MetaAutomation Sample2

Installation Instructions

Contents

[Sample Overview 3](#_Toc469389870)

[Setup 3](#_Toc469389871)

[Installing the Integrated Development Environment (IDE) 3](#_Toc469389872)

[Getting the MetaAutomation Sample 2 Solution 3](#_Toc469389873)

[First Run of MetaAutomation Sample 2 3](#_Toc469389874)

[Selecting Other Checks in the Sample 4](#_Toc469389875)

[Enabling the Different Example Checks in the Map File 5](#_Toc469389876)

[Example check 1 5](#_Toc469389877)

[Example check 2 5](#_Toc469389878)

[Example check 3 5](#_Toc469389879)

[Example check 4 5](#_Toc469389880)

[Example check 5 5](#_Toc469389881)

[Example check 6 6](#_Toc469389882)

[Iterative Runs for Test and Debugging 6](#_Toc469389883)

[Outputs from a Successful Run 6](#_Toc469389884)

[The XML and the C# Source Code 6](#_Toc469389885)

[How To Use The XML Artifacts to Drive The Checks and Read Results 7](#_Toc469389886)

[A. Entries in the XML artifact that drive the check 7](#_Toc469389887)

[B. Entries that are determined by the check code and the engine during a check run 8](#_Toc469389888)

[Troubleshooting 9](#_Toc469389889)

# Sample Overview

The MetaAutomation Sample 2 is a Visual Studio solution developed with Microsoft Visual Studio Community 2015. There are 8 projects in the solution.

Microsoft Visual Studio Community 2015 is free to download, use and install from the Microsoft web site VisualStudio.com.

This document concerns setup and run of the sample as downloaded, including the 6 example checks included in the sample.

MetaAutomation Sample 2 distributes checks and check components across processes on the same machine, but it is not capable of distributing checks across machines. Please see MetaAutomation Sample 3 on the MetaAutomation web site for that capability, and the corresponding sample chapter in the book on MetaAutomation published in 2016.

Both MetaAutomation Sample 2 and MetaAutomation Sample 3 demonstrate checks with

* self-documenting, hierarchical check steps that show pass, fail, or blocked
* distribution across processes with synchronization
* pure, compact, descriptive check artifacts in a schema-defined grammar of XML

For more information and diagrams on how this system works, to customize the checks or adapt the infrastructure, and more on how this implements the Atomic Check pattern and works in the bigger setting of the MetaAutomation pattern language, please see the site <http://MetaAutomation.net> and the book on MetaAutomation (2nd Edition or later).

# Setup

## Installing the Integrated Development Environment (IDE)

If you don’t have a version of Visual Studio 2015, please install the latest free version:

1. Browse to this page <https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>
2. Select “Download Community 2015” or a newer version
3. Verify that it’s published by Microsoft, and Select “Run”
4. In the installer, choose the “Typical” installation
5. Proceed with the installation

## Getting the MetaAutomation Sample 2 Solution

The MetaAutomation Sample 2 is available as a zip file here:

<http://metaautomation.net>

Click the “Samples” tab, download MetaAutomation Sample 2 and un-compress it to a new folder in your workspace.

## First Run of MetaAutomation Sample 2

1. Give the current user read/write access to the directory of the MetaAutomation Sample 2 solution, and all folders, subfolders and files. Clear the read-only flags as well.
2. Open the solution in Visual Studio 2015
3. Check that the active solution configuration is “Debug”
4. Build the entire solution of 8 projects. NOTE due to the fact that the assembly that implements the checks is loaded at runtime, this step is not optional and may have to be repeated to ensure that CheckProcess.exe is correctly built.
5. Verify that the startup project is “CheckLauncher”
6. F5 to run the first example check of the sample, out of six checks.

CheckLauncher is a command-line application that will show status information on the check run. This is expected to succeed; if not, there is a configuration issue. See the Troubleshooting section below for some ideas of how to resolve issues.

The first example check of the sample is the only one that is enabled in the MetaAutomation Sample 2 as downloaded.

For the results of the check, see the file CheckMap.xml in the Artifacts directory in the file location where you are running the MetaAutomation Sample 2 solution. This file now points to the new artifact file in the file system. Open it in your favorite text / XML editor, and check it out.

This file will be used to run the same check, for the next run of that check.

The other checks listed in the CheckMap.xml file work the same way.

## Selecting Other Checks in the Sample

In Visual Studio, see the file directory “Artifacts” for the file “CheckMap.xml.” This file is the map that CheckLauncher uses to find the artifact files that are needed to tell the system how to run the check. Note that if a check runs to completion, a new artifact file for the check is created, and the map is updated to point to that file.

The artifact files called “CheckRunArtifact\_<GUID>.xml” do double-duty in this system; each file is a result of a check run, and is needed to run the check next time. This is how the check knows what check steps are expected as part of the check, and also how the check steps can get updated if necessary.

The map file determines which of the example checks are run. Each Check element in the XML corresponds to an artifact XML file which includes

* The results of the last check run
* Check steps information, for use with the next run of the same check
* Some information needed to run the check

In the file CheckMap.xml, as downloaded, only one Check element is not commented out in the distributed version of the project:

<Check>

<DataElement Name="DirectoryName" Value="Local\TwoTierFileSystemCheck" />

<DataElement Name="CurrentCheckRunArtifact" Value="CheckRunArtifact\_Original.xml" />

<DataElement Name="Comment" Value="Simple example of a two-tier check, in two different processes" />

</Check>

This instructs the engine to load the file called CheckRunArtifact\_Original.xml in the relative directory Local\TwoTierFileSystemCheck, and run a check starting with that CheckRunArtifact XML file. On successful completion of the check run (whether the check itself results in a PASS or a FAIL) the XML element here will be updated to point to the new artifact file.

## Enabling the Different Example Checks in the Map File

Each un-commented “Check” element in the map file is enabled, and will run a different example check. All the enabled example checks will run in parallel in different processes on the machine on launch of the CheckLauncher executable.

### Example check 1

This example check is enabled by default. To disable it, comment out the “Check” element in XML in the map file that represents this check.

This example check does a minimally complex demonstration of self-documenting, hierarchical check steps.

### Example check 2

Un-comment the “Check” element in the map file for Example\_2\_CheckOfWebPageBuiltToFailRandomly to enable this check.

This check requests a page of the MetaAutomation.net web site, but the page is deliberately built to fail 80% of the time. There are 5 possible behaviors of this page, based on the time of page request. Only one behavior passes the check, and the others fail, to demonstrate how a check can report the failure.

### Example check 3

Un-comment the “Check” element in the map file for Example\_3\_TwoTierCheckOfFileSystem to enable this check, but there is one more step needed: edit the artifact file that will guide the next run of this check to choose a file path and name for the temporary file that this check needs to test the file system. MetaAutomation Sample 2 runs all tiers of the checks on the same machine, so this file share could be, e.g., “C:\temp\TestFile.txt.”

This check demonstrates a minimal behavioral check that requires driving check steps in two different operating system processes.

### Example check 4

Un-comment the “Check” element in the map file for Example\_4\_DeeplyNestedCrossProcessCheck to enable this check.

This check demonstrates how a check can have an arbitrarily complex hierarchical structure of check steps across an arbitrarily complex set of operating system processes. In this example, MetaAutomation Sample 2, these processes all happen on just one operating system instance, however. MetaAutomation Sample 3 demonstrates (using a WCF service) that these processes can happen across any number of systems, given that each of these systems can run a simple XML-based service.

### Example check 5

Un-comment the “Check” element in the map file for Example\_5\_StaticFailureIndicatedInArtifact\_DebugBuildOnly to enable this check.

This check will only work with a DEBUG build of the MetaAutomation sample system. It demonstrates a hard-coded failure in a check to aid with debugging and demonstrate how failures are handled in multi-tiered checks.

### Example check 6

Un-comment the “Check” element in the map file for Example\_6\_RollingFailureIndicatedInArtifact\_DebugBuildOnly to enable this check.

This check will only work with a DEBUG build of the MetaAutomation sample system. It demonstrates rolling failures in a check to aid with debugging, and failure handing in a multi-tiered check.

## Iterative Runs for Test and Debugging

In the CheckLauncher project, class LaunchAsynchronousChecks, see the local variable

const int IterateCount = 1;

This determines how many times the checks referenced in the file CheckMap.xml are run. To start out, just leave this as 1.

### Outputs from a Successful Run

Look in the directory where you edited the file CheckRunArtifact\_Original.xml. There is now a new file called CheckRunArtifact\_<GUID>.xml. Open up this file to see the result of the run.

The file CheckMap.xml now points to this file, and doesn’t know about the CheckRunArtifact.Original.xml. To make changes to the check run, edit the new file; it is the new reference for running the check.

### The XML and the C# Source Code

Open the new artifact file, and in project CheckMethods, open class Checks.

In the artifact file, note that there are two DataElement elements with name “CheckMethodGuid.” Look in the C# code to find those two GUIDs, and the two C# methods that they represent.

The artifact XML determines where to find the executable to run the checks, what methods to run, the user name for named system semaphore security, and other things, as well as holds all the information from the prior check run including the steps. The element SubCheckData determines how subcheck(s) are called. There are many examples of this with the sample that you already downloaded; see the check map in file CheckMap.xml.

The artifact XML also enables fault injection (causing selected steps to fail), for Debug or LocalDebug solution builds only. See the examples in the check map described with comments “Examples with injected failures for debugging” in CheckMap.XML.

The C# methods follow a pattern of the xUnit testing tools. The code determines what is done at the code level, including method and step names. From here, the SDET would load, exercise, and measure output from the product, all in self-documenting check steps.

To see how self-documenting steps reflect themselves in check results, compare the artifact with the C# code step names. This gets more complex with the more elaborate example checks.

### How To Use The XML Artifacts to Drive The Checks and Read Results

Here is the form of the XML artifact from example check 3 from MetaAutomation Sample 2, which shows a basic file system check spanning three processes, one for launching the check, one for creating and writing a file, and one for reading and verifying the file. The text in various shades of red and blue are as seen in the Visual Studio text editor, with default colorization settings, and the green annotations are added to this graphic to correspond to the notes below:



### A. Entries in the XML artifact that drive the check

For this sample implementation, these values are set by editing the XML with a text editor, if they need to be set or changed.

For an adapted or more complete implementation of the patterns of MetaAutomation, these values might be set by the automation infrastructure outside of the scope of the actual check runs.

#### A1. PathAndFileToRunner

A relative path is used here, and is determined by the name of the EXE used to establish the new process for running a check or subcheck.

#### A2. CheckMethodGuid

This GUID determines the method in the CheckMethods assembly that is executed to begin the check. In C# code, the GUID is set with the CheckMethodAttribute class of the MetaAutomationClientSpLibrary.

#### A3. SemaphoreTimeoutMilliseconds

Global operating system semaphores are used to synchronize checks with subchecks, and these have timeouts defined. If a semaphore times out, the check might end with no artifact record of the check run, so judicious use of the check step timeouts is recommended. See item A5 below.

#### A4. CheckLibraryAssembly

For this simple sample, the check code all lives in this one assembly, so the name of the assembly is specified here.

#### A5. CheckStep msTimeLimit

These values determine the timeout for a check step, including all child steps that run as part of that step.

This value is set for any existing check step for a check. If the check steps change or are initialized at first run of a given check, the msTimeLimit will default to the value determined by the MetaAutomation libraries (currently 30,000 milliseconds i.e. 30 seconds).

For check steps that do not change as a result of the check run, the msTimeLimit attribute values will persist to the next check run in the XML.

See entry B3 below for more information on the lifecycle of the CheckStep elements.

#### A6. SubCheckData

This is how a subcheck is defined, i.e. a portion of the check that would run in a different operating system process. These may be nested or defined in peer sequence. The CheckMethodGuid (see item A2) defines what code is run in the new process.

### B. Entries that are determined by the check code and the engine during a check run

#### B1. CheckRunGuid

This uniquely describes a run of a check.

#### B2. CheckBeginTime and CheckEndTime

These values are determined by the run of the check.

#### B3. SubCheckMap

You may ignore this element.

#### B4. CheckObjectStorageKey

You may ignore this element.

#### B5. CheckStep elements and hierarchy

CheckStep elements describe the step hierarchy of a repeatable check.

Before the first run of a given check, the element CompleteCheckStepInfo would have no child elements at all. The artifact that results from the check run will show the CheckStep elements to reflect the steps of the check, following from the code and other elements of the check.

If the steps of a check change, that is detected by the engine during the first run of the check in which the changed steps occur, and the new or changed steps are reflected in the artifact of the check run.

#### B6. CheckStep Name

The name of the check step is determined by the C# code. The names are hard-coded and therefore stable and unchanging, the better to support analysis on the data that is created by the check runs.

#### B7. CheckStep msTimeElapsed

The elapsed time is determined by the check run. The time includes the running time for all child steps as well.

#### B8. CheckStep Value

These values describe the result of a given step in the check run: Pass, Fail, or Blocked.

A result of “Fail” for a given check step propagates up through ancestor steps.

A result of “Blocked” means that the check step was not executed at all due to an earlier failure.

### Troubleshooting

If the checks go on until they time out, verify that the executable “CheckProcess” built successfully. A full rebuild might be required.

If the debugging session continues for more than 10 seconds while the console window continues to report queries for the results for the process, try forcing a debugger attach: In project CheckProcess, class CheckProcessMain.cs, un-comment the 5 lines of code that check Debugger.IsAttached and sleep. Set a breakpoint after this loop. Rebuild all, then start debug, and attach the debugger to process CheckProcess.exe.

If a step times out while a child step is running in the same process, the child step is halted and the root cause of the timeout is reported in an exception. However, if the child step is in a different process, any information in the child process is lost. There might be a semaphore-not-found error reported in this case, as a side effect of the step timing out.

If a semaphore times out due to the SemaphoreTimeoutMilliseconds name/value pair DataElement element in the CheckRunData of the artifact XML, information will probably be lost. Use the msTimeLimit attributes in the check steps to specify timeout intervals. The semaphore timeout is a last resort for recovering resources in case checks fail and run much too long.