Terminology SDK

Contents

[Terminology SDK 1](#_Toc461619342)

[Versions 2](#_Toc461619343)

[Overview 2](#_Toc461619344)

[The workflow for creating and distributing a plugin 2](#_Toc461619345)

[Terminology framework in memoQ 3](#_Toc461619346)

[Terminology plugins 3](#_Toc461619347)

[Base classes and interfaces 3](#_Toc461619348)

[PluginDirectorBase 3](#_Toc461619349)

[IModule and IModuleEx interfaces 3](#_Toc461619350)

[IPluginDirector interface 3](#_Toc461619351)

[EngineBase 3](#_Toc461619352)

[IEngine 3](#_Toc461619353)

[SessionBase 3](#_Toc461619354)

[ISession 3](#_Toc461619355)

[Implementation steps of a TB plugin 4](#_Toc461619356)

[Create the new class library 4](#_Toc461619357)

[The plugin director 4](#_Toc461619358)

[PluginDirectorBase members 4](#_Toc461619359)

[The engine component 7](#_Toc461619360)

[Engine members 7](#_Toc461619361)

[The session component 7](#_Toc461619362)

[Engine members 7](#_Toc461619363)

[The plugin options 10](#_Toc461619364)

[The configuration dialog 10](#_Toc461619365)

[Localization 10](#_Toc461619366)

[Implementation checklist 11](#_Toc461619367)

[Testing the new plugins 11](#_Toc461619368)

[Checklist to update a plugin for memoQ 8.0 11](#_Toc461619369)

# Versions

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Who | Change |
| June 1, 2016 | 1.0 | NG | Initial version |
| Sept 13, 2016 | 1.2 | NG | Hit coloring (supported from memoQ version 7.8.158). |
| Sept 14, 2016 | 2.0 | DÁ | Changes in memoQ 8.0 |

# Overview

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

Creating plugins using the SDK is supported from memoQ version 7.8.154.

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

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

The TB SDK has a Visual Studio solution, which can be opened by Visual Studio 2013 or higher.

# The workflow for creating and distributing a plugin

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

* CompanyA develops a new TB plugin using the TB 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 TB plugin, signs the resulting DLL with its private key (unsigned plugins are not loaded by memoQ!), makes it part of the memoQ client installer. As a result, the TB plugin is distributed with the memoQ client installer from this point.
* The source code of the TB 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 TB 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.

# Terminology framework in memoQ

The terminology framework provides the possibility to use external terminology services from memoQ. Kilgray delivers several built-in term plugins with the memoQ client (EuroTermBank, TaaS client), but companies also have the possibility to write brand new terminology plugins.

## Terminology plugins

Every terminology plugin should be a standalone .NET DLL, which has the following references:

* MemoQ.Addins.Common.dll
* MemoQ.MBInterfaces.dll

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

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

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

## Base classes and interfaces

The memoQ application and the plugins can communicate with the help of a few base classes and interfaces. Every terminology plugin should use the following abstract classes:

* PluginDirectorBase
* EngineBase
* SessionBase

### PluginDirectorBase

This is the base class of plugin directors. Each plugin director represents an entry point of a plugin. This class implements *IModule*, *IModuleEx* and *IPluginDirector* interfaces.

### IModule and IModuleEx interfaces

memoQ manages the MT, TM and TB plugins as individual modules. These interfaces 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.

### IPluginDirector interface

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

### EngineBase

This is the base class of plugin engines. This class implements *IEngine* interface.

### IEngine

An object implementing the *IEngine* interface is requested by memoQ for a particular language combination at project open with the help of the plugin director.

### SessionBase

This is the base class of plugin sessions. This class implements *ISession* interface.

### ISession

memoQ calls the object implementing this interface to perform the translation. A new session object is created on lookup. *ISession* objects are always created by *IEngine* objects.

# Implementation steps of a TB 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 project and the type of the project should be Class Library. Being done with the project creation mark the assembly with the *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 TB", ClassName = "DummyTBPlugin.DummyTBPluginDirector")]

This attribute can be found in the *MemoQ.Addins.Common.Framework* namespace.

memoQ will check this attribute when it loads the terminology assemblies. The module name field should be the name of the terminology 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 **public** class inside the project. The naming convention is: **<plugin\_name>PluginDirector.cs**

This class should derive from the *PluginDirectorBase* class which can be found in the *MemoQ.TBInterfaces* namespace.

### PluginDirectorBase members

The class members:

* *environment* variable: the memoQ's application environment; e.g., to provide UI language settings etc. to the plugin.
* *IsActivated* property: you can tell here whether the plugin is activated or not.
* *Initialize* function: you can implement the plugin’s initialization logic here.
* *Cleanup* function: you can implement the plugin’s cleanup logic here.
* *PluginConfigured* property: you should return true here if the plugin has been set up correctly (e.g. account details, supported languages)
* *PluginEnabled* property: this property tells whether the plugin is enabled or disabled. The plugin should store this information among its own settings.
* *Environment* property: sets memoQ's application environment; e.g., to provide UI language settings etc. to the plugin.
* *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.
* *DisplayIcon* property: you should return here the icon of the TB plugin. This image will be displayed on the user interface where memoQ lists the available plugins.
* *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.
* *PluginID* property: you have to get back the plugin’s identifier here.
* *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.
* *ShowOptionsForm* function: memoQ calls this function when the user would like to configure your terminology plugin. In this function you should display the configuration dialog of the plugin.
* *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 terminology engine here.
* *SupportsAddingNewTerms* property: you can tell here whether the plugin supports adding new terms or not.
* *GetAddTermsUrl* function: you can tell the url where the user will be able to add new terms. This will be called only if the *SupportsAddingNewTerm* property is true.
* *SupportsModifyingExistingTerms* property: you can tell here whether the plugin supports modifying existing terms or not.
* *GetModifyTermsUrl* function: you can tell the url where the user will be able to modify existing terms. This will be called only if the *SupportsModifyingExistingTerms* property is true.

The class is the following:

/// <summary>

/// Abstract base class of the TB plugin directors.

/// </summary>

public abstract class PluginDirectorBase : IModule, IModuleEx, IPluginDirector

{

/// <summary>

/// The memoQ's application environment; e.g., to provide UI language settings

/// etc. to the plugin.

/// </summary>

protected IEnvironment environment;

/// <summary>

/// Gets whether the plugin is activated.

/// </summary>

public abstract bool IsActivated { get; }

/// <summary>

/// Initializes the plugin.

/// </summary>

public abstract void Initialize();

/// <summary>

/// Cleans up the resources used by the plugin if needed.

/// </summary>

public abstract void Cleanup();

/// <summary>

/// Gets whether the plugin is configured.

/// </summary>

public abstract bool PluginConfigured { get; }

/// <summary>

/// Gets or sets whether the plugin is enabled.

/// </summary>

public abstract bool PluginEnabled { get; set; }

/// <summary>

/// Sets memoQ's application environment; e.g., to provide UI language

/// settings etc. to the plugin.

/// </summary>

public virtual IEnvironment Environment

{

set { environment = value; }

}

/// <summary>

/// Gets if the plugin supports the provided language combination. The strings

/// provided are memoQ languages codes.

/// </summary>

public abstract bool IsLanguagePairSupported(string srcLangName,

string trgLangName);

/// <summary>

/// Gets a 48x48 display icon to show in memoQ's Tools / Options. Black is the

/// transparent color.

/// </summary>

public abstract Image DisplayIcon { get; }

/// <summary>

/// Gets the friendly name to show in memoQ's Tools / Options.

/// </summary>

public abstract string FriendlyName { get; }

/// <summary>

/// Gets the plugin's non-localized name.

/// </summary>

public abstract string PluginID { get; }

/// <summary>

/// Gets the copyright text to show in memoQ's Tools / Options.

/// </summary>

public abstract string CopyrightText { get; }

/// <summary>

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

/// </summary>

public abstract void ShowOptionsForm(Form parentForm);

/// <summary>

/// Create an TB engine for the supplied language pair.

/// </summary>

public abstract IEngine CreateEngine(string srcLangName, string trgLangName);

/// <summary>

/// Indicates whether the plugin supports adding new terms.

/// </summary>

public abstract bool SupportsAddingNewTerms { get; }

/// <summary>

/// Gets the url of the add terms page. Called only if SupportsAddingNewTerms

/// is true.

/// </summary>

public virtual string GetAddTermsUrl(string externalId, string sourceLang,

string sourceTerm, string targetLang, string targetTerm)

/// <summary>

/// Indicates whether the plugin supports modifying existing terms.

/// </summary>

public abstract bool SupportsModifyingExistingTerms { get; }

/// <summary>

/// Gets the url of the modify terms page. Called only if

/// SupportsModifyingExistingTerms is true.

/// </summary>

public virtual string GetModifyTermsUrl(string externalId)

}

## The engine component

The memoQ calls the plugin director’s *CreateEngine* function to get back a terminology engine for a language pair. memoQ uses this engine to perform the lookups. The engine component should derive from the *EngineBase* class which can be found in the *MemoQ.TBInterfaces* namespace. The naming convention is: **<plugin\_name>Engine.cs**.

### Engine members

The class members:

* *CreateSession* function: memoQ calls this function to be able to perform the lookup. You should instantiate and return a session object here. This session will not be used in multi-threaded way.
* *SetProperty* function: this function sets an engine-specific property, for example subject matter area.
* *Dispose* function: in this function you should release the allocated resources.

The class is the following:

/// <summary>

/// Abstract base class of the TB plugin engines.

/// </summary>

public abstract class EngineBase : IEngine

{

/// <summary>

/// Creates a session for translating terms. Session will not be used in

/// a multi-threaded way.

/// </summary>

public abstract ISession CreateSession();

/// <summary>

/// Sets an engine-specific custom property, e.g., subject matter area.

/// </summary>

public virtual void SetProperty(string name, string value)

{ }

/// <summary>

/// Disposes the resources used by the engine if needed.

/// </summary>

public abstract void Dispose();

}

## The session component

This component is responsible for lookup. It should derive from the *SessionBase* class. The naming convention is: **<plugin\_name>Session.cs**.

### Engine members

The class members:

* *Lookup* function: the function should return an array of *TerminologyResult* objects.
* *Dispose* function: in this function you should release the allocated resources.

The class is the following:

/// <summary>

/// Abstract base class of the TB plugin sessions.

/// </summary>

public abstract class SessionBase : ISession

{

/// <summary>

/// Translate the terms of the given segment.

/// </summary>>

public abstract TerminologyResult[] Lookup(Segment segment);

/// <summary>

/// Disposes the resources used by the engine if needed.

/// </summary>

public abstract void Dispose();

}

The *Lookup* function works with MemoQ.Addins.Common.DataStructures.*Segment* objects. You can should use the *IEnvironment.BuildWordsOfSegment* helper function get the words of the actual segment. The result should contain, among other items, a Segment object. Use the MemoQ.Addins.Common.DataStructures.*SegmentBuilder* class to create *Segment* objects.

If any exception occurred during the terminology lookups you have to throw a *TBAddinException* object. You have to use the *TBAddinException* class to wrap the original exception.

The *TerminologyResult* class is the following:

/// <summary>

/// This class represents one translated term.

/// </summary>

public class TerminologyResult

{

/// <summary>

/// The external identifier (used by the TB service) of the hit.

/// Can be null or empty.

/// </summary>

public string ExternalId;

/// <summary>

/// The term in the source language.

/// </summary>

public Segment SourceTerm;

/// <summary>

/// Translation of SourceTerm.

/// </summary>

public Segment TargetTerm;

/// <summary>

/// Indicates the beginning of the term in the source segment.

/// </summary>

public int StartPosInSegment;

/// <summary>

/// Indicates the length of the term in the source segment.

/// </summary>

public int LengthInSegment;

/// <summary>

/// Confidence of the translation between 0 and 101 (101 meaning

/// "Exact match with context"). If no confidence level available,

/// supply 0.

/// </summary>

public int Confidence;

/// <summary>

/// The 3+2 letter target language of the term.

/// </summary>

public string TargetLanguage;

/// <summary>

/// Entry-level metadata.

/// </summary>

public List<Metadata> EntryLevelMetadata;

/// <summary>

/// Language-level metadata of the source language.

/// </summary>

public List<Metadata> LanguageLevelMetadataSource;

/// <summary>

/// Language-level metadata of the target language.

/// </summary>

public List<Metadata> LanguageLevelMetadataTarget;

/// <summary>

/// Term-level metadata of the source term.

/// </summary>

public List<Metadata> TermLevelMetadataSource;

/// <summary>

/// Term-level metadata of the target term.

/// </summary>

public List<Metadata> TermLevelMetadataTarget;

/// <summary>

/// The HTML pretty print of the entry. The HTML described

/// by this member will appear under the hit list of the

/// memoQ client. It is optional. If it is null or empty,

/// then memoQ’s default pretty print will appear and it

/// displays the term and the meta information as well.

/// </summary>

public string PrettyPrintHtml;

/// <summary>

/// The color to be used when displaying the hit on the

/// translation results control. The color of the TB hits

/// will be used if not defined.

/// </summary>

public Color Color;

}

The *Metadata* class is the following:

/// <summary>

/// Holds name-value pair metadata.

/// </summary>

public class Metadata

{

/// <summary>

/// The name of the meta. Appears on the pretty print

/// view of the hit generated by memoQ.

/// </summary>

public string Name;

/// <summary>

/// The value of the meta. Appears on the pretty print

/// view of the hit generated by memoQ.

/// </summary>

public string Value;

}

The *TBAddinException* class is the following:

[Serializable]

public class TBAddinException : UserException

{

public TBAddinException(string message, string englishMessage,

Exception innerException = null)

: base(message, englishMessage, innerException)

{ }

public TBAddinException(SerializationInfo info, StreamingContext context)

: base(info, context)

{ }

}

You can use the first constructor to instantiate a *TBPluginException*. 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 plugin options

You have to create a class to be able to store the settings of the plugin. The naming convention is: **<plugin\_name>Settings.cs**. This class should be responsible for:

* Save and load the settings to/from the following location: <IModuleEnvironment. PluginSettingsDirectory>\<PluginName>Settings.xml
* Give back the settings for the other components of the plugin.
* Tell whether the plugin is configured properly based on the actual settings.

This class should give back only the saved or the default settings. Do not call any services from this class directly, this class should be a simple entity class which can save and load itself.

memoQ does not use this class directly, therefore there is no interface related to this component.

## The configuration dialog

The plugin should have a configuration 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 ShowOptionsForm 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.

## Localization

The third-party terminology 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 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 an other parameter which is the *pluginId*. This parameter should be the unique identifier of the terminology 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 terminology 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 derives from the *PluginDirectorBase* class and properly implements the necessary functions and properties.
* All allocated resources are properly disposed in the plugin director.
* There is an engine component, which derives from the *EngineBase* class and properly implements the necessary functions and properties.
* All allocated resources are disposed correctly in the engine.
* There is a session component, which derives from the *SessionBase* class and properly implements the necessary functions and properties.
* The *TBAddinException* class is used to wrap the original exceptions occurred during the translation.
* All allocated resources are disposed correctly in the session.
* There is a settings class, which is able to save and load itself to/from a proper location.
* The settings class is a simple entity class, does not call any services, and simply gives back the saved or the default 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 terminology service only during the configuration process or during operations. 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 new plugins

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

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

<ClientDevConfig>

<LoadUnsignedPlugins>true</LoadUnsignedPlugins>

</ClientDevConfig>

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

# Checklist to update a plugin for memoQ 8.0

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

* Target .NET 4.6.1 or newer version.
* Remove all memoQ codebase references, except for MemoQ.TBInterfaces.dll, and add MemoQ.Addins.Common.dll as reference.
* Update the AssemblyInfo.cs file, replace the namespace of MemoQ.Common.Framework to MemoQ.Addins.Common.Framework (for ModuleAttribute attribute).
* Update the implementation class of the IPluginDirectory interface to match the slightly changed interface.
* Update the implementation class of the SessionBase/ISession interface to match the new interface. All required entity classes (such as Segment) are now in MemoQ.Addins.Common assembly.
* Compile the library and fix compilation errors. In most only cases namespaces have to be fixed.
* Test the new plugin with the test client part of the SDK, or memoQ 8.0.
* Deploy the new plugin.