Quality Assurance SDK

Contents

[Quality Assurance SDK 1](#_Toc5016487)

[Versions 2](#_Toc5016488)

[Overview 2](#_Toc5016489)

[Necessary software 2](#_Toc5016490)

[Fundamentals of QA in memoQ 2](#_Toc5016491)

[Deploying, configuring and enabling QA addins in memoQ 3](#_Toc5016492)

[Fundamentals of QA addins 3](#_Toc5016493)

[How to implement a QA addin for memoQ 4](#_Toc5016494)

[Structure of a QA addin 4](#_Toc5016495)

[Implementing module behavior 4](#_Toc5016496)

[Implementing QA functinality 5](#_Toc5016497)

[Localization 7](#_Toc5016498)

[Error handling 7](#_Toc5016499)

[Testing in memoQ client 7](#_Toc5016500)

[Addin signing 7](#_Toc5016501)

[Sample addin 8](#_Toc5016502)

[Tips and hints 8](#_Toc5016503)

[Checklist 9](#_Toc5016504)

[Reference 9](#_Toc5016505)

[Interfaces 9](#_Toc5016506)

[IModule 9](#_Toc5016507)

[IPluginDirector 10](#_Toc5016508)

[IBatchQAChecker 11](#_Toc5016509)

[ISegmentLevelQAChecker 12](#_Toc5016510)

[IEnvironment 12](#_Toc5016511)

[Formats 13](#_Toc5016512)

[MQXliff 13](#_Toc5016513)

[Export format 13](#_Toc5016514)

[Addin response format 14](#_Toc5016515)

[Checklist to update a plugin for memoQ 8.0 15](#_Toc5016516)

# Versions

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Who | Change |
| Aug 12, 2013 | 1.0 | SzG | Initial version |
| Sept 18, 2013 | 1.1 | SzG | Update to renames and changes in API |
| Jan 21, 2016 | 1.2 | NG | Segment-by-segment QA checks |
| May 24, 2016 | 1.3 | NG | Problem category in MQXliff response |
| Sep 14, 2016 | 2.0 | DÁ | Changes in memoQ 8.0 |

# Overview

memoQ 6.8 has a new feature regarding Quality Assurance: developers and 3rd party companies are now enabled to implement QA addins in order to customize their Quality Assurance methods.

From the version 7.8.121 memoQ allows segment-level Quality Assurance as well.

The plugin framework since memoQ 8.0 is not compatible with plugins in previous versions of memoQ. This means that plugins developed for previous versions of memoQ will not work in memoQ 8.0 and newer versions. The plugins have to be adjusted and recompiled to target memoQ 8.0.

This document describes the fundamentals of QA and QA Addins in memoQ and gives a guide on how to implement QA addins. It also serves as a reference for the SDK.

**Note**: throughout this document “addin” and “plugin” are considered to be synonyms.

# Necessary software

QA addins are supported from memoQ 6.8. To be able to open the Visual Studio solution found within the SDK, developers need Visual Studio 2013 or later. Addins are basically .NET Class Libraries targeting .NET Framework 4.6.1. In order to run and debug addins the Kilgray AddinSigner tool (distributed with this document) and a memoQ client is needed.

# Fundamentals of QA in memoQ

**Note**: this section describes only the very basics of the already known QA features of memoQ and only covers what is needed for the understanding of addin development. For detailed description refer to the memoQ Help.

Quality Assurance is a way of ensuring the high quality of translation. memoQ has some built-in functionality for QA, like checking number formats for specific languages or ensuring that the tag count is the same on the source and the target side.

These checks can generate errors and warnings about the translation. Errors and some warnings prevents the document from being exported, other warnings do not and can be ignored.

Some type of the built-in QA checks run immediately if the user edits a translation row and checks only that row. In this document they are called *quick* QA checkers. Other QA checkers run only when the user selects “Run QA” command from the “Operations” menu. After the user chooses the command, she selects the scope of the check (this can be a selection, the active document, some documents or the whole project) and runs QA. These checkers process translation rows in batches, so they are called in this document *batch* checkers. This type of checkers is executed when the user selects the Run QA command (if they are enabled) and they process one document at a time.

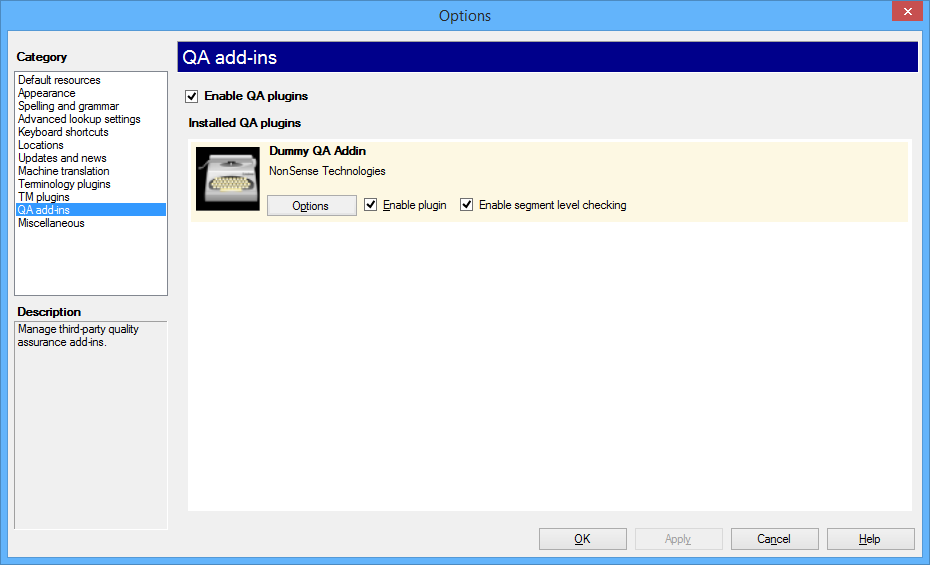
memoQ supports QA addins that work like the built-in quick or the batch checkers.

# Deploying, configuring and enabling QA addins in memoQ

QA addins are modules loaded by memoQ at startup. Modules are .dll files and they are loaded from the <memoQ client installation directory>\Addins folder. The details of creating such .dll files are left for the rest of this document, though, it is important to know to deploy the finished addins to the above mentioned directory.

**Note**: The .dll files need to be signed with Kilgray’s AddinSigner tool for memoQ to load them.

The loaded QA addins are listed under “Tools” / “Options” on the “QA addins” panel.



Execution of QA addins can be disabled as a whole with the checkbox labelled “Enable QA addins”, or individually with each “Enable batch checking”/”Enable segment level checking” checkbox located inside the addin’s panel. On the previous image only one addin is loaded by memoQ; more row appear if more addin are loaded.

Each addin description panel displays the addin’s name, copyright text, image, enabling checkbox and an “Options” button. Clicking this “Options” button displays the addin’s options dialog. What can be found on the addin’s option dialog depends on the addin and the manufacturer.

# Fundamentals of QA addins

When memoQ executes a QA addin, basically four things happen:

* memoQ requests a stream from the addin
* the document is exported into the stream
* the addin processes the document inside the stream and returns an answer stream containing the result of the QA check
* memoQ processes the answer of the stream displaying errors and warnings to the user

This stream oriented approach is useful, as the format of the document and the storage of the data are separated.

Regarding the format the document is exported to, the addin can specify its list of supported formats. In this release of the QA SDK two formats are supported by memoQ: a custom Xliff variant called *MQXliff*, and xliff:doc. Detailed description of the MQXliff format can be found in the Reference section of this document. Xliff:doc format is not covered in this document, since it is the standard xliff:doc format used by many translation tools.

MQXliff is a format defined for being the standard, default export format for QA addins. It is based on the Xliff 1.2 standard using only a small subset of the standard. Every document in memoQ can be exported to this format.

On the other hand, a document in memoQ can only be exported into xliff:doc format if it was originally imported from that format.

The approach to formats in the SDK is based on the difference of the two supported formats mentioned above: there are *default* formats which every document can be exported to, and there are *special* formats which only those documents can be exported to whose original format is that format. Now only one format falls into each category, but this is expected to grow in the future.

When a QA addin defines its list of supported formats the logic choosing the export format is the following:

* memoQ checks the first element in the list whether the document can be exported to that format
* If yes, the document is exported into that format. If no, the same is done for the second, third, etc. element in the list.
* If memoQ reaches the end of the list, meaning there are no supported formats of the addin that the document can be exported into, then the addin will not be executed for that document.

As a consequence of this logic, an addin is not obligated to support *any* of the formats. Though an addin supporting no format is not really useful, if an addin wants to support only e.g. xliff:doc, it is completely fine, it is not needed to implement support for MQXliff.

# How to implement a QA addin for memoQ

## Structure of a QA addin

A QA addins is basically a .NET Class Library targeting .NET Framework 4.6.1.

**Note**: Given the features of the .NET platform, the addin can be developed in any of the CLS-complaint .NET languages (C#, VB.NET, etc.). However, in this document illustrations are found in C#, and Kilgray can offer support only for C# language, so developing in C# is highly recommended.

**Important note**: Developers must NOT use native code in QA addins. Similarly, unsafe code is also prohibited.

Three memoQ specific dlls need to be referenced (among the commonly used .NET dlls and any other assemblies needed for the addin):

* MemoQ.Addins.Common.dll
* MemoQ.QAInterfaces.dll

Please note that these are the sole memoQ assemblies that should be referenced.

This has changes in memoQ 8.0, please make sure to update the references!

## Implementing module behavior

In order to utilize this feature, a class implementing IModule interface has to be defined, and an assembly level attribute, called ModuleAttribute needs to be placed on the assembly. Both types are found in the MemoQ.Addins.Common.Framework namespace, so a using directive at the beginning of the source files can be useful.

The IModule interface consists of three members:

* IsActivated: a boolean getter-only property returning whether the addin has been properly activated for the user. Addins not activated will not be executed when running QA.
* Initialize: a method that is executed when memoQ loads the addin. Initialization logic can be implemented here.
* Cleanup: a method that is executed when memoQ terminates. Cleanup logic can be implemented here.

**Note**: the class that implements IModule also has to extend the later described PluginDirector abstract class.

After implementing the IModule interface, the ModuleAttribute needs to placed on the assembly. Its most convenient place in a Visual Studio Class Library project in C# is the project’s AssemblyInfo.cs file:

[assembly: Module(ModuleName = “MyCompany QA”, ClassName = ”MyCompany.QAAddin.PluginDirector”)]

The ModuleName field should be assigned a descriptive name of the addin, and the ClassName should be the full qualified name of the class implementing IModule. When memoQ loads the module, it creates an instance of the class implementing the IModule interface. The class is found by the ClassName given to the ModuleAttribute. Be careful when placing this attrubite on the assembly, as a typo in the ClassName field causes memoQ to fail to load the addin.

## Implementing QA functinality

MemoQ.QAInterfaces.dll defines three contracts the addin needs to implement:

* PluginDirector abstract class
* IBatchQAChecker interface
* ISegmentLevelQAChecker

The PluginDirector abstract base class defines properties and methods that in general applies to the addin itself, not the actual QA check. These are:

* PluginID: a string ID of the addin, should be unique and contain no whitspace.
* FriendlyName: a string holding a user-friendly name of the addin. This name is displayed to user under Tools / Options / QA addins
* GeneratorID: an integer number between 10 and 99. Obtained from Kilgray. This number distinguishes the errors and warnings generated by different QA addins. The actual code of an error or warning consists of three additional digits, but the meaning of those digits are specific for the addin.
* DisplayIcon: A 48x48 pixel image to display under Tools / Options in memoQ.
* CopyrightText: The copyright text to display under Tools / Options in memoQ.
* BatchQACheckSupported: A boolean value whether the addin supports batch processing of translations.
* CreateBatchQAChecker: returns an instance of the addin’s IBatchQAChecker implementation. The implementing object will only be used in a single threaded manner and will be disposed after usage.
* SegmentLevelQACheckSupported: A boolean value whether the addin supports segment level processing of translations. Virtual method, returns false by default.
* CreateSegmentLevelQAChecker: returns an instance of the addin’s ISegmentLevelQAChecker implementation. The implementing object will only be used in a single threaded manner and will be disposed after usage. Virtual method, returns null by default.
* ShowOptionsForm: a method that is called when the user clicks the options button of the plugin under Tools / Options in memoQ.
* SupportedBatchFormats: a string array containing the names of the supported formats by the batch checker. The possible string values are found in the class MemoQ.QAInterfaces.ExportFormats as const strings.
* SupportedSegmentLevelFormats: a string array containing the names of the supported formats by the segment level checker. The possible string values are found in the class MemoQ.QAInterfaces.ExportFormats as const strings. Virtual property, returns an empty string array by default.
* RequiredRequestVersion: The request format required by the addin for the batch and/or segment level checks. It has effect only if the chosen format is MQXliff. The possible values are found in the enum MemoQ.QAInterfaces.MqXliffRequestVersion.
* ResponseVersion: The response format of the addin for the batch and/or segment level checks. memoQ will take it into consideration if the chosed format is MQXliff. The possible values are found in the enum MemoQ.QAInterfaces.MqXliffResponseVersion.
* Environment: a write-only (setter only) property that the framework sets for the addin. The addin can obtain read-only information about the environment through this property. This is now limited to the current UI languge for localization purposes.

**Note**: The PluginDirector is defined as an abstract class in order to be able to load and run older QA addins when this PluginDirector contract is extended in the future.

**Note**: the same class have to implement the IModule interface and extend the PluginDirector abstract class. It makes also sense, since exactly one instance of the IModule implementer will be created and the same amount is needed of the PluginDirector extender. However, it is not possible to separate the two classes.

The IBatchQAChecker interface defines member directly needed for running QA:

* GetStreamForDocument: this method should return a stream, which the document will be exported into. The method has a transUnitCount parameter which gives how many translation rows are going to be exported to the stream, and can be used as a hint for the type of stream.
* PerformCheck: the method has three parameters: the stream containing the document (obtained from the GetStreamForDocument method), the transUnitCount parameter, and a string describing the chosen document export format by memoQ. The method has to return a stream containing the result of the QA check. This is the place to implement the QA check logic. The format of the result is the same as the input format, with a minor difference in MQXliff. Details are found in the reference section.
* FinishedProcesingAnswer: This method has a stream input parameter. By calling this method memoQ signals to the addin that the answer stream is no longer needed by memoQ so the addin can dispose it, if it is the appropriate thing to do.

The IBacthQAChecker interface inherits from the IDisposable interface, so it also has a Dispose method. After memoQ finished using an IBatchQAChecker implementer, the Dispose method is called. Unclosed streams can be closed, temporary files can and should be deleted, and any other terminating logic should be implemented there.

The ISegmentLevelQAChecker interface defines member directly needed for running QA:

* GetStreamForSegment: this method should return a stream, which the segment will be exported into. The export format is the same as in case of the batch check, but the document contains exactly one translation unit.
* PerformCheck: the method has two parameters: the stream containing the document with exactly one translation unit (obtained from the GetStreamForSegment method) and a string describing the chosen document export format by memoQ. The method has to return a stream containing the result of the QA check. This is the place to implement the QA check logic. The format of the result is the same as the input format, with a minor difference in MQXliff. Details are found in the reference section.
* FinishedProcesingAnswer: This method has a stream input parameter. By calling this method memoQ signals to the addin that the answer stream is no longer needed by memoQ so the addin can dispose it, if it is the appropriate thing to do.

The ISegmentLevelQAChecker interface inherits from the IDisposable interface, so it also has a Dispose method. After memoQ finished using an ISegmentLevelQAChecker implementer, the Dispose method is called. Unclosed streams can be closed, temporary files can and should be deleted, and any other terminating logic should be implemented there.

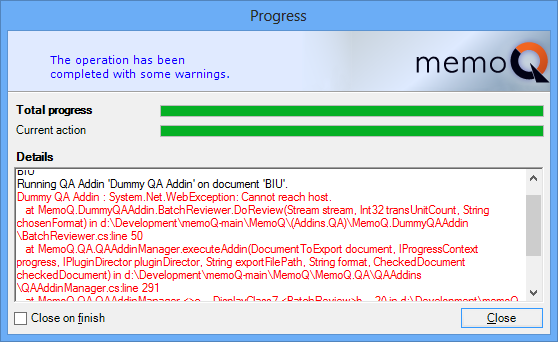
## Localization

The current UI language of memoQ can be obtained through the PluginDirector’s Environment property. It contains the two-letter language code of the UI language. This can be used to localize e.g. the options dialog for the user. The currently available UI languages of memoQ are English, German, Spanish, French, Hungarian, Japanese, Polish, Portuguese and Russian.

However, the actual error and warning messages (e.g. “the number format on the target side is incorrect”) should not be localized, since they are imported and exported with some document formats and changing the UI language of memoQ will not change the language of error or warning messages.

## Error handling

QA addins should handle errors gracefully, and only propagate them to the user if the addin can do nothing about the error (like no network connection). Exceptions thrown by the addin are caught by the QA Addin Framework and dumped to the user in the job log. That means that an addin can throw virtually any kind of exception (built-in BCL or custom type), the framework will catch it.



The message dumped to the user consists of the plugin’s name, the exception’s message and stack trace.

## Testing in memoQ client

You can test your QA plugin in the memoQ client from version 7.8.55. First copy your plugin dll file into the *Addins* folder in the installation folder of memoQ client. By default memoQ does not load unsigned plugins. To enable loading your plugin you have to create an XML file named ClientDevConfig.xml in the *%programdata%/MemoQ* folder with the following content:

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

<ClientDevConfig>

<LoadUnsignedPlugins>true</LoadUnsignedPlugins>

</ClientDevConfig>

Now memoQ will load your plugin if it was implemented correctly.

## Addin signing

QA addins need to be signed with Kilgray’s AddinSigner tool for memoQ to load them at startup. The tool is distributed along this document (MemoQ.AddinSigner.exe). It is a command line application.

When an addin is signed for the first time, a key pair needs to be generated. Usage:

MemoQ.AddinSigner.exe –g <prefix>

The prefix should be name of the addin assembly without extension. For example if the addin is called MyCompany.QAAddin.dll, then the command should be:

MemoQ.AddinSigner.exe –g MyCompany.QAAddin

Running this command generates two files: <prefix>PublicKey.xml and <prefix>PrivatePublicKey.xml. In the example above the output files will be MyCompany.QAAddinPublicKey.xml and MyCompany.QAAddinPrivatePublicKey.xml. The first file contains the public key, and the second files contains both the private and public key.

**Note**: The file containing the public key can and should be made available to public (at least for Kilgray as described later), but the file containing the private key should never get out of the developer company’s possession.

**Note**: After the key files are generated for an addin, there is no need (and discouraged) to generate new key files each time a change is made to the addin.

**Note**: The generated key files can be used to sign multiple addins written by the company, but it is also reasonable to generate separate key pairs for each addin.

After that the addin assembly should be signed with the private key. Usage:

MemoQ.AddinSigner.exe –s <assembly path> <private key file path>

For the two path parameters both relative and absolute paths can be used. In the above example assume that the assembly and the private key file are located under “C:\Plugins”. Then the following command will sign the addin:

MemoQ.AddinSigner.exe –s C:\Plugins\MyCompany.QAAddin.dll C:\Plugins\MyCompany.QAAddin.PrivatePublicKey.xml

This command generates a .kgsign file, like MyCompany.QAAddin.kgsign. This file should be deployed right next to the addin assembly, under “<memoQ installation folder>\Addins”.

The developer company should send its **public** key file to Kilgray (<prefix>PublicKey.xml). Kilgray then will prepare memoQ for validating the QA addin’s digital signature with the given public key. These changes will be available in the next public build of memoQ (which usually occurs in a two week period).

## Sample addin

There is a sample addin provided with the QA Addin SDK, that can be used as a starting point or a sample for other addins. It follows the guidelines described in the document and implements the necessary interfaces. It also gives helper classes and an object model for parsing the input MQXliff format, inserting errors and warnings in an object oriented way, and writing the output MQXliff format.

The sample addin does not deal with formats other than MQXliff, so there is no sample code for handling xliff:doc.

## Tips and hints

Here are some hints the developer of a QA addin might want to consider:

* Do NOT use native or unsafe code!
* Deploy finished addins to “<memoQ Client installation directory>\Addins”. The client installation directory usually looks like “C:\Program Files (x86)\Kilgray\memoQ<version>”.
* When a stream is requested from the addin for batch checking, the count of translation rows are passed to the addin as a hint. For few rows MemoryStream is efficient, for many rows FileStream is advisable. Do not use too much memory.
* On the Options form of the addin, when the addin has to perform a long operation (network communication, web service call, etc.) do it on a background thread, not on the main/GUI thread
* If needed, the addin should save options to <IModuleEnvironment. PluginSettingsDirectory>\<PluginName>Settings.xml.
* The easiest way to save settings is to define a class holding the values of the settings and XML-serialize an instance of it.
* On the Options form only save the settings when the user clicks the OK button. Allow to save settings only when all the mandatory parameters are set correctly.
* Debugging the addin is possible by placing the addin’s assembly into memoQ’s addin folder (<Program Files>\Kilgray\<memoQ folder>\Addins), running memoQ and attaching a Visual Studio debugger to the process. The corresponding .pdb file may be needed as well.   
  Another way for debugging is logging into a file.

## Checklist

Here is a checklist that a developer can follow and should check when the addin is finished:

* The addin is a .NET Class Library (.dll) targeting .NET Framework 4.6.1
* The addin references MemoQ.Addins.Common.dll and MemoQ.QAInterfaces.dll
* There is a PluginDirector class implementing MemoQ.Addins.Common.Framework.IModule AND extending MemoQ.QAInterfaces.PluginDirector
* The PluginDirector class properly returns the supported formats as a string array. The preferred format should be at the beginning of the array.
* The PluginDirector class correctly returns the GeneratorID obtained from Kilgray.
* MemoQ.Addins.Common.Framework.ModuleAttribute is placed on the assembly with proper ClassName parameter.
* Optionally there is a BatchQAChecker class implementing MemoQ.QAInterfaces.IBatchQAChecker
* Optionally there is a SegmentLevelQAChecler class implementing MemoQ.QAInterfaces.ISegmentLevelQAChecker
* No native or unsafe code is used
* Each and every stream given by the addin is correctly disposed after it is not needed anymore. It is the addin’s responsibility to dispose the streams.
* The addin is signed with the provided AddinSigner tool.
  + The public key file is sent to Kilgray. (Needs to be done only once per key pair.)
  + The .kgsign file is deployed next to the addin.

If you use the sample addin as a starting point, make sure to do the following:

* Change the name and the default namespace of the project
* Change the namespace in every file
* Change the ClassName string of the ModuleAttribute in the AssemblyInfo.cs file according to the new namespace and class name of the plugin director class
* In the PluginDirector class at the DisplayIcon property change the string containing the name of the embedded resource. Basically it is <root namespace>.<file name>, e.g. MyCompany.QAAddin.icon.bmp. If it is inside a folder called for example Resources, then the resource string should be MyCompany.QAAddin.Resources.icon.bmp.
* Change the GeneratorId value in the PluginDirector class.

# Reference

## Interfaces

### IModule

This interface defines the basic functionality for being loaded at startup and disposed at termination.

public interface IModule

{

bool IsActivated { get; }

void Initialize(IModuleEnvironment env);

void Cleanup();

}

### IPluginDirector

This interface defines members for accessing information about the addin and creating batch and/or segment level reviewers.

/// <summary>

/// MemoQ's starting point to the plugin. One instance will be created at

/// application startup, and disposed at shutdown.

/// </summary>

public abstract class PluginDirector

{

/// <summary>

/// Return a 48x48 display icon to show in MemoQ's Tools / Options.

/// Black is the transparent color.

/// </summary>

public abstract Image DisplayIcon { get; }

/// <summary>

/// Return the friendly name to show in MemoQ's Tools / Options.

/// </summary>

public abstract string FriendlyName { get; }

/// <summary>

/// The plugin's non-localized name.

/// </summary>

public abstract string PluginID { get; }

/// <summary>

/// The first to digits of error or warning type id. Obtained from Kilgray.

/// </summary>

public abstract int GeneratorID { get; }

/// <summary>

/// Return the copyright text to show in MemoQ's Tools / Options.

/// </summary>

public abstract string CopyrightText { get; }

/// <summary>

/// Returns if plugin supports QA check for a document at one call.

/// </summary>

public abstract bool BatchQACheckSupported { get; }

/// <summary>

/// Returns an instance of the plugin's IBatchQAChecker implementation.

/// It will not be used in a multi-threaded environment.

/// </summary>

public abstract IBatchQAChecker CreateBatchQAChecker();

/// <summary>

/// Gets whether the plugin supports segment-level QA checks.

/// </summary>

public virtual bool SegmentLevelQACheckSupported

{

get { return false; }

}

/// <summary>

/// Returns an instance of the plugin's ISegmentLevelQAChecker implementation.

/// It will not be used in a multi-threaded environment.

/// </summary>

public virtual ISegmentLevelQAChecker CreateSegmentLevelQAChecker()

{

return null;

}

/// <summary>

/// Show the plugin's options/about form

/// </summary>

public abstract void ShowOptionsForm(Form parentForm);

/// <summary>

/// Returns the list of formats supported by the batch checker.

/// The order should be by preference descending.

/// </summary>

public abstract string[] SupportedBatchFormats { get; }

/// <summary>

/// Returns the list of formats supported by the segment level

/// checker. The order should be by preference descending.

/// </summary>

public virtual string[] SupportedSegmentLevelFormats

{

get { return new string[0]; }

}

/// <summary>

/// Returns the request format required by the plugin for

/// the batch and/or segment level checks. It has effect

/// only if the chosen format is MqXliff.

/// </summary>

public virtual MqXliffRequestVersion RequiredRequestVersion

{

get { return MqXliffRequestVersion.V1; }

}

/// <summary>

/// Returns the response format of the plugin for the

/// batch and/or segment level checks. memoQ will take

/// it into consideration if the chosen format is MqXliff.

/// </summary>

public virtual MqXliffResponseVersion ResponseVersion

{

get { return MqXliffResponseVersion.V1; }

}

/// <summary>

/// The memoQ QA Addin Framework sets this property to provide

/// information about the environment (e.g. current UI language

/// for localization purposes)

/// </summary>

public abstract IEnvironment Environment { set; }

}

### IBatchQAChecker

/// <summary>

/// This interface defines the contract the object performing the

/// batch QA check has to implement.

/// </summary>

public interface IBatchQAChecker : IDisposable

{

/// <summary>

/// Returns a stream that the QA addin framework will use to

/// write the exported document into it (in one of the supportted

/// formats). The framework asks for one stream per document.

/// </summary>

/// <param name="transUnitCount">

/// The number of translation units (source and target segment pairs)

/// in the document. It is a hint for the addin about the type of stream

/// it should return. For small numbers memory stream is a good choice,

/// for large documents file stream is a good choice.

/// </param>

/// <remarks>

/// The addin should record the streams the framework asked for, since

/// it should perform the QA check on those streams.

/// </remarks>

Stream GetStreamForDocument(int transUnitCount);

/// <summary>

/// Perform QA check on the requested stream.

/// </summary>

/// <param name="stream">

/// It is the stream that the framework requested from the addin by calling

/// GetStreamForDocument.

/// </param>

/// <param name="transUnitCount"><see cref="GetStreamForDocument"/></param>

/// <param name="chosenFormat">

/// The name of the format the QA Addin Framework chose to export the document

/// into. It is one of supported formats specified by

/// <see cref="PluginDirector.SupportedBatchFormats"/>.

/// </param>

/// <returns>

/// The stream containing the \_answer\_ document with warnings and errors.

/// </returns>

/// <remarks>

/// The parameter stream is not closed by the framework, the addins should

/// close it after it is not needed anymore. The framework will not close the

/// answer stream either.

/// </remarks>

Stream PerformCheck(Stream stream, int transUnitCount, string chosenFormat);

/// <summary>

/// A signal from the framework that it has finished processing the answer

/// of the addin, so the addin can release the stream.

/// </summary>

/// <param name="stream">

/// The answer stream taken from PerformCheck's return value.

/// </param>

void FinishedProcessingAnswer(Stream stream);

}

### ISegmentLevelQAChecker

/// <summary>

/// This interface defines the contract the object performing the

/// segment-level QA check has to implement.

/// </summary>

public interface ISegmentLevelQAChecker : IDisposable

{

/// <summary>

/// Returns a stream that the QA addin framework will use to

/// write the exported segment into it (in one of the supportted

/// formats). The framework asks for one stream per segment.

/// </summary>

/// <remarks>

/// The addin should record the streams the framework asked for,

/// since it should perform the QA check on those streams.

/// </remarks>

Stream GetStreamForSegment();

/// <summary>

/// Perform QA check on the requested stream.

/// </summary>

/// <param name="stream">

/// It is the stream that the framework requested from the addin by

/// calling GetStreamForSegment

/// </param>

/// <param name="chosenFormat">

/// The name of the format the QA Addin Framework chose to export the

/// segment into. It is one of supported formats specified by

/// <see cref="PluginDirector.SupportedSegmentLevelFormats"/>.

/// </param>

/// <returns>

/// The stream containing the \_answer\_ segment with warnings and

/// errors.

/// </returns>

/// <remarks>

/// The parameter stream is not closed by the framework, the addins

/// should close it after it is not needed anymore. The framework will

/// not close the answer stream either.</remarks>

Stream PerformCheck(Stream stream, string chosenFormat);

/// <summary>

/// A signal from the framework that it has finished processing

/// the answer of the addin, so the addin can release the stream.

/// </summary>

/// <param name="stream">

/// The answer stream taken from PerformCheck's return value.

/// </param>

void FinishedProcessingAnswer(Stream stream);

}

### IEnvironment

/// <summary>

/// Interface holding read-only information about the environment the addin runs in.

/// </summary>

public interface IEnvironment

{

/// <summary>

/// Two-letter language code of the current User Interface language.

/// </summary>

string UILanguage { get; }

}

## Formats

In the library MemoQ.QAInterfaces there is a class defined for holding the string constant values of format names.

public class ExportFormats

{

public const string XliffDoc = "xliff:doc";

public const string MqXliff = "MemoQ-Xliff";

}

In the library MemoQ.QAInterfaces there is an enum defined for holding the values of the supported MQXliff request versions.

/// <summary>

/// The possible versions of the MqXliff request xml

/// sent from the memoQ to the plugin.

/// </summary>

public enum MqXliffRequestVersion

{

/// <summary>

/// The version conform to Export-MqXliff-V1.xsd.

/// It does not contain the row-level information

/// comments.

/// </summary>

V1 = 0,

/// <summary>

/// The version conform to Export-MqXliff-V2.xsd

/// It contains the row-level information comments.

/// </summary>

V2 = 1,

/// <summary>

/// The version conform to Export-MqXliff-V3.xsd

/// It contains the row-level "mq:maxlengthchars" attributes.

/// </summary>

V3 = 2

}

In the library MemoQ.QAInterfaces there is an enum defined for holding the values of the supported MQXliff response versions.

/// <summary>

/// The possible versions of the MqXliff response xml

/// sent from the plugin to the memoQ.

/// </summary>

public enum MqXliffResponseVersion

{

/// <summary>

/// The version conform to AddinAnswer-MqXliff.xsd.

/// </summary>

V1 = 0

}

## MQXliff

### Export format

This format is a stripped down Xliff 1.2 variant with memoQ specific extensions. This example demonstrates the format (the nodes highlighted with red can appear only if the required request version is V2, the attribute highlighted with green can appear only if the required request version is V3):

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

<xliff xmlns="http://www.kilgray.com/2013/mqxliff-qaaddin" xmlns:mq="http://www.kilgray.com/2013/mqxliff-qaaddin" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.kilgray.com/2013/mqxliff-qaaddin Export.xsd">

<file source-language="en" target-language="hu" original="memoq-qa-addin-framework" datatype="x-memoq" mq:documentguid="e31c4bbc-6ed0-44a9-9748-b386f37944fd">

<body>

<trans-unit id="f1be833d-d5bb-4f13-b886-c98b7b8ec0e9"

mq:originalhash="1966907221" mq:segmenthash="80205031" mq:maxlengthchars="30">

<source xml:space="preserve"><bpt id="1" ctype="bold" />red<ept id="1" /></source>

<target xml:space="preserve">piros</target>

<note>row-level information comment 1</note>

<note>row-level information comment 2</note>

</trans-unit>

<trans-unit id="d6a611f0-8cdc-4f9c-adc0-fc4e566c0fb0" mq:originalhash="1044819561" mq:segmenthash="-335173644">

<source xml:space="preserve"><bpt id="1" ctype="italic" />green<ept id="1" /></source>

<target xml:space="preserve">zöld</target>

</trans-unit>

</body>

</file>

</xliff>

You can find the describing XSD document in the Export-MqXliff-<V1|V2|V3>.xsd file.

Important tags and attributes:

* xliff: root element
* file: represents a document
  + mq:documentguid: the ID of the document in memoQ
* body: the body of the document
* trans-unit: a pair of text which are each other’s translation
  + id: the ID of the translation row in memoQ
  + mq:segmenthash: hash of the target text
  + mq:originalhash: hash of the source and target text pair
  + mq:maxlengthchars: maximum number of characters allowed in translation (set by xliff converter only)
* bpt: begin paired tag, used for formatting, denoting start
* ept: end paired tag, used for formatting, denoting end
* ph: placeholder for memoQ Inline tags with content
* x: placeholder for memoQ structural tags

It is important that in the response the QA Addin Framework gets back the attributes starting with mq: and the id attribute of trans-units correctly.

### Addin response format

The response format is similar to the export format with a few differences:

* markers can be placed in the target text to mark positions
* errors and warnings can be placed inside the trans-unit tag
* errors and warnings can refer to markers if the error or warning is about a specific range.

For example:

<target xml:space="preserve">4 T<mrk mtype="x-ewloc" mid="1" />ervra<mrk mtype="x-ewloc" mid="2" />jz</target>

<mq:errors><mq:errorwarning mq:code="23438" mq:category="spelling-and-grammar" mq:problemname="High voltage" mq:shorttext="You can get burned" mq:longdesc="You shall respect electricity otherwise you can get burned, shocked and will not be able to translate" mq:dataspecific="true" mq:ignorable="false" mq:range-start-mid="1" mq:range-end-mid="2" />

</mq:errors>

You can find the describing XSD document in AddinAnswer-MqXliff.xsd.

Description of the attributes of mq:errorwarning tag:

* code: numeric identifier of the problem
  + first two digits represents the addin itself
  + other three digits identify the problem
* category: the category of the problem, possible values:
  + uncategorized
  + code-page
  + consistency
  + formatting
  + numbers
  + punctuation-and-whitespace
  + segment-level-problems
  + spelling-and-grammar
  + tags
  + terminology
* problemname: name of the problem, unique to each code
* shorttext: short description of the problem
* longdesc: long description of the problem
* dataspecific: true if the problem is not general, but certain expressions in the text can be blamed
* ignorable: true if the user can ignore it, false, if the error or warning prevents exporting the document. errors are never ignorable
* range-start-mid, range-end-mid: id of the markers. these two markers select the range to highlight. can be omitted then the beginning and ending of the text is assumed. if none is present then no range is highlighted

# Checklist to update a plugin for memoQ 8.0

Given an existing plugin and its codebase the following steps describe the process to update the library to make it compatible with memoQ 8.0

* Target .NET 4.6.1 or newer version.
* Remove all memoQ codebase references, except for MemoQ.QAInterfaces.dll, and add MemoQ.Addins.Common.dll as reference.
* Update the AssemblyInfo.cs file, replace the namespace of MemoQ.Common.Framework to MemoQ.Addins.Common.Framework (for ModuleAttribute attribute).
* Update the implementation class of the IPluginDirectory interface to match the slightly changed interface.
* Compile the library and fix compilation errors. In most cases no changes will be required, or only namespaces have to be fixed.
* Test the new plugin with the test client part of the SDK, or memoQ 8.0.
* Deploy the new plugin.