memoQ

Machine Translation SDK

KILGRAY
Translation Technologies

© 2004-2017 Kilgray Translation Technologies.
All rights reserved.
www.memoQ.com

Contents

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

Overview

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

The current documentation describes the MT SDK supported by memoQ from version 8.2. To develop plugins for previous versions of memoQ, please refer to the documentation of the appropriate version. Existing plugins remain compatible – however, those plugins cannot be used on memoQ servers and they also have limitations with regards to their usage in memoQ.

These plugins have to be developed for **.NET Framework 4.6.1** in **C# language**.

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

The workflow for creating and distributing a plugin

Assuming CompanyA wants to create a new MT plugin, the steps for the recommended workflow are the following:

- CompanyA develops a new MT plugin using the MT SDK using C# language.
- CompanyA sends the source code of the plugin to Kilgray.
- Kilgray reviews the code and tests the functionality of the plugin. CompanyA performs fixes based on the review if required.
- Kilgray compiles the source code of the MT plugin, signs the resulting DLL with its private key (unsigned plugins require confirmation from the user every time they are loaded), makes it part of the memoQ client installer. As a result, the MT plugin is distributed with the memoQ client installer from this point.
- The source code of the MT plugin becomes part of the memoQ code base at Kilgray.
- Information about bugs reported by customers are forwarded to CompanyA by Kilgray. CompanyA is responsible for fixings the bugs; bug fixes are reviewed by Kilgray.

This workflow is required to ensure that plugins meet the quality requirements of memoQ and do not jeopardize the stability of the entire product.

If your company does not want to distribute the MT plugin (only wants to use it internally), it is to be handled based on a different workflow. Please contact to Kilgray to agree on the details.

Machine translation framework in memoQ

The machine translation framework provides the possibility to use external translation services from memoQ. Kilgray delivers several built-in machine translation plugins with the memoQ client (such as Google MT, Let's MT, Microsoft MT, etc.), but companies also have the possibility to write brand new machine translation plugins.

Machine translation plugins

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

- MemoQ.Addins.Common.dll
- MemoQ.MTInterfaces.dll

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

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

These libraries contain all of the necessary classes for the plugins. **The use of any other external libraries is not allowed in machine translation plugins.** If it is required absolutely necessary consult it with the Kilgray.

Machine translation interfaces

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

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

Moreover, the plugins should also derive from the following base classes:

- MemoQ.MTInterfaces.PluginDirectorBase
- MemoQ.MTInterfaces.EngineBase

The machine translation plugins may also implement the following, 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, cleanup the modules and to be able to get general information about the modules.

ISession interface

memoQ calls the object implementing this interface to perform a lookup. A new session object is created on 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 has to communicate with the plugin.

EngineBase class

An object deriving from the EngineBase class is requested by memoQ for a particular language combination with the help of the plugin director.

Optional: ISessionForStoringTranslations interface

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

Optional: IPluginSettingsMigrator interface

From memoQ 8.2, machine translation plugins are no longer responsible for storing their own settings. Instead, the plugin-related settings are stored in MT light resources. When you're upgrading your machine translation plugin to support version 8.2, make sure to implement the IPluginSettingsMigrator interface, as it makes possible to migrate all your old plugin settings into a new resource file.

Machine translation SDK sample application

Kilgray 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. 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 4.6.1. 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 name of the module and the plugin director class):

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

memoQ will check this attribute when it loads the machine translation assemblies. The module name field should be the name of the machine translation plugin and the class name should be the name of the plugin director class.

Now you have to set up the memoQ library references. The necessary DLLs are under the *References* folder.

The plugin director

This component is the entry point of the plugin. First of all you have 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 members

This interface has two functions and one property:

- *Cleanup* function: you can implement the plugin's cleanup logic here.
- *Initialize* function: you can implement the plugin's initialization logic here.
- *IsActivated* property: you can tell here whether the plugin is activated or not.

The interface is the following:

```
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 members

This class has seven properties and three functions:

- *BatchSupported* property: you can tell here whether the plugin supports the batch translation (lookup). memoQ uses batch translation during the pre-translate operations.
- *CopyrightText* property: you should return the plugin's copyright information here; it will be displayed on the user interface where memoQ lists the available plugins.
- *DisplayIcon* property: you should return here the icon of the MT plugin. This image will be displayed on the user interface where memoQ lists the available plugins.
- *Environment* property: you can use some basic services with the help of this property. The members of the IEnvironment interface are the following:
    - *UILang* property: gives back the two-letter language code of the memoQ's user interface.
    - *ParseTMXSeg* function: this function has one string input parameter. You can send here a segment in TMX format, and the function gives back the proper memoQ *Segment*.
    - *PluginAvailabilityChanged* function: you should call this function if the availability of your plugin has been changed and you would like to signal it.
    - *WriteTMXSegment* function: this function has one input parameter which is a memoQ *Segment*, and the function converts this segment into TMX format. Please note that the value of the translatable attributes of the segments will not be written into the TMX. Because of this, if you would like to keep this information intact, you have to restore the original attribute after the TMX round trip.
    - *GetResourceString* function: this function has one string input parameter and it returns the localized text belonging to the key specified in the parameter.
    - *BuildWordsOfSegment* function: tokenizes a Segment on whitespace and word boundaries.
- *FriendlyName* property: you should get back the plugin's human-readable name. It will be displayed on the user interface where memoQ lists the available plugins.

- *InteractiveSupported* property: you can tell here whether the plugin supports interactive translation or not. memoQ uses this information when the user works in the translation grid, and the memoQ tries to get back translation hits from the machine translation plugin.
- *PluginID* property: you have to get back the plugin's identifier here.
- StoringTranslationSupported property: you can tell here whether the plugin supports the adaptive behavior.
- *CreateEngine* function: this function has two input parameters, the source, and the target language. Based on these languages you should instantiate and give back a machine translation engine here.
- *IsLanguagePairSupported* function: you have to give back whether the plugin supports the given language pair or not. Do not call any service here, give back the result based on the saved plugin settings.
- *EditOptions* function: memoQ calls this function when the user would like to configure your machine translation plugin. In this function, you should display the configuration dialog of the plugin.

The class is the following:

```
/// <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

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

The engine component should derive from the the EngineBase class. The naming convention is: **<plugin_name>Engine.cs**. The class has the following members:

- *SmallIcon* property: memoQ displays this icon under translation results when an MT hit is selected from this plugin.

- *SupportsFuzzyCorrection* property: you can tell here whether the engine supports the adjustment of fuzzy TM hits through machine translation.
- *SetProperty* function: this functions sets an engine-specific property, for example, subject matter area.
- *CreateLookupSession* function: memoQ calls this function to be able to perform the translations. You should instantiate and return a session object here. This session will not be used in 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 multi-threaded way.

The class is the following:

```
/// <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 class* inherits from the *IDisposable* interface. You have 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 the following:

- *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:
  - *Translation*: this member should contain the translation as *Segment* object.
  - *Confidence*: you can give back the confidence of the translation between 0.0 and 1.0. If no confidence level available, supply 0.0.
  - *Info*: you can send back additional information about the translation, to be presented to the user (it can be null).
  - *Exception*: if an exception occurred during translation, then log the exception into this member.
- *TranslateCorrectSegment* second overload: this overload of the function has three input parameters as well. But it has three segment arrays instead of three segments. All arrays have the size, and the function should give back a result array of the same size.

The interface is the following:

```
/// <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. You can use the *PlainText* property of them to get the content of the actual segment as a string or work with any of the public methods available in this class.

The *ISession* interface inherits from the *IDisposable* interface. You have 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 have to set the Exception member of the *TranslationResult* class. You have 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;
}
```

You should give back the result of the translation as a *Segment* object. Use the MemoQ.Addins.Common.DataStructures.*SegmentBuilder* class to create *Segment* objects from plaintext (see the *DummyMTSession* class for more details).

The *MTException* class is the following:

```
[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)
    { }
}
```

You can 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 lookup error. See localization details later.

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 is the following:

```
/// <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;
}
```

The plugin options

You have to create a class to store the settings of the plugin. The naming convention is: **<plugin_name>Options.cs**.

Please note: starting from memoQ 8.2, the machine translation plugins are no longer responsible for managing (storing and loading) their own settings. Instead, all the plugin-related settings are stored in MT light resources. All plugin settings must be xml serializable for memoQ to work with; the class(es) used for storing options must adhere to xml serialization rules (public getter-setter properties, parameterless 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 when handling the secure settings that the contents are not stored as plain text. To facilitate this behavior, follow the steps below:

- 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 light 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 options

When you are updating your existing (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 options object.

The IPluginSettingsMigrator interface is the following:

```
public interface IPluginSettingsMigrator
{
    PluginSettings ReadSettingsFromFile(string pluginSettingsDirectory);
}
```

The plugin may chose not to implement this interface, in which case any existing configurations available in previous versions of memoQ will not automatically be migrated in memoQ 8.2 and the memoQ user will have 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 have 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 to the default settings.
- Do not allow to save the settings until all of the mandatory parameters were not configured correctly.
- If the user modifies the settings, collect the modifications in the memory, and save them only when the user OKs the dialog.
- Do not call any long operations from the user interface thread. Do this call 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.

Localization

The third-party machine translation plugins will be localized by the Kilgray. 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 which appear on the graphical user interface should be localized. Therefore the developer of the plugin has to provide the list of these strings for Kilgray. This list should contain key-value pairs. The key has to uniquely identify the string value. You will be able to use this localized texts inside your plugin with the help of the *GetResorceString* function by simply pass the key of the required text to the function. Apart from this, the function has another parameter which is the *pluginId*. This parameter should be the unique identifier of the machine translation plugin. It's recommended to place this identifier as a public constant into the *PluginDirector* class.

It is possible that the *GetResourceString* function gives back null or an empty string. In this case the plugin should use its own default strings.

Implementation checklist

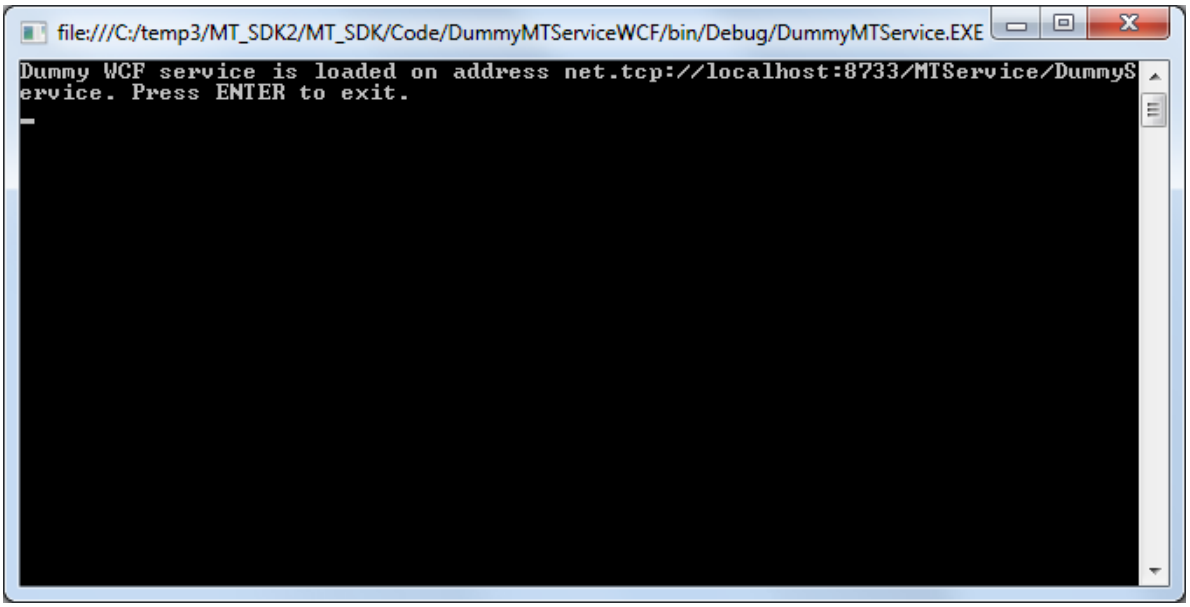
If you are done with the implementation of the machine translation plugin, you have to check:

- The implementation is in a single class library, which contains references to the necessary memoQ libraries. The class library is written in C#.
- 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 the 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 gives back the saved or the default settings.
- The options class does not store/load its own settings.
- There is a configuration dialog, where the user is able to configure the plugin.

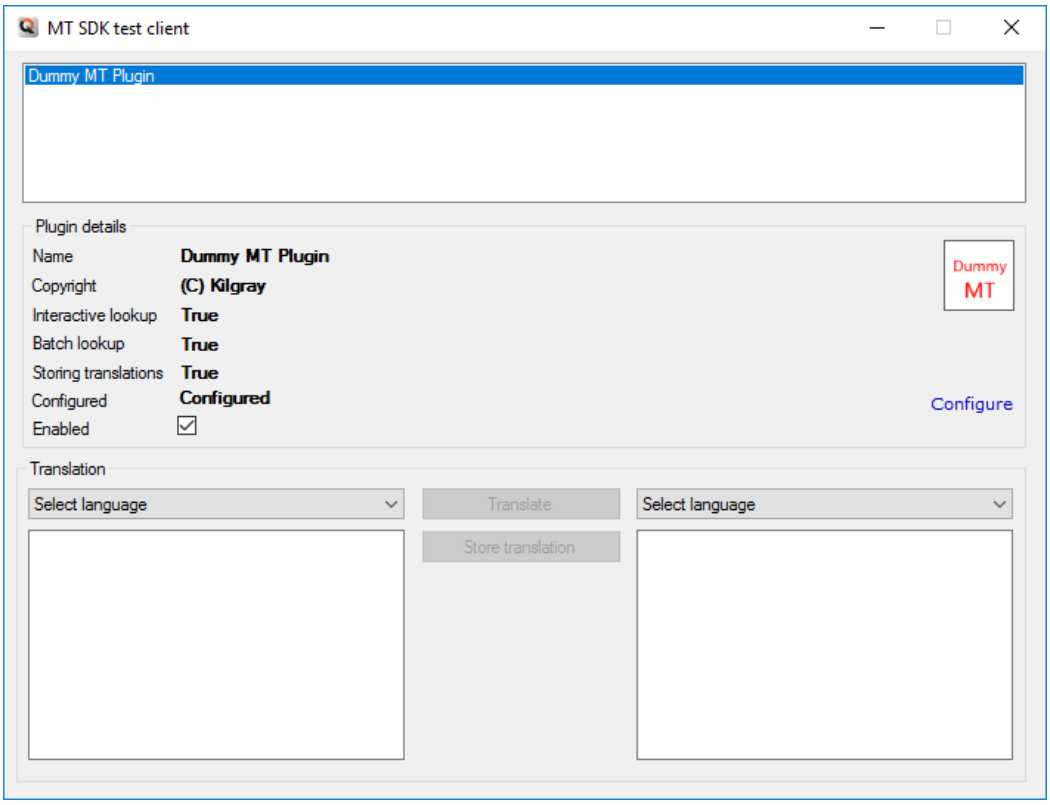
- The user cannot save the settings until all of the mandatory parameters were not configured correctly.
- The dialog collects the user modifications in the memory and saves only when the user OKs the dialog.
- The dialog does not call any blocking service in the user interface thread; it has to use background threads.
- Call the translation service only during the configuration process or during the translation. In all other cases use the stored plugin settings to give back the plugin information (for example the supported languages of the plugin).

Testing the sample plugin

If you would like to test the existing sample plugin open the *MT_SDK* solution in Visual Studio, and **set the *TestClient* and DummyMTService projects as startup projects** (multiple startup projects are to be set) and start debugging. The DummyMTService runs as a console application and emulates an MT service:

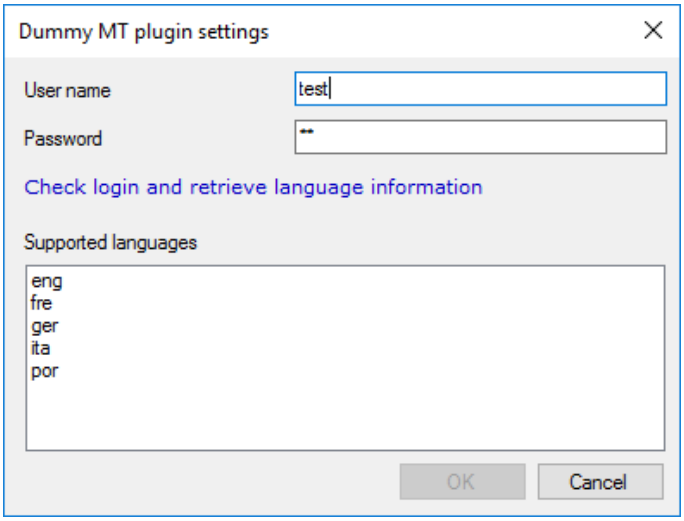


The TestClient emulates the memoQ client: it loads and uses the MT plugins:



Currently there is only one MT plugin registered. The properties of the dummy plugin are in the *Plugin details* section. To be able to translate texts with the plugin, configure it, because the plugin is not configured yet. Click on the *Configure* link.

The dialog allows you to setup the plugin. Write something into the *User name* field, write something else into the *Password* field and click on the *Check login and retrieve language information*. In this case, an error dialog appears because the sample plugin allows the logins only if the username and the password are the same. Now try to write the same strings into the username and the password fields, and click on the check link again. Now the supported languages will appear:

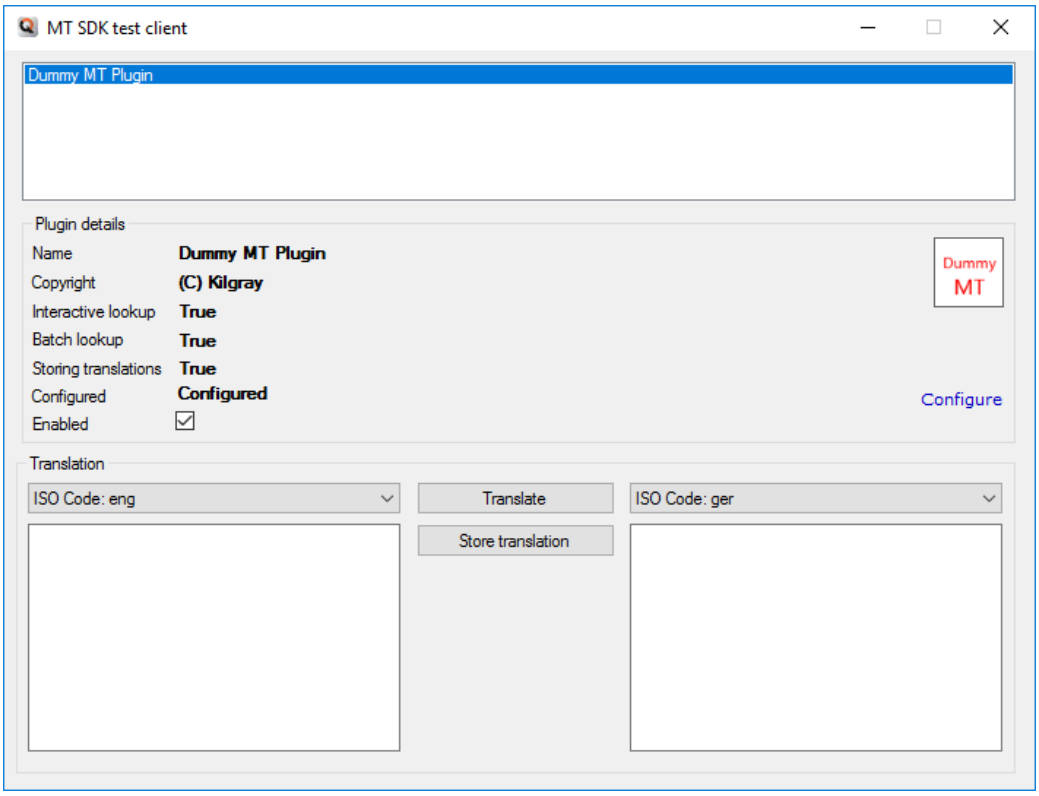


Click on the *OK* button and enable the plugin inside the *Plugin details* section.

Select a language pair to be able to translate something. If you select a language pair which is not supported by the plugin, you will get the following error message: “This language pair is not supported by the selected plugin.”

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

A batch translation function will be performed if you write more than one line into the source text box. Otherwise, a simple translation will be performed.



Testing the new plugins

Testing in the sample application

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

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

If the plugin is implemented correctly your plugin will be listed on the main form of the sample application. If you select the plugin from the list, you can see the general information of your plugin in the *“Plugin details”* box. If you click on the *“Configure”* link, you can set up your plugin. You will be able to test the translation if the plugin is configured and it is enabled. Select the source and the target languages (if you choose a language pair which is not supported by your plugin, a red message will appear between the two text boxes), and write something into the left-side textbox. If the

textbox contains multiline text, the batch translation will be called. If there was any exception during the translation a message box will appear.

Testing in memoQ client

You can also test your MT 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 requires confirmation at startup to load unsigned plugins. To enable loading your plugin automatically 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.

Plugin supported by memoQ Server

memoQ server as of version 8.2 supports Machine Translation plugins as part of the project through MT light resources. The plugin architecture is built such that a plugin may be used by memoQ clients without having to install the plugin to the clients if the plugin is installed in the server. This allows central management and configuration of both plugins and the settings of individual plugins (memoQ client users have no access to password and other sensitive information required to use the MT system, but they are still able to issue lookups).

Existing plugins will not be loaded by memoQ Server. Only plugins that adhere to the checklist below are loaded by memoQ Server.

A plugin developer should be aware that configuring the plugin settings is carried out through the user interface of memoQ client even if memoQ client does not have the plugin installed. This is achieved by the client downloading the plugin dll from the server to show the configuration user interface. (The plugin dll is then discarded; it is never written to disk.) If a plugin is built such that it has external dependencies, it still must be able to work without its external dependencies for showing the configuration user interface.

Checklist to update a plugin for memoQ 8.2

Given an existing 8.0 plugin and its codebase the following steps describe the process to update the library to make it fully compatible with memoQ 8.2

- 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, it can be omitted everywhere.

    - 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 light resource, then the director class should implement the IPluginSettingsMigrator interface.