

Machine Translation

SDK

© memoQ Zrt.

All rights reserved.  
www.memoQ.com

© 2004-2018 memoQ Translation Technologies.

All rights reserved.  
www.memoQ.com

© 2004-2018 memoQ Translation Technologies.

All rights reserved.  
www.memoQ.com

© 2004-2018 memoQ Translation Technologies.

All rights reserved.  
www.memoQ.com

Contents

[Machine Translation 1](file:///D:\GIT_repos\memoq-sdk\src\MemoQ.SDK.MT\Client\MT_SDK.docx#_Toc117500352)

[SDK 1](file:///D:\GIT_repos\memoq-sdk\src\MemoQ.SDK.MT\Client\MT_SDK.docx#_Toc117500353)

[Versions 3](#_Toc117500354)

[Overview 4](#_Toc117500355)

[The workflow for creating and distributing a plugin 4](#_Toc117500356)

[Creating and distributing a signed private plugin 5](#_Toc117500357)

[Creating and distributing a public MT plugin 5](#_Toc117500358)

[Recommended code exchange infrastructure 6](#_Toc117500359)

[Machine translation framework in memoQ 7](#_Toc117500360)

[Machine translation plugins 7](#_Toc117500361)

[Machine translation interfaces 7](#_Toc117500362)

[IModule interface 7](#_Toc117500363)

[ISession interface 7](#_Toc117500364)

[PluginDirectorBase class 7](#_Toc117500365)

[EngineBase class 7](#_Toc117500366)

[Optional: ISessionForStoringTranslations interface 8](#_Toc117500367)

[Optional: IPluginSettingsMigrator interface 8](#_Toc117500368)

[Supporting tags and formatting 8](#_Toc117500369)

[Space normalization around tags in MT services 9](#_Toc117500370)

[Scale special UI elements to high DPI 10](#_Toc117500371)

[Language data handling in MT plugins 10](#_Toc117500372)

[Machine translation SDK sample application 11](#_Toc117500373)

[Implementation steps of an MT plugin 11](#_Toc117500374)

[Create the new class library 11](#_Toc117500375)

[The plugin director 12](#_Toc117500376)

[IModule 12](#_Toc117500377)

[PluginDirectorBase 12](#_Toc117500378)

[The engine component 14](#_Toc117500379)

[The session for lookups component 15](#_Toc117500380)

[Lookup session with extended parameters 17](#_Toc117500381)

[The session for storing translations component 18](#_Toc117500382)

[Plugin settings 19](#_Toc117500383)

[Migrating settings 19](#_Toc117500384)

[The configuration dialog 20](#_Toc117500385)

[UI Design Guidelines 20](#_Toc117500386)

[Localization 22](#_Toc117500387)

[Implementation checklist 22](#_Toc117500388)

[Testing the sample plugin 23](#_Toc117500389)

[Testing the new plugins 25](#_Toc117500390)

[Testing in the sample application 25](#_Toc117500391)

[Testing a new plugin in memoQ 26](#_Toc117500392)

[Testing a new plugin in memoQ server 26](#_Toc117500393)

[Plugin supported by memoQ Server 26](#_Toc117500394)

[Checklist to update a plugin for memoQ 8.2 26](#_Toc117500395)

# Versions

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Who | Change |
| May 11, 2013 | 1.0 | NG | Initial version |
| May 20, 2013 | 1.0 | BZ | Workflow chapter added, a few fixes |
| May 23, 2013 | 1.0 | NG | Interface definitions added, step-by-step test guide added |
| July 0, 2013 | 1.01 | BZ, BÁ | Dummy service and dummy plugin has been changed to use WCF instead of ASMX web service: this enables the SDK to be used from Express edition of Visual Studio |
| Sept 14, 2016 | 2 | DÁ | Changes related to memoQ 8.0 |
| Apr 10, 2017 | 2.1 | DÁ | Supporting adaptive MT, i.e. sending translations back to the MT engine |
| July 14, 2017 | 3 | JM | Changes related to memoQ 8.2: plugin settings and base class changes |
| Dec 13, 2018 | 3.1 | GyK | Changes related to memoQ 9.0: ShowHelp  Update Localization (GetResourceString return value was incorrect)  Overview: DummyMTPlugin |
| Apr 24, 2019 | 3.2 | GyK | Supporting tags and formatting  Target framework change: .NET Framework 4.7.2 |
| Jun 06, 2019 | 3.3 | CsÁ | Help button is required on the OptionsForm in case of public plugins |
| Dec 11, 2019 | 3.4 | GyK | *DisplayIcon*, *SmallIcon* size requirements |
| March 6, 2020 | 3.5 | PMA | *Testing the new plugins in memoQ server* |
| Jun 12, 2020 | 4.0 | PMA | Changes related with segment to XML and HTML converters |
| Jun 20, 2020 | 4.1 | RT | Adding UI Design Guidelines section |
| Oct 28, 2020 | 4.6 | PMA | Adding new whitespace normalization function around tags in MS translations |
| Oct 29, 2020 | 4.6 | PMA | Expansion of the “Implementations checklist” |
| Dec 7, 2020 | 4.7 | PMA | Adding Scale special UI elements to high DPI section |
| Dec 7, 2020 | 4.7 | PMA | Expansion of the “UI Design Guidelines” |
| Mar 9, 2021 | 4.9 | PMA | Consolidate Private MT SDK documentation into public MT SDK documentation. |
| Jan 3, 2022 | 9.10 | PMA | Expand package usage with info: user can select the version of the packages |
| Jan 31, 2022 | 9.10 | PMA | MT SDK to include "no internet connection handling" use case |
| Feb 25, 2022 | 9.10 | PMA | Include missing info about escaping special characters |
| May 11, 2022 | 9.12 | PMA | Language data handling in MT plugins |
| June 6, 2022 | 9.12 | PMA | Target framework change: .NET Framework 4.8. |
| Sept 16, 2022 | 9.14 | PMA | Lookup session with extended parameters |

# Overview

memoQ Ltd. enables customers and 3rd party developers to create machine translation plugins for memoQ. This document describes the machine translation framework’s fundamentals, and provides a step-by-step guide for creating a new plugin.

The documentation describes the MT SDK supported by memoQ 8.2 and newer. To develop plugins for an earlier memoQ version, refer to the documentation of that version. Existing plugins remain compatible – however, those plugins cannot be used on memoQ servers, and they also have limitations in their usage in memoQ.

Plugins need to be developed for *.NET Framework 4.8.* in the *C#* programming language.

The MT SDK has a Visual Studio solution that can be opened by Visual Studio 2015 or higher.

memoQ Ltd. developed a sample machine translation plugin, called DummyMTPlugin. You must use this dummy plugin as the starting point when developing your plugin. For more information, please see section *Machine translation SDK sample application*. Please remember to check the implementation steps below in section Implementation checklist.

# The workflow for creating and distributing a plugin

When CompanyA wants to create a new MT plugin, first they need to develop a new MT plugin based on the MT SDK, in C#. When the plugin is ready, CompanyA needy to choose its type. memoQ supports these three types:

* **Unsigned private MT plugin**: Without any memoQ-side code review, CompanyA can just distribute their plugin directly to users – or make it publicly available for download from a shared folder or website. memoQ users can simply download the files, and use the plugin in memoQ and in memoQ server as well. To learn more about unsigned private MT plugins, see sections “[Testing a new plugin in memoQ client](#_Testing_in_memoQ)” and „[Testing a new plugin in memoQ server](#_Testing_the_new)”. Since these plugins do not need any memoQ-side intervention, CompanyA does not need to wait for a new memoQ release to use their plugin.
* **Signed private MT plugin**: There is no memoQ-side code review in this case either. CompanyA sends memoQ Ltd. the plugin’s public key. This public key will be stored in memoQ’s code base. It ensures that memoQ can load the plugin without warning the user about unsigned .dll files every time it starts. For details about generating the public key and having your private plugin signed, see section “[Creating and distributing a signed private plugin](#_Creating_and_distributing_1)”. Note: *The plugin itself* will not be a part of memoQ. If CompanyA changes the plugin (even by adding malicious code), the public key will still be valid, and memoQ will still run the plugin without warning the user. Once memoQ Ltd. receives the public key, „signing” the plugin itself takes little time, and the change in the code is typically implemented in the next public memoQ release. This usually means the next maintenance release, so that CompanyA doesn't need to wait for months (until the next feature release) to be able to use the signed plugin.
* **Public (built-in) MT plugin**: Plugins of this type are part of memoQ’s code base, so they are signed, and they are listed among memoQ’s built-in MT services after install. Before integrating into memoQ, CompanyA’s plugin will undergo thorough design, localization, and code reviews as well as testing, to make sure it’s fully compliant with memoQ’s MT SDK and free of potential risks. See the advantages of public MT plugins and the recommended workflow steps of creating a public MT plugin in section “[Creating and distributing a public MT plugin](#_Creating_and_distributing)”. From memoQ’s side, a public MT plugin requires market and product validation, planning, the rounds of reviews and testing detailed above. This means a much longer turnaround time. A public plugin’s maintenance also requires considerable memoQ-side efforts, so such plugins may only be updated *up to twice every year*.

## Creating and distributing a signed private plugin

If CompanyA creates a new MT plugin and they want memoQ to sign it, they need to follow these steps. The workflow’s key point is how the plugin becomes signed:

1. CompanyA generates a key pair for the plugin using MemoQ.AddinSigner.exe. This application is distributed in the MT SDK package. Usage:  
     
   MemoQ.AddinSigner.exe -g <Plugin.Assembly.Name>  
     
   Note: Do not include the file extension in <Plugin.Assembly.Name>
2. This command will generate two files, <Plugin.Assembly.Name>PublicKey.xml and <Plugin.Assembly.Name>PrivatePublicKey.xml.  
   Example: If the plugin’s .dll is *CompanyA.MyPlugin.dll*, the command should be:  
     
   MemoQ.AddinSigner.exe –g CompanyA.MyPlugin

and the resulting files will be *CompanyA.MyPluginPublicKey.xml* and *CompanyA.MyPluginPrivatePublicKey.xml*.

1. CompanyA signs the assembly file with the private key, using memoQ.AddinSigner.exe.

Usage:

MemoQ.AddinSigner.exe -s <assembly\_file\_path> <private\_key\_file\_path>

This will generate a .kgsign file.

Example: If the plugin’s .dll is *CompanyA.MyPlugin.dll*, the already generated private key file is *CompanyA.MyPluginPrivatePublicKey.xml*, and they are both in the *C:\Plugins* folder, the command should be:

MemoQ.AddinSigner.exe -s C:\Plugins\CompanyA.MyPlugin.dll C:\Plugins\CompanyA.MyPluginPrivatePublicKey.xml

The result will be a file named *CompanyA.MyPlugin.kgsign*.

1. CompanyA sends memoQ Ltd. the plugin’s public key file (<Plugin.Assembly.Name>PublicKey.xml). For testing purposes, they also need to send the .dll and the .kgsign files.

memoQ Ltd. needs this public key to validate the assembly’s digital signature. The changes will be available in the next public memoQ release (usually the next maintenance release).

1. CompanyA can now use the plugin. They can even make it available to other users, for example via a shared folder or a website.
   * In memoQ client: users will need to copy the .kgsign file to the *<memoQ\_install\_folder>\Addins* folder, together with the plugin’s .dll file.
   * In memoQ sever: users will need to copy the .kgsign file to the C:\Program *Files\Kilgray\MemoQ Server\Addins* folder, together with the plugin’s .dll file AND change the .kgsign file’s extension to .skgsign.

Note: With this workflow, memoQ does not test the quality of the plugin itself. The plugin’s developer is responsible for quality.

## Creating and distributing a public MT plugin

The recommended workflow in creating a public MT plugin:

1. CompanyA shares the plugin’s source code with memoQ Ltd. (See section [Recommended code exchange infrastructure](#_Recommended_code_exchange) below.)
2. memoQ Ltd. reviews the code and design, and tests the plugin’s functionality. CompanyA performs fixes based on the review’s findings, if needed.
3. memoQ Ltd. compiles the MT plugin’s source code, signs the resulting .dll with its private key, and makes it a part of the memoQ installer. The MT plugin will be distributed with the memoQ client installer from this point.
4. The MT plugin’s source code becomes part of the memoQ code base at memoQ Ltd.
5. Information about bugs reported by customers/testers are forwarded to CompanyA by memoQ Ltd. CompanyA is responsible for fixings these bugs; bug fixes are reviewed by memoQ Ltd.

The above workflow is required to ensure that the plugin meet the quality requirements of memoQ and do not risk the entire product’s stability.

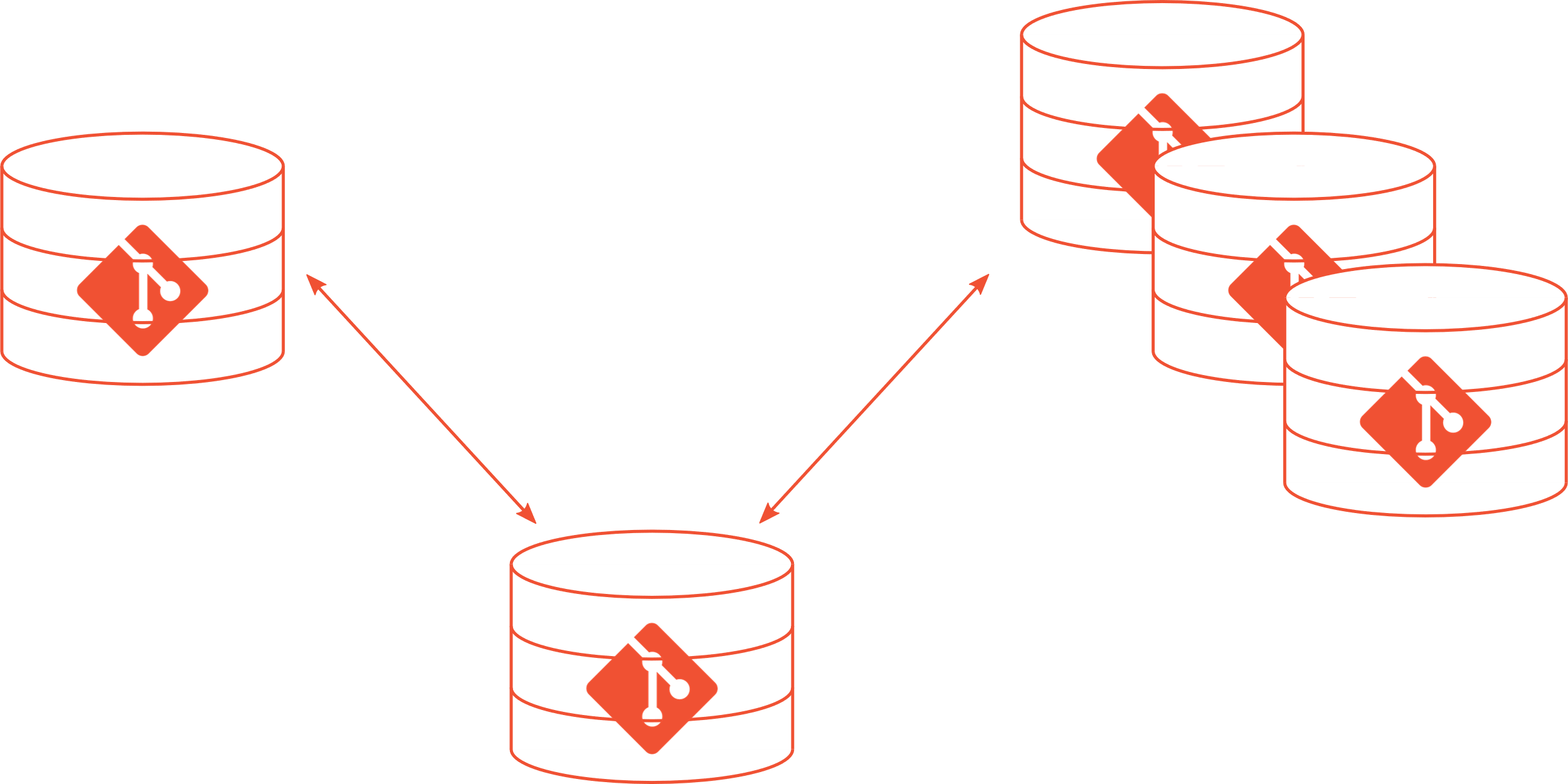
### Recommended code exchange infrastructure

Plugin creators need to follow a standardized workflow when sharing code with memoQ Ltd. The workflow is based on git repositories. The minimal requirement posed by memoQ Ltd. is that the code is available in a git repository and at least one developer from the memoQ dev team has read access to the repository. The *master* branch of that repository needs to contain the most up-to-date version of the plugin’s source code. From time to time, the code will be transferred to the memoQ code base.

This minimal requirement can be matched in many ways, but based on collaboration with multiple plugin developers, memoQ established a required workflow (see Figure 1 below). Three types of repositories are involved in the workflow – memoQ’s code base, the exchange repository, and CompanyA’s code base repositories (if there are any beside the exchange repository). The exchange repository is created and maintained by CompanyA. Actual plugin development should happen one of CompanyA’s repositories – either directly in the exchange repository, or one of the code base repositories.

memoQ Ltd. will include the plugin’s code base from the exchange repository into memoQ’s code base using [git subtree integration](https://github.com/git/git/blob/master/contrib/subtree/git-subtree.txt). This requires read (or possibly write) access to the exchange repository for (1-2) memoQ developers working on the plugin integration. This setup allows memoQ Ltd. to monitor changes in the plugin code. In case write access is also granted, memoQ developers can create and push small fixes to the exchange repository – which is often simpler and faster than asking CompanyA for trivial changes.

This integration also allows full flexibility for CompanyA to prepare the MT plugin’s code to comply with the requirements in the section [Implementation checklist](#_Implementation_checklist).



git subtree integration

MT plugin’s exchange repository

memoQ’s code base memoQ

MT provider’s code bases

Figure 1. Recommended code exchange infrastructure

# Machine translation framework in memoQ

The machine translation framework allows using external translation services in memoQ. memoQ Ltd. delivers several built-in machine translation plugins with memoQ (such as Google MT or Microsoft MT), but companies can also create new machine translation plugins themselves.

## Machine translation plugins

Every machine translation plugin should be a standalone .NET .dll, which has the following references to the memoQ codebase:

* MemoQ.Addins.Common.dll
* MemoQ.MTInterfaces.dll

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

These libraries contain all necessary classes for the plugins. **You must not use any other external libraries in machine translation plugins.** If you think you need to use one, consult with memoQ Ltd.

## Machine translation interfaces

The memoQ application and the plugins can communicate with the help of a few interfaces. Every machine translation plugin should implement these interfaces:

* *MemoQ.Addins.Common.Framework.IModule*
* *MemoQ.MTInterfaces.ISession*

The plugins should also derive from the following base classes:

* *MemoQ.MTInterfaces.PluginDirectorBase*
* *MemoQ.MTInterfaces.EngineBase*

The machine translation plugins may also implement these optional interfaces:

* *MemoQ.MTInterfaces.ISessionForStoringTranslations*
* *MemoQ.MTInterfaces.IPluginSettingsMigrator*

### IModule interface

memoQ manages all plugins as individual modules. This interface provides some general functions for memoQ to be able to initialize and clean up the modules, and to be able to get general information about them.

### ISession interface

memoQ calls the object implementing this interface to perform a lookup. A new session object is created on a segment-by-segment basis, and once for batch operations. ISession objects are always created by engine objects.

### PluginDirectorBase class

This is memoQ’s entry point to the plugin. memoQ instantiates one instance for each plugin at application startup, and this base class is used after this point when memoQ needs to communicate with the plugin.

### EngineBase class

For particular language combinations, memoQ requests an object deriving from the EngineBase class with the plugin director’s help.

### Optional: ISessionForStoringTranslations interface

Implementing the *ISessionForStoringTranslations* interface enables the plugin to store the machine translation results - if the machine translation service supports that behavior.

### Optional: IPluginSettingsMigrator interface

You only need to implement this interface if you have a machine translation plugin created before memoQ 8.2, and now you would like to upgrade it to version 8.2 or higher, OR if you have clients who have machine translation settings created before memoQ version 8.2, and they want to use those settings in newer memoQ versions.

From memoQ 8.2, machine translation plugins no longer store their own settings. Instead, plugin settings are stored in light resources called “MT settings”. When you’re upgrading a legacy machine translation plugin to support version 8.2 or higher, make sure to implement the *IPluginSettingsMigrator* interface, as it allows migrating all your old plugin settings into a new resource file.

## Supporting tags and formatting

Some machine translation providers support tags and formatting as well. These providers usually receive them in HTML or XML format and keep the proper positions of tags and formatting in the translated text. memoQ’s code base already provides the functionality for performing this conversion. You can find these functions in *MemoQ.Addins.Common.Utils.SegmentHtmlConverter* and *MemoQ.Addins.Common.Utils.SegmentXmlConverter*.

* SegmentHtmlConverter
  + ConvertSegment2Html(Segment segment, bool includeTags, bool insertTagPlaceholder = true, Dictionary<char, string> mandatoryRepresentationOfSpecialChars = null)
  + ConvertHtml2Segment(string html, IList<InlineTag> tags)
* SegmentXmlConverter
  + ConvertSegment2Xml(Segment segment, bool includeTags, bool includeFormatting = false, bool convertPairlessTagsAsOpenClose = false)
  + ConvertXml2Segment(string xml, IList<InlineTag> tags)

The *ConvertSegment2Html* and *ConvertSegment2Xml* functions let you decide whether to insert inline tags into the translated text or not. If parameter *includeTags* is true then the request will include memoQ tags, inline tags, and formatting tags. If *includeTags* is false and *includeFormatting* is true, then only formatting tags will be included. If both parameters are false, then the request will contain no kind of tags. The request will not include content of memoQ tags – those will be replaced by unique placeholders.

In the XML converter, some special characters are not allowed as XML content. These need to be escaped, and this requires extra modifications to the segment. This function transforms special characters (currently, ">", "<" and "&") into a *spec\_char* tag whose *val* attribute is the original special character. Example:

**Convert segment to XML:**

“Text & Text“ → **ConvertSegment2Xml** → “Text <spec\_char val="&amp;"/> Text“

**Convert XML to segment (any of the following):**

“Text <spec\_char val="&amp;"/> Text”

“Text <spec\_char val="&"/> Text” → **ConvertXML2Segment** → “Text & Text“

“Text <spec\_char val="&#x26;"/> Text“

“Text <spec\_char val="&#38;"/> Text“

A special character is used to mark a memoQ inline tag’s location (*tagPlaceholders*, see below). This special character was introduced because many times the providers normalized (threw out) memoQ inline tags from segments converted to html. In case of some providers, the translations were better without the markers, so we added an optional parameter to the converter function to specify if the markers are used or not. This parameter's value is true by default. This means that the converter will insert the markers.

In version 9.8 and up, there is another optional parameter for both segment converter methods. In ConvertSegment2Xml, the new parameter is *convertPairlessTagsAsOpenClose*. It indicates if the client wants to convert pairless tags according to their type, or as open/close tag (for example, <br> as <br/>). By default, this parameter is false, so the tags are added according to their type. The ConvertSegment2Html method’s new parameter is *mandatoryRepresentationOfSpecialChars*. This is a dictionary with mandatory representations, keyed by the special characters. It could be useful if the client wants to escape a character in a special way.

To convert the tag placeholders back, you need to give the list of the original segment’s inline tags as a parameter to *ConvertXml2Segment* or *ConvertHtml2Segment*:

var text = SegmentXMLConverter.ConvertSegment2Xml(segment, true);

var translatedText = useTranslationService(text);

return SegmentXMLConverter.ConvertXML2Segment(translatedText, segment.ITags);

In the following example, we have a segment with a formatted word and an inline tag.



In the first case, we need to convert the source segment with the *ConvertSegment2Xml* method before sending it to the MT provider:

This is a sample <b><i>sentence</i></b> with <inline\_tag id="0"/> an inline tag.

If the machine translations provider supports formatting and tags we will get the following response:

Dies ist ein Beispiel <b><i>satz</i></b> mit <inline\_tag id="0"/> einem Inline-Tag.

In the second case, we need to convert the source segment with the *ConvertSegment2Html* method before sending it to the MT provider in two different ways. First, we use the *tagPlaceholders* character:

This is a sample <b>sentence</b> with <span data-mqitag="0">◿</span> an inline tag.

In the second example we convert the segment without *tagPlaceholders*:

This is a sample <b>sentence</b> with <span data-mqitag="0"></span> an inline tag.

The responses should be converted back to a (target) segment with the *ConvertXml2Segment* or *ConvertHtml2Segment* method, respectively. The result is in the picture above, on the right.

## Space normalization around tags in MT services

MT services sometimes return translations with extra spaces (or missing spaces) around tags, which requires a lot of post-editing. In version 9.6, a new function, *TagWhitespaceNormalizer.NormalizeWhitespaceAroundTags* was introduced. It allows normalizing spaces around tags in MT engines’ translation results before displaying in memoQ. If you think the algorithm detailed below could potentially improve the quality of translations received by your plugin, you should consider using this functionality in the implementation of your plugin (maybe depending on user options). This new functionality is accessible as a function in *MemoQ.Addins.Common* with name *TagWhitespaceNormalizer.NormalizeWhitespaceAroundTags*.

* TagWhitespaceNormalizer
  + NormalizeWhitespaceAroundTags(Segment source, Segment target, string srcLangCode, string trgLangCode)

The idea behind the normalization algorithm is to detect extra and unnecessary spaces introduced by MT engines. The algorithm’s inputs are the original source segment, the provided translation, and the source and target language codes. Depending on the languages, the function works like this:

* Translation from non-CCJK languages to CCJK languages:
* If there is a single-byte character before an open or open-close tag, it adds a space between them.
* If there is a single-byte character after a close or open-close tag, it adds a space between them.
* It removes space between tags and double-byte characters.
* In case of translation from CCJK to non-CCJK
* The normalization function returns the target segment without any changes.
* In case of languages with matching CCJK properties
* It returns the target segment with the exact same amount whitespaces around the tags as in the source segment.

## Scale special UI elements to high DPI

In most cases, UI elements are resized dynamically based on display settings (for example: scaling, orientation or resolution), but some Windows Forms elements (for example: button icons) cannot do this. You need to resize these special UI elements manually. From memoQ 9.7, there is a new functionality that can help you resolve such problems. If you have special UI elements that need manual resizing, try using these functions in *MemoQ.Addins.Common*: *DPIHelper.StretchImageDPI* and *DPIHelper.ScaleToHDPI*. Please note that a plugin using these functions will only work with memoQ versions 9.7 or higher.

* DPIHelper
  + StretchImageDPI(Image imageIn, int currentDPI)
  + ScaleToHDPI(int x, int currentDPI)

Use these functions when UI elements need manual resizing. The first input parameters are an image (*imageIn*) or a UI element's size (*x*, an integer) that needs to be scaled to the current system DPI value. The second parameter (*currentDPI*, an integer) is the system DPI value. You can get that value dynamically using these commands:

PropertyInfo dpiXProperty = typeof(SystemParameters).GetProperty("DpiX", BindingFlags.NonPublic | BindingFlags.Static);  
int systemDPIValue = (int)dpiXProperty.GetValue(null, null);

## Language data handling in MT plugins

To make language data handling easier in MT plugins, the MT SDK offers a unit for MT providers, handles efficiently the most frequent language data problems. Currently, memoQ’s *Kilgray.Utils* package is responsible for language data management.

For MT plugin developers, the most useful functionalities are available under the public (and static) class *Kilgray.Utils.LanguageData*. The most useful and frequently used functions available here are *GetIsoCode2LetterFromIsoCode3Letter* (convert 3-letter language codes to 2-letter codes) and *GetIsoCode3LetterFromIsoCode2Letter* (convert 2-letter language codes to 3-letter codes). This class also allows querying all memoQ’s supported languages, or finding useful information about a given language – e.g. whether a language can be used as a source (*IsValidSourceLang*) or as a target (*IsValidTargetLang*) language in memoQ projects, or to discover if the given language is a CCJK (Chinese, Japanese, Korean) language or not (*IsCCJK*), etc.

However, if the plugin needs a particular language’s display name, you should create a new *Kilgray.Utils.Language* object with that language's code. The newly created object will contain the display name. Similarly, the major language code is also associated to the *Kilgray.Utils.Language* type object.

*Kilgray.Utils* is a large and complex package, so we at memoQ are happy to help you if you run into difficulties.

# Machine translation SDK sample application

memoQ Ltd. implemented a small application for the developers who would like to implement new machine translation plugins. Developers will be able to test their machine translation plugins with the help of this application.

You can see three projects if you open the *MT\_SDK* solution from the SDK:

* DummyMTPlugin
* DummyMTService
* TestClient

The sample application is implemented inside the TestClient project. This project references the DummyMTPlugin project, which contains the implementation of a sample machine translation plugin. You must use this dummy plugin as the starting point when developing your plugin.

The third project contains a simple web service, which is used by the sample plugin.

In the next section, we’re going to see how to implement a brand new machine translation plugin.

# Implementation steps of an MT plugin

## Create the new class library

As mentioned above, all plugins should be implemented as standalone libraries. To achieve this, create a new Visual Studio *Class Library* project targeting .NET 4.8. Then mark the assembly with the *MemoQ.Addins.Common.Framework*.*ModuleAttribute* attribute. Open the project’s *AssemblyInfo.cs* file, and insert the following line after the last line (change the module’s name and the plugin director class as needed):

[assembly: Module(ModuleName = "Dummy MT", ClassName = "DummyMTPlugin.DummyMTPluginDirector")]

memoQ will check this attribute when it loads the machine translation assemblies. *ModuleName* should be the machine translation plugin’s name, and *ClassName* should be the plugin director class’s name.

Now you need to set up the memoQ library references. The necessary .dll files are under the *References* folder.

Note: Next to the memoQ libraries Newtonsoft.Json is the only allowed external package. By default, memoQ contains the latest version of the allowed packages, but it is also possible to use a specific version of a package: open *MemoQ.exe.config* with a text editor, and in the *runtime* section, add a new *dependentAssembly* unit to the *assemblyBinding* component. The *assemblyIdentity* element contains identifying information about the assembly, and the *bindingRedirect* element redirects one assembly version to another.

...  
<runtime>   
 <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">   
 <dependentAssembly>   
 <assemblyIdentity name="myAssembly" publicKeyToken="myAssemblyPublicToken" culture="neutral" />   
 <bindingRedirect oldVersion="the default assembly version" newVersion="the assembly version you want to redirect"/>  
 </dependentAssembly>   
 </assemblyBinding>   
</runtime>

## The plugin director

This component is the plugin’s entry point. First of all you need to create a new class inside the project. The naming convention is: *<plugin\_name>PluginDirector.cs*

This class should implement the following interfaces:

* *MemoQ.Addincs.Common.Framework.IModule*

This class should derive from the following base class:

* *MemoQ.MTInterfaces.PluginDirectorBase*

### IModule

This interface has two functions and one property:

* *Cleanup* function: implements the plugin’s cleanup logic.
* *Initialize* function: implements the plugin’s initialization logic.
* *IsActivated* property: tells if the plugin is activated or not.

The interface:

public interface IModule

{

bool IsActivated { get; }

void Initialize(IModuleEnvironment env);

void Cleanup();

}

The *IModuleEnvironment* interface provides information about the environment where the plugin is used, such as a directory path for storing configuration files.

### PluginDirectorBase

This class has seven properties and three functions:

* *BatchSupported* property: tells if the plugin supports batch translation (lookup). memoQ uses batch translation during the pre-translate operations.
* *CopyrightText* property: should return the plugin’s copyright information. This will be shown on the user interface where memoQ lists the available plugins.
* *DisplayIcon* property: should return the MT plugin’s icon. This image will be shown on the user interface where memoQ lists the available plugins. Minimum icon size: 128x128 pixels. Preferred icon size: 256x256 pixels.
* *Environment* property: allows using some basic services. The members of the IEnvironment interface are:
  + *UILang* property: should return the two-letter language code of memoQ’s user interface.
  + *ParseTMXSeg* function: has one string input parameter (a segment in TMX format), and returns the related memoQ *Segment*.
  + *PluginAvailabilityChanged* function: call this function to indicate that your plugin’s availability has changed.
  + *WriteTMXSegment* function: has one input parameter (a memoQ *Segment*), and converts this segment into TMX format. Note that values of the segment’s translatable attributes will not be written into the TMX. Because of this, to keep such information intact, you need to restore the original attribute values after the TMX round-trip.
  + *GetResourceString* function: has one string input parameter (a key), and returns the related localized text.
  + *BuildWordsOfSegment* function: tokenizes a Segment on whitespace and word boundaries.
  + *ShowHelp* function: shows the localized web help; otherwise the deployed (offline) English help. This function is present if the *Environment* property implements the *IEnvironment2* interface, supported from memoQ 9.0. To check this, use

environment.GetType().GetInterface(nameof(IEnvironment2)) != null;

* *FriendlyName* property: should return the plugin’s human-readable name. This will be shown on the user interface where memoQ lists the available plugins.
* *InteractiveSupported* property: tells if the plugin supports interactive translation or not. memoQ uses this information when the user works in the translation grid, and memoQ tries to get translation hits from the machine translation plugin.
* *PluginID* property: should return the plugin’s identifier.
* *StoringTranslationSupported* property: tells if the plugin supports the adaptive (self-learning) behavior.
* *CreateEngine* function: has two input parameters (source and target language). Based on these languages, should instantiate and return a machine translation engine.
* *IsLanguagePairSupported* function: returns if the plugin supports a language pair or not. Do not call any service here, return the result based on the saved plugin settings.
* *EditOptions* function: memoQ calls this function when the user starts configuring your machine translation plugin. Should display the plugin’s configuration dialog.

The class:

/// <summary>

/// Base class for plugin director; implements <see cref="IPluginDirector2"/>

/// </summary>

public abstract class PluginDirectorBase : IPluginDirector2

{

public abstract bool BatchSupported { get; }

public abstract string CopyrightText { get; }

public abstract Image DisplayIcon { get; }

public abstract IEnvironment Environment { set; }

public abstract string FriendlyName { get; }

public abstract bool InteractiveSupported { get; }

public abstract string PluginID { get; }

public abstract bool StoringTranslationSupported { get; }

public abstract IEngine2 CreateEngine(CreateEngineParams args);

public abstract bool IsLanguagePairSupported(LanguagePairSupportedParams args);

public abstract PluginSettings EditOptions(IWin32Window parentForm, PluginSettings settings);

}

## The engine component

memoQ calls the plugin director’s *CreateEngine* function to get a machine translation engine for a language pair (depending on required and supported functionality). memoQ uses this engine to perform the requested type of operation (lookup or store translations).

The engine component should derive from the the *EngineBase* class. The naming convention is: *<plugin\_name>Engine.cs*. Class members are:

* *SmallIcon* property: memoQ displays this icon under translation results when an MT hit is selected from this plugin. Minimum icon height: 128 pixels. Preferred icon height: 256 pixels.
* *SupportsFuzzyCorrection* property: tells if the engine supports the adjustment of fuzzy TM hits through machine translation (MatchPatch). This means that if there is a TM match for the source segment, but it is not perfect, memoQ will try to improve the suggestion by sending the difference to an MT provider for translation. If your MT service can only translate complete segments reliably, but not partial ones (e.g., two separate words), disable this feature. But if the service is good at translating segment parts, enable it. If the feature is disabled, your plugin will not appear in the *MatchPatch* list on the *Edit machine translation settings* dialog's *Settings* tab. To learn more about MatchPatch, see [our Documentation](https://docs.memoq.com/current/en/Places/translation-results-list.html?#matchpatch).
* *SetProperty* function: sets an engine-specific property, for example, subject matter area.
* *CreateLookupSession* function: memoQ calls this function to be able to perform the translations. Instantiate and return a session object here. This session will not be used in a multi-threaded way.
* *CreateStoreTranslationSession* function: memoQ calls this function to store translations if the plugin supports adaptive behavior. You should instantiate and return a session object here. This session will not be used in a multi-threaded way.

The class:

/// <summary>

/// Base class for engines; implements <see cref="IEngine2"/>.

/// </summary>

public abstract class EngineBase : IEngine2

{

public abstract Image SmallIcon { get; }

public abstract bool SupportsFuzzyCorrection { get; }

public abstract void SetProperty(string name, string value);

public abstract ISession CreateLookupSession();

public abstract ISessionForStoringTranslations CreateStoreTranslationSession();

public abstract void Dispose();

}

The *EngineBase* classinherits from the *IDisposable* interface. You need to implement this interface as well, and you should release the allocated resources during the dispose mechanism.

## The session for lookups component

This component is responsible for the translation (lookup). The naming convention is: *<plugin\_name>Session.cs*. The interface members are:

* *TranslateCorrectSegment* first overload: this function has three parameters; all of them are of type *MemoQ.Addins.Common.DataStructures.Segment*. The first segment is the translatable segment, and you can use the other two segments for the fuzzy correction. The function should return a *TranslationResult* object. This object’s members are:
  + *Translation*: should contain the translation as *Segment* object.
  + *Confidence*: returns the confidence of the translation between 0.0 and 1.0. If no confidence level available, should return 0.0.
  + *Info*: returns additional information about the translation, to be presented to the user (can be null).
  + *Exception*: if an exception occurred during translation, log the exception into this member.
* *TranslateCorrectSegment* second overload: this overload of the function also has three input parameters, but these are *segment arrays*, not segments. All arrays have the same size, and the function should return a result array of the same size.

The interface:

/// <summary>

/// Session that perform actual translation. Created on a segment-by-segment

/// basis, or once for batch operations.

/// </summary>

public interface ISession : IDisposable

{

/// <summary>

/// Translate segment, possibly using a fuzzy TM hit for improvement

/// </summary>

TranslationResult TranslateCorrectSegment(Segment segm,

Segment tmSource, Segment tmTarget);

/// <summary>

/// Translate a batch of segments, possibly using a fuzzy TM hit for improvement

/// </summary>

TranslationResult[] TranslateCorrectSegment(Segment[] segs,

Segment[] tmSources, Segment[] tmTargets);

}

Both functions should work with *Segment* objects. Use their *PlainText* property to get the actual segment’s content as a string, or work with any of the public methods available in this class.

The *ISession* interface inherits from the *IDisposable* interface. You need to implement this interface as well, and you should release the allocated resources during the dispose mechanism.

If an exception occurred during the translation you need to set the *Exception* member of the *TranslationResult* class. You need to use the *MTException* class to wrap the original exception.

The *TranslationResult* class is the following:

/// <summary>

/// One translated segment

/// </summary>

public class TranslationResult

{

/// <summary>

/// Translation

/// </summary>

public Segment Translation;

/// <summary>

/// Confidence of the translation between 0.0 and 1.0. If no

/// confidence level available, supply 0.0.

/// </summary>

public double Confidence;

/// <summary>

/// Additional info about the translation, to be presented to the user

/// (can be null)

/// </summary>

public string Info;

/// <summary>

/// If an exception occured during translation, then log the exception

/// into this member.

/// </summary>

public Exception Exception;

}

Return the translation result as a *Segment* object. To create *Segment* objects from plain text, use the *MemoQ.Addins.Common.DataStructures.SegmentBuilder* class (see the *DummyMTSession* class for more details).

The *MTException* class:

[Serializable]

public class MTException : UserException

{

public MTException(string message, string englishMessage,

Exception innerException = null)

: base(message, englishMessage, innerException)

{ }

public MTException(SerializationInfo info, StreamingContext context)

: base(info, context)

{ }

}

Use the first constructor to instantiate an *MTException*. It is important to fill the *message* parameter with localized text, because memoQ displays this message under the translation grid as the lookup error. See localization details later.

## Lookup session with extended parameters

In memoQ 9.14 and newer versions, extra information is available in lookup sessions for a better translation result. Until this version, when a segment was received via the SDK, the MT service knew nothing about the segment’s origin. So that MT providers can get smarter, memoQ’s MT SDK offers additional information (metadata) with the segment’s content. If an MT service can utilize metadata, its plugin should implement the *ISessionWithMetadata* interface. (NOTE: The implementation of this interface is optional: if the MT service cannot use metadata, you can safely ignore it.)

The newly added metadata contains 8 pieces of information in 2 groups. The first group contains project-level metadata provided by the user: project ID, client ID, plus the project's domain and subject. The second group is segment-level information: project GUID, document ID, segment ID, and segment status.

This is the new extended session interface:

public interface ISessionWithMetadata : ISession

    {

        /// <summary>

        /// Translate segment, possibly using project and segment level metadata for improvement

        /// </summary>

        TranslationResult TranslateCorrectSegment(Segment segm, Segment tmSource, Segment tmTarget, MTRequestMetadata metaData);

        /// <summary>

        /// Translate a batch of segments, possibly using project and segment level metadata for improvement

        /// </summary>

        TranslationResult[] TranslateCorrectSegment(Segment[] segs, Segment[] tmSources, Segment[] tmTargets, MTRequestMetadata metaData);

    }

This is the *MTRequestMetadata* class:

public class MTRequestMetadata

   {

       /// <summary>

       ///Content of "Project" field from the project creation form

       /// </summary>

       public string ProjectID { get; set; }

       /// <summary>

       ///Content of "Client" info filed from the project creation form

       /// </summary>

       public string Client { get; set; }

       /// <summary>

       ///Project's domain

       /// </summary>

       public string Domain { get; set; }

       /// <summary>

       ///Project's subject

       /// </summary>

       public string Subject { get; set; }

       /// <summary>

       ///Id value of the document where the segments to translate are from

       /// </summary>

       public Guid DocumentID { get; set; }

       /// <summary>

       /// Project’s GUID identifier

       /// </summary>

       public Guid ProjectGuid { get; set; }

       /// <summary>

       ///Metadata of translation segments

       /// </summary>

       public List<SegmentMetadata> SegmentLevelMetadata { get; set; }

   }

public class SegmentMetadata

     {

        /// <summary>

        /// Original segment's ID

        /// </summary>

        public Guid SegmentID { get; set; }

        /// <summary>

        /// Shows the status of the segment

        /// </summary>

        public ushort SegmentStatus { get; set; }

        /// <summary>

        /// SegmentIndex indicates the source segment's index of the current metadata, in the source segment list

        /// </summary>

        public int SegmentIndex { get; set; }

    }

Important notes!

* In the case of patched matches, we cannot talk about complete segments, only fragments: In this case we don't have real segment data either. This means that in such cases, the plugin will only receive the project-level info, the project GUID, and the document ID.
* If segments do not come from a standard translation document, but from a view, the document ID will correspond to the View ID. With this ID info we also want to indicate the connection between the segments.

## The session for storing translations component

Optional component. It is responsible for the storing finished translation units.

/// <summary>

/// Session that performs storing finished translations.

/// Created on a segment-by-segment basis, or once for batch operations.

/// </summary>

public interface ISessionForStoringTranslations : IDisposable

{

/// <summary>

/// Stores a finished translation unit.

/// </summary>

public StoreTranslation(TranslationUnit transunit);

/// <summary>

/// Stores a batch of finished translation units.

/// </summary>

/// <retuns>

/// The indices regarding the parameter array that were added succesfully.

/// </returns>

public[] StoreTranslation(TranstionUnit[] transunits);

}

The *TranslationUnit* class:

/// <summary>

/// Describes a translation unit to be stored by the MT plugin.

/// </summary>

public class TranslationUnit

{

/// <summary>

/// Translation

/// </summary>

public Segment Source;

/// <summary>

/// Translation

/// </summary>

public Segment Target;

}

## Plugin settings

You need to create a class to store the plugin’s settings. The naming convention is: *<plugin\_name>Options.cs*.

Note: Starting from memoQ 8.2, machine translation plugins no longer manage (store and load) their own settings. Instead, all plugin-related settings are stored in MT settings resources. All plugin settings must be XML serializable for memoQ to work with. The class(es) used for storing options must conform to XML serialization rules (public getter-setter properties, parameter-less constructor, avoiding unserializable data types such as Dictionary, etc).

The options have two distinct parts for storing general settings and secure settings (such as passwords). memoQ makes sure that secure settings are not stored as plain text. To facilitate this behavior, follow these steps:

* Create a class to store the general, non-secure settings. The naming convention is: *<plugin\_name>GeneralOptions.cs*
* Create a class to store the secure settings. The naming convention is: *<plugin\_name>SecureOptions.cs***.** Everything you store in there will be encrypted in the MT settings resource. This class is optional: if the machine translation plugin doesn’t have any sensitive settings (e.g.: API keys, passwords, etc.), this class can be omitted.
* Derive your original options class from *MTInterfaces.PluginSettingsObject*, and set the general and secure classes as type parameters.

When deriving from the base class, the plugin infrastructure takes care of serializing the settings. However, plugins are allowed to override the default serialization behavior in method *GetSerializedSettings* by providing a custom serialization.

## Migrating settings

When you are updating your legacy (pre-8.2) machine translation plugin, make sure to implement the *IPluginSettingsMigrator* interface to keep your old plugin settings. memoQ will automatically call the director’s *ReadSettingsFromFile* method, where you can load your existing options and create a new settings object.

The *IPluginSettingsMigrator* interface:

public interface IPluginSettingsMigrator

{

PluginSettings ReadSettingsFromFile(string pluginSettingsDirectory);

}

You may choose not to implement this interface. In this case, any existing configurations in previous memoQ versions will not automatically be migrated to memoQ 8.2 (or newer), and memoQ users will need to configure the plugin by hand.

## The configuration dialog

The plugin should have a configuration user interface, where the user will be able to set up the plugin. You need to create a dialog with the proper user interface elements. This dialog will be displayed by the plugin director’s *EditOptions* function. The naming convention is: *<plugin\_name>OptionsForm.cs*. The requirements are the following:

* This dialog should be initialized based on the existing plugin settings. If there are no saved settings yet, initialize the dialog with the default settings.
* Allow to save settings only if all mandatory parameters are configured correctly.
* If the user modifies the settings, collect the modifications in memory, and save them only when the user OKs the dialog.
* Do not call any long operation from the user interface thread. Do this in background threads.
* The configuration dialog may be displayed from a dedicated application domain. Generally, there are no specific actions to allow this, however, using non-standard practices in the user interface or in the code may prohibit this. Testing is advised.
* A Help button which is linked to the correct memoQ Help page is required for all public MT plugins.
* UI is displayed correctly at high DPI settings.
* Parameters related to secure settings are masked with char ‘\*’ (API key, password etc.).
* During credential data verification, or any other necessary interaction with the MT provider’s server, no or poor quality internet connection may cause problems. A recent improvement in memoQ (a uniform system for exception handling, including internet connection errors) allows global handling of the “no internet connection” issue for public plugins. In order to be processed in memoQ's uniform system, the exceptions thrown by the plugins need to be of the type *WebException*, and their status code needs to be either *WebExceptionStatus.ConnectFailure* or *WebExceptionStatus.NameResolutionFailure*.

## UI Design Guidelines

If you remember just one of these guidelines, it should be:  
  
**Do not reinvent the wheel!**

There’s a reason most apps are structured the way they are — because it works. Why? Because users learned it, and they are used to it. If they meet a new user interface (UI) component or something that behaves differently than in other apps, they will need to learn that new behavior. That takes unnecessary time and effort. It is better to use tried and tested layouts everywhere in the product, with minor tweaks to fit your goals. This way, users will be able to use your plugin easily, and without having to learn new layouts. So:

**Use standard Windows UI**

Use standard Windows UI components, and make the whole plugin look like Windows and memoQ. Your users will feel at home and will be able to use the plugin easily.

Here are some Microsoft resources on how to create an intuitive user interface and user experience in a Windows app:  
<https://docs.microsoft.com/en-us/windows/win32/uxguide/top-violations><https://docs.microsoft.com/en-us/windows/win32/appuistart/-user-interface-principles><https://docs.microsoft.com/en-us/windows/win32/uxguide/guidelines>

(We know that even memoQ does not follow all these resources, but we are working on it.)

We recommend that you read and follow all those guidelines. But here is a shorter list:

* Proximity  
  Just like in life, logically related components should be physically close together. But do not put components too close, always have some empty space between them.
* Alignment  
  Items should not be randomly placed: every component should have a visual bond with another one.
* Sizes  
  Make the same UI components the same height, e.g. all buttons should be the same height. If you are unsure about a component’s size, the easiest way is to measure one on a Windows dialog.
* Fonts  
  We recommend using "Segoe UI" 12 pt as a general font. Use semi-bold only for short headers and only if it's absolutely necessary to highlight those words. Avoid italic. All texts should be black, except links (or command links) which should be blue (#0000EE). Important error texts may be red (#EE0000), but do not overuse it.
* Icons and logos

If there is a well-known Windows icon for indicating something (e.g.: yellow warning sign, blue info dot etc.) then use those, do not create your own. For custom icons and logos, use a transparent background. They will look nicer and more professional.

* Scale to high DPI

Icons on the UI must scale. Please check 100, 125, 150, 200 and 300 scaling. For more information, refer to section [“Scale special UI elements to high DPI”](#_Scale_special_UI).

* Localization  
  Always remember that the UI will be localized to other languages as well. German or Spanish texts are usually longer than English. Generally, leave 50% more place for text, so that other languages fit in. For shorter text (1 or 2 words), leave even more space: sometimes, even 100% more space can be too short.
* Add tooltips  
  Tooltips are a simple but powerful way to give the user instant help. Use tooltips for as many UI components as you can. For complicated tasks, when the user does not know clearly what to do, use the well-known blue dot icon with an “i” to show users that there's some help here.
* Test the UI  
  Show it to your colleague or a friend who does not know the feature. Ask them how they think the components work. It’s a lot better than not testing at all and their answers may even surprise you.

When in doubt, ask us at [design@memoq.com](mailto:design@memoq.com). memoQ's Design Team is happy to help you if you need advice about your plugin’s user interface.

## Localization

The third-party machine translation plugins will be localized by the memoQ Ltd. The *IEnvironment* interface provides the *GetResourceString* function for the developers to be able to get localized texts from the machine translation environment.

All textual information appearing on the graphical user interface should be localized. Therefore, the plugin’s developer must provide the list of these strings for memoQ Ltd. This list should contain key-value pairs. The key must uniquely identify the string value. You will be able to use these localized texts inside your plugin using the *GetResorceString* function – simply pass the required text’s key to the function. Apart from this, the function has another parameter, *pluginId*. This parameter should be the machine translation plugin’s unique ID. Place this identifier as a public constant into the *PluginDirector* class.

It is possible that the *GetResourceString* function gives back the searched resource key in the form of “MTPlugin.<PluginId>.<Key>” (e.g.: MTPlugin.MyPlugin.ErrorMsg). In this case the plugin should use its own default strings.

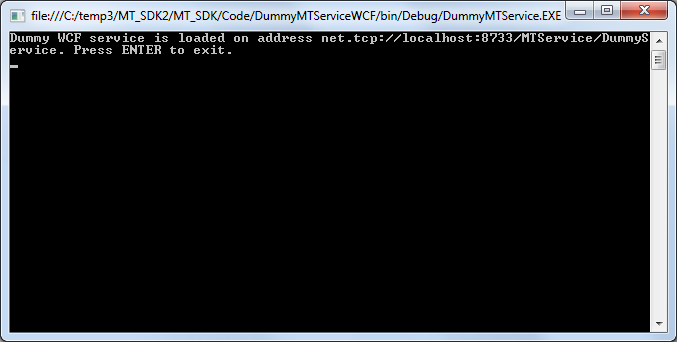
# Implementation checklist

If you are done with the machine translation plugin’s implementation, you need to check that:

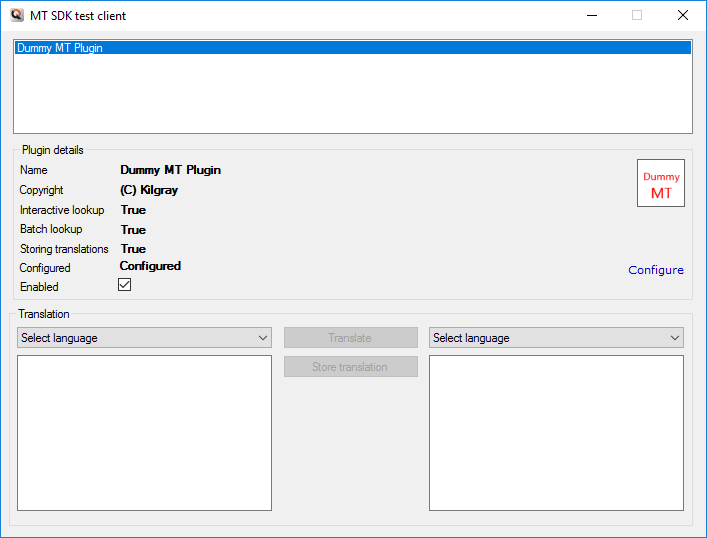
* The implementation is in a single class library, which contains references to the necessary memoQ libraries. The class library is written in C#.
* All source code text added during implementation (comments, naming of functions, variables, classes, etc.) is in English.
* No package references are used, except packages from the following list: *MemoQ.Addins.Common, Kilgray.Utils, MemoQ.MTInterfaces, Newtonsoft.Json, Microsoft.IdentityModel.Tokens, System.IdentityModel.Tokens.Jwt.*
* The class library’s *AssemblyInfo.cs* contains the *ModuleAttribute* attribute.
* There is a plugin director component which properly implements the *IModule* interface and derives from the *PluginDirectorBase* class.
* All allocated resources are properly disposed in the plugin director.
* There is an engine component which properly implements the *EngineBase* interface.
* All allocated resources are disposed correctly in the engine.
* There is a session component which properly implements the *ISession* interface.
* The *MTException* class is used to wrap the original exceptions occurred during translation.
* All allocated resources are disposed correctly in the session.
* There is an options class with proper *generic* and *secure* subclasses (the secure options class can be omitted).
* The options class is a simple entity class, does not call any services, and simply returns the saved or the default settings.
* The options class does not store and load its own settings.
* There is a configuration dialog where the user can configure the plugin.
* The user can only save the settings when all mandatory parameters are configured correctly.
* The dialog collects the user’s changes in memory, and saves only when the user OKs the dialog.
* The dialog does not call any blocking service or lengthy operation in the user interface thread; it must use background threads.
* The translation service is only called during configuration and translation. Everywhere else use the stored plugin settings to return plugin information (for example, the plugin’s supported languages).
* All UI is displayed correctly at high DPI settings.
* Parameters related to secure settings (API key, password etc.) are masked with the asterisk ‘\*’ character on the settings UI.

# Testing the sample plugin

To test the sample plugin: Open the *MT\_SDK* solution in Visual Studio, and set the *TestClient* and *DummyMTService* projects as startup projects, and start debugging. The *DummyMTService* runs as a console application and emulates an MT service:

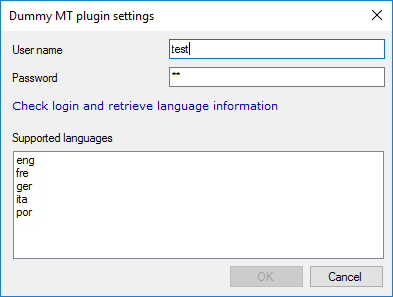


The *TestClient* emulates memoQ – it loads and uses the MT plugins:



Currently there is only one MT plugin registered. The dummy plugin’s properties are in the *Plugin details* section. To be able to translate texts with the plugin, you need configure it first. Click the *Configure* link.

The dialog allows you to set up the plugin. Type something into the *User name* field, something else into the *Password* field and click the *Check login and retrieve language information* link. An error dialog appears, because the sample plugin allows logging in only if the username and the password are the same. Now enter the same string into the *User name* and *Password* fields, and click the *Check login* link again. Now the supported languages appear:

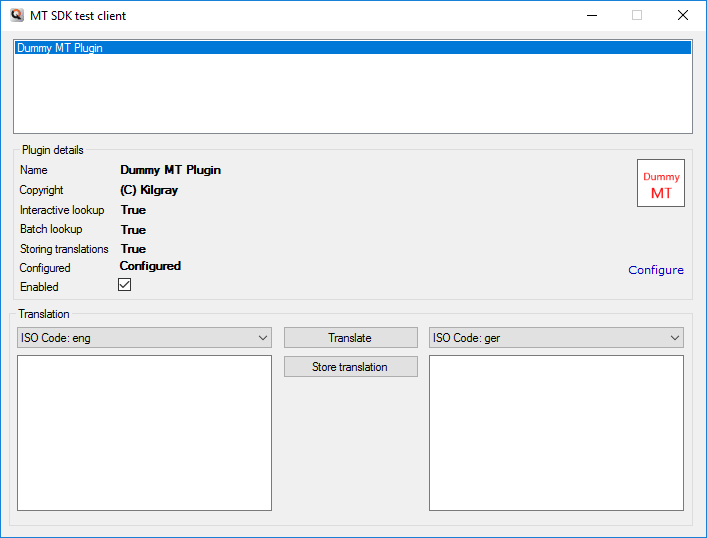


Click the *OK* button and enable the plugin inside the *Plugin details* section.

To translate something, select a language pair. If you select one which is not supported by the plugin, you will get the error message “This language pair is not supported by the selected plugin.”

Select a supported language pair and enter something into the source text box and click the *Translate* button. The translation will appear after a few seconds in the target text box.

If you enter more than one line into the source text box, the plugin performs batch translation. Otherwise, it does a simple translation.



# Testing the new plugins

## Testing in the sample application

If you would like to test your machine translation plugin, you need to add your project as project reference to the *MT\_SDK* project. After that, you need to extend the constructor of the *MainForm* class. Insert the following line after the “add other plugin directors” comment (instantiate your own plugin director instead of the *DummyMTPluginDirector*):

plugins.Add(PluginInfoFactory.Create(new DummyMTPluginDirector()));

If the plugin is implemented correctly, it will be listed on the sample application’s main form. If you select the plugin from the list, you can see its general information in the *“Plugin details”* box. If you click the *“Configure”* link, you can set up your plugin. You will be able to test the translation if the plugin is configured and enabled. Select the source and the target languages (if you choose an unsupported language pair, a red message appears between the two text boxes), and enter something into the left-side text box. If the text box contains multiline text, the batch translation will be called. If there was any exception during the translation a message box will appear.

## Testing a new plugin in memoQ

You can also test your MT plugin in memoQ (version 7.8.55 and newer). First, copy your plugin’s .dll file into the *Addins* folder under memoQ’s installation folder. By default, memoQ requires confirmation at startup to load unsigned plugins. To enable loading your plugin automatically, you need 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.

## Testing a new plugin in memoQ server

You can also test your MT plugin in the memoQ server. First, copy your plugin dll file into the *Addins* folder under memoQ server’s installation folder – just like with memoQ. To load your unsigned plugin automatically, add the .dll plugin’s filename (without the file extension) to the file *UserApprovedUnsignedMTplugins.xml* in the *%PROGRAMDATA%\MemoQ Server* folder, and restart memoQ server.

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

<ApprovedUnsignedMTPluginsCatalog xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<ApprovedUnsignedMTPlugins>

<Plugin>

<Name>MemoQ.ExampleMT</Name>

</Plugin>

<Plugin>

<Name>MemoQ.TestMT</Name>

</Plugin>

</ApprovedUnsignedMTPlugins>

</ApprovedUnsignedMTPluginsCatalog>

# Plugin supported by memoQ Server

memoQ server 8.2 and newer supports Machine Translation plugins as parts of the project through MT settings resources. The plugin architecture is built such that if a plugin is installed in a memoQ server, memoQ clients can use it without having to install the plugin locally. This allows central management and configuration of both the plugins and their settings –users have no access to password and other sensitive information required to use the MT service, but they can still perform lookups).

memoQ server will not load legacy plugins, only plugins that conform to the checklist below.

Plugin developers should be aware that configuring the plugin settings is done on memoQ’s user interface – even if the plugin is not installed locally. To show the configuration user interface, memoQ downloads the plugin’s dll from memoQ server. (The plugin dll is then discarded; it is never written to disk.) If a plugin is built with external dependencies, it still must be able to show the configuration user interface without those external dependencies.

# Checklist to update a plugin for memoQ 8.2

To make a legacy (version 8.0) plugin and its codebase fully compatible with memoQ version 8.2 or higher, update the library by going through these steps.

* Update the implementation class of the *IPluginDirector2* interface.
  + Do not use this interface directly, derive from *PluginDirectorBase* instead.
  + Override the necessary methods and fields.
* Remove the *IModuleEx* implementation from the director altogether.
* Update the implementation class of the *IEngine2* interface.
  + Do not use this interface directly, derive from *EngineBase* instead.
  + Override the necessary methods and fields.
* Update the plugin's option class.
  + Do not use static fields and methods to access the options instance.
  + Instead, pass an option object wherever it's needed.
* Create two new options classes: one for the general and one for the secure settings. (The secure settings class is optional.)
  + Your original options class should derive from *PluginSettingsObject* and you should pass the generic and secure classes as the type parameters.
  + Create two constructors for the options class: one with a *PluginSettings* parameter and one with the two general and secure settings parameters. Make sure to pass these parameters to the base class.
  + Move your existing options fields from the original options class into the correct classes.
  + Update your plugin’s code to access these fields through the general and the secure settings classes.
* If you wish to keep your old plugin settings by migrating them into an MT settings resource, then the director class should implement the *IPluginSettingsMigrator* interface.