.205.20185.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\SimioTypes.dll

5 SimioReplicationRunnerContracts Full=SimioReplicationRunnerContracts, Version=11.205.20185.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\SimioReplicationRunnerContracts.dll

6 MathNet.Numerics Full=MathNet.Numerics, Version=4.5.1.0, Culture=neutral, PublicKeyToken=null

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\MathNet.Numerics.dll

7 ServiceModelEx Full=ServiceModelEx, Version=1.0.0.0, Culture=neutral, PublicKeyToken=d0f8277329bba7a0

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\ServiceModelEx.dll

8 QlmLicenseLib Full=QlmLicenseLib, Version=12.1.20063.7, Culture=neutral, PublicKeyToken=9c65e32c060523a1

Path=C:\Program Files (x86)\Simio LLC\Simio API Helper\QlmLicenseLib.dll

## Tab .Net Versions

This tab shows general information about the computer that it is run on, as well as the .NET versions that are installed.

A screenshot of a social media post

Description automatically generated

## Tab: Find User Extensions

Given a starting point (which is generally the default SimioUserExtensions folder beneath a user’s Documents folder), this tab is used to locate all the assemblies (DLLs) containing Simio interfaces.

All the Simio interfaces within each DLL are listed. For example, in the listing below, there is a DLL called ExcelReadWriteEPPLus.dll that contains interfaces such as IStepDefinition and IStep, which identifies it as a User Step extension.

A screenshot of a cell phone

Description automatically generated

## Simio Automated SimEngine Workflow and Recommendations

When you are running Simio from the desktop version you are running the Simio “Engine” with the standard bolted-on Simio UI. This is the most popular and recommended configuration for designing, building, and interacting with your model. However, there are several use cases for running your model without the Simio UI, and this is where the automated SimEngine workflow configuration comes in.

As Simio is designed to be data driven, a common scenario is to have bare SimEngine application awaiting the arrival of new data, and then processing the Simio project (to run experiments or schedule a plan) and then storing the resulting data. Simio has a range of APIs to assist this with this direct SimEngine mode.

Achieving success when building a SimEngine application often depends upon selecting and using the correct components. These binary components are called “assemblies” and consist of libraries (files with DLL extensions and/or executables, which are files with EXE extensions.

The task can appear daunting because Simio is very modularized and determining which DLLs to use can be a confusing task. It is further challenging because the executable that you build will reference Simio API DLLs to load the Simio Engine, and then the engine will - in turn -require more DLLs dependent upon other requirements, such as the type of User Extensions that you are using.

This section will provide you with some background knowledge, recommendations, and debugging techniques to help sort this out.

A close up of a map

Description automatically generated

Let’s start with a recommendation: when you wish to have a SimEngine configuration, it is wise to collect or “harvest” all the Simio components that you need into a dedicated folder, such as c:/test/SimEngineTest

In theory you *could* use the Simio installation folder, but this would mean that every time you update Simio you will have to re-test your SimEngine application. And perhaps more importantly, when the Simio Engine is running it looks for DLLs (such as custom user extensions) in many places. ***When you are using SimEngine directly, this searching does not occur!*** So, if you unwisely decided to do this, you would have to move all the DLLs that your application uses into the Simio installation folder, causing unnecessary clutter, duplication, and confusion.

Note that the Simio API Helper has a utility that can help you harvest the most likely needed DLLs from your Simio installation and put copies in a folder of your choice. This function is found under the tab named “SimEngine DLL Harvester”.

This has the advantage of protecting your application (which are often production oriented) from updates or upgrades to the Simio software, and you have all your dependent DLLs in a convenient package for backing up. The downside of course is that when you do update Simio to a new version you must decide if you want your SimEngine application to use the new update, and then re-harvest (and re-test) your SimEngine system.

When you are building (and testing) an automated SimEngine application (e.g. with Visual Studio) your executable will generally “look” for its supporting DLLs that are in the same folder as the executable.

The diagram below illustrates this workflow.

A close up of a map

Description automatically generated

### The Code for Automating SimEngine

The actual code is conceptually simple and generally of the form

1. Look for a reason to run, such as new data arriving, or a new Simio project file.
2. Set the extensions folder (which instructs the code where to look for the DLLs)
3. Load the Model (and check for errors)
4. Run the Experiment or Plan for the model.
5. Optionally Save the results

BTW: The most common error when using the direct SimEngine mode is to omit Step 2, which instructs the Simio Engine where to find its dependent DLLs

## Tabs: SimEngine Harvester with direct SimEngine Run

These tabs can help you set up, test, and run your SimEngine application.

The Harvester tab permits you to take (harvest) all the files that your SimEngine needs from where an installed Simio folder, and places all the non-UI files into a folder of your choice.

Once you have chosen the Simio installation location and the target location it will show you (in the checkbox list) all the files that will be moved. You can selectively uncheck files that you are sure you do not need, or just leave the recommended files checked; in general, extra files will not hurt.

Pressing the “Start Harvest” will move all the checked files to your target location.

A screenshot of a social media post

Description automatically generated

The direct SimEngine Run provides a few options.

## Tabs: Logs and Settings

There are two utility tabs to assist with running and debugging.

The tab Logs shows messages that the SimioApiHelper generates. These are made by a utility class called Loggerton, which generates in-memory logging. Prudent logging will help you debug your program.

A screenshot of a social media post

Description automatically generated

The tab Settings allows the program to remember your configuration, so you don’t have to reenter each time you run. For example, the folder where Simio is installed is stored here, as well as the folder that you choose as the target for harvesting the assemblies.

A screenshot of a social media post

Description automatically generated

# Running Simio API Helper

The files built from SimioApiHelper are manually moved to the top-level Executables folder. There you will find the SimioApiHelper executable (SimioApiHelper.exe) as well as all its dependent assemblies (\*.DLL) and its configuration file.

You can start is by double clicking on SImioApiHelper.exe.

When starting, you may see errors like this:

A screenshot of a social media post

Description automatically generated

This is from the harvesting refresh, and it means that there were DLLs with the same name that exist in multiple folders of the Simio installation. It is mostly harmless, but you should look at the Logs tab and investigate why there were duplicates.

# Using the Simio API Helper Code

Although you can simply run the Simio API Helper by going to the Executables file and double clicking on SimioApiHelper.exe, you can also run and debug the code yourself using Visual Studio. A familiarity with .NET, Visual Studio, and the C# language is assumed.

After cloning the repository from GitHub, the entire solution can be loaded in Visual Studio (version 2017 or 2019) via the top-level Visual Studio solution file SimioApiHelper.sln.

A screenshot of a cell phone

Description automatically generated

The solution file contains several example projects, including the SimioApiHelper.

A screenshot of a computer screen

Description automatically generated

You can switch from one project to another by select a project, right-clicking and choosing “Set as Startup Project”. Once done, that project will appear bolder than the others.

A screenshot of a cell phone

Description automatically generated

At this point, clicking on Start will cause that project to execute.

When you first try to run, you may get an error indicating that one of more of the Simio DLLs cannot be located.

You will need to locate these files in the folder where Simio is installed.

So first, delete the files from the projects References folder (you will get a warning dialog that the Selected items will be removed).

A screenshot of a computer screen

Description automatically generated

Next, right click on References and select “Add Reference…”, which brings up a dialog box.

At the lower right there is a button “Browse…” that will allow you to find your Simio DLLs:

A screenshot of a computer screen

Description automatically generated

You may have to do this same procedure for each project. It may seem a bit annoying, but it is a worthwhile exercise, and it is making sure that you are using the correct Simio libraries. Once done, you won’t have to repeat the exercise.

# Appendix: SimEngine Debugging

One of the hardest things to determine is what DLLs are required, and/or what the dependencies between the DLLs is.

There are two free tools that can help with this:

1. Process Explorer from SysInternals (Microsoft) to examine DLL dependencies
2. DotnetPeek from JetBrains to examine assemblies (such as DLLs)

A screenshot of a computer

Description automatically generated

Process Explorer can be used to examine a running program. This is incredibly useful because we can see what DLLs are employed regardless of when they were loaded. It can be downloaded for free from:

<https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer>

A screenshot of a computer

Description automatically generated

So, in the example above the included DLLs are shown.

Below is Process Explorer being applied against Simio with the same model being run.

A screenshot of a computer

Description automatically generated

And below is the result of one of the example projects “SimEngineFormsExperiment” which uses the call “System.Appdomain.CurrentDomain.BaseDirectory” to pick up the location of the SimEngineFormsExperiment.exe to locate all of the DLLs.

# Appendix – Simio Licensing (Server and Node-Locked)

This appendix describes some of the issues surrounding Simio licensing. When running in bare SimEngine mode, Simio of course will still require a license, so it is important to make sure that your program is able to properly find its license. Machine based licensing (also called Node-Locked) is usually without problems, as the license information is simply on the machine. The difficulties usually arise with Server licensing, where the machine must locate the License Server on the network. The usual suspects of network location and firewalls then come into play.

## Server Licensing

There are currently two types of Service licenses in play, as Simio transitions from RLM licensing to QLM licensing. Each is discussed in turn.

### RLM Licensing

The best documentation about RLM Server licensing can be found in this document:

Here are some key points:

The licenses are stored on the License Server under ProgramData > Simio LLC > Simio Network Licensing.

If there is a file SimioConfig.lic, it should hold configuration information about the “random” port used by the license simulator.

The files rlm.dlog and simio.dlog hold debugging information.

The files that have the extension “.LIC” hold the licenses.

The Service on the License Server that runs licensing is named “RLM Simio”. It obviously must be running.

If it is stopped, the desktop Simio programs will raise a message box:

A screenshot of a cell phone

Description automatically generated

If the Server has no licenses, you get the message:

A screenshot of a social media post

Description automatically generated

For more information, please see the document “Simio Network License Server.pdf” that accompanies this document.

# Appendix – API Updates

This appendix will highlight and document significant changes to the API.

Version 12.209.21005

RunPlan() has been deprecated in favor of RunPlan(RunPlanOptions options).

Where options currently are

A screenshot of a cell phone

Description automatically generated

# Appendix: Web API References

RESTful API Design – Step by Step Guide by Tanmay Deshpande

<https://medium.com/better-programming/restful-api-design-step-by-step-guide-2f2c9f9fcdbf>

Summary

* Keep it simple
* Use Noun and not verbs (product/1234)
* Use the right HTTP method:
  + GET – to get a resource/collection
  + POST – To create a resource/collection
  + PUT/PATCH – To update the existing resource/collection
  + DELETE – To delete the existing resource/collection
* Use Parameter (/products?name=’ABC’)
* Use proper HTTP Codes
  + 200 OK — This is most used HTTP code to show that the operation performed is successful.
  + 201 CREATED — This can be used when you use the POST method to create a new resource.
  + 202 ACCEPTED — This can be used to acknowledge the request sent to the server.
  + 400 BAD REQUEST— This can be used when client-side input validation fails.
  + 401 UNAUTHORIZED / 403 FORBIDDEN— This can be used if the user or the system is not authorized to perform a certain operation.
  + 404 NOT FOUND— This can be used if you are looking for a certain resource and it is not available in the system.
  + 500 INTERNAL SERVER ERROR — This should never be thrown explicitly but might occur if the system fails.
  + 502 BAD GATEWAY — This can be used if the server received an invalid response from the upstream server
* Versioning (/v1/products)
* Use pagination (/products?limit=25&offset=50) Also keep defaults for limit and offset
* Use Supported Formats. Prefer json, but XML if necessary.
* Use proper error messages. Example from Facebook:

{  
"error": {  
"message": "(#803) Some of the aliases you requested do not exist: products",  
"type": "OAuthException",  
"code": 803,  
"fbtrace\_id": "FOXX2AhLh80"  
}

* Use OpenAPI specifications: <https://swagger.io/resources/open-api/>

The OpenAPI current has standard with statements such as:

“ Fields are case-sensitive. Keys are case-insensitive”

“APIs are defined either by YAML or JSON”

And also define Parameters, etc.