Silvestris Cyclotis Plugin for Trados Studio Configuration manual

Table of Contents

Introduction

This document is a complement of the « Installation » page of the web site, for more advanced users. It describes some technical aspects of the plugin, then the options you can modify in the configuration files to have a plugin more conform to your needs. In any case the plugin is delivered with a configuration file containing the options probably acceptable by most users, and comments explaining the other options.

Please note that this document concerns only the Trados plugin: neither other plugins nor configuration of the server are in the scope of this document. The fact that the plugin has a lot of configuration parameters means that it is highly generic, meaning that it could work with another server outside DGT. But for this document we will try to keep examples related to DGT usages.

Installation

The plugin must be installed in one (and only one) of the following locations:

- C:\ProgramData\SDL\SDL Trados Studio\11\Plugins (for Studio 2014)
- C:\Users\[account]\AppData\Roaming\SDL\SDL Trados Studio\11\Plugins (with [account] replaced by your login; for Studio 2014)
- C:\Users\[account]\AppData\Local\SDL\SDL Trados Studio\11\Plugins (with [account] replaced by your login; for Studio **2014**)
- C:\Users\[account]\AppData\Local\SDL\SDL Trados Studio\10\Plugins (with [account] replaced by your login; for Studio **2011**)
- For users of Studio 2015, take all values of Studio 2014 and replace 11 with 12 in all directories.

Actually the automatic deployment uses the first location for Studio 2014, and the last (only possible) one for Studio 2011. But it may change in the future. If you need to change configuration files, first have a look in all of these locations to see in which one the plugin is already present. In any case, *never* install the plugin for 2014 in more than one of the possible locations : this can cause bugs very hard to find.

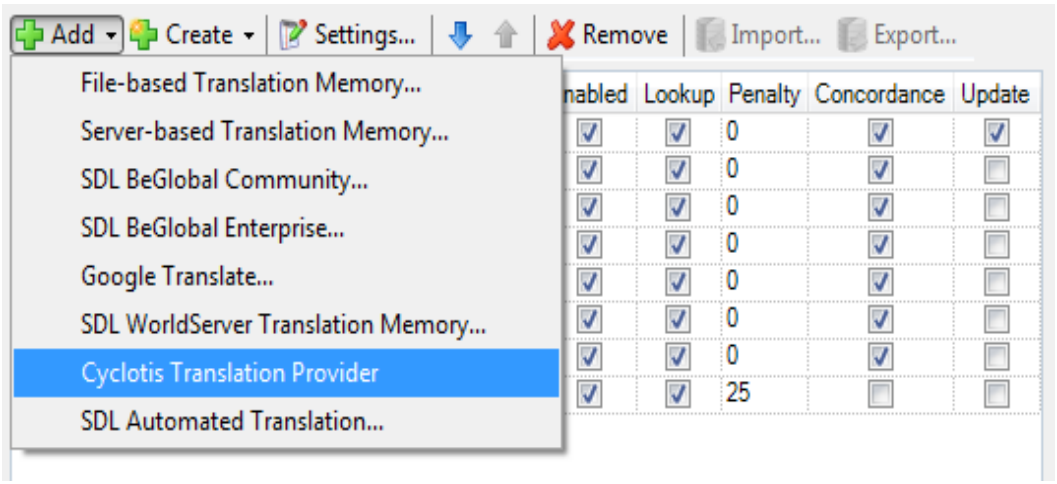
The plugin is divided in two directories:

- Packages : for each plugin this directory contains *one single file* with ".sdltm" extension. If you install a new version of the plugin, simply replace this file but do not make changes to other files which would be in this directory.
- Unpacked : for each plugin this directory contains one subdirectory with the name of the plugin. To install a new version of a plugin, delete the directory which has same name (and only this one) and replace with the contents of the distribution. Please note also that the configuration files you are susceptible to modify, and log files you could have to send for analysis will take place in this directory.

Access Modes

A memory is a database configured to store segments and respond to fuzzy search queries. There are two ways to access it: either using a SQL driver, or via an http proxy. Actually only the first solution is in use at DGT but the second one (both are supported by one plugin) will probably be used in some critical cases (secured documents, teleworking, etc.) in the future.

You select the access mode when you create the instance in the Studio interface:



Terminology

Before to describe the configuration itself, let's explain some terms which will be used later in this document.

Instance. In a studio project you can connect to more than one "memory". This is also true when you use Cyclotis: you can connect to one or more databases, some of them in update mode or not. Also all projects are not connected to the same memories. They can be in the same server or not, in the same database… To avoid any confusion I use the term *instance* for each separate memory. So, you have one plugin (Cyclotis) and one or more *instances*, either in the same project or not.

Variable interpolation. In the config file some values could change during the lifecycle of the project. As an example, imagine you want to set a parameter as "current segment number": of course the real value changes each time you go to a new segment, not letting you the possibility to hardcode it in the configuration file! For that reason, the configuration files include a syntax where you will use "variables", like ${Current.Segment.Id}, which will be recalculated each time the plugin requires for the value. This mechanism is called interpolation and will be detailed at the end of the document.

Configuration files

The configuration of the plugin has been divided in two files:

- Plugin-config.ini contains parameters which are loaded once and are related to the plugin itself and/or are common to all instances.

- Instances-config.ini contains parameters which may change from one instance (one connection) to another. If you copy this file in your project with the name of the instance you are connected to (because in your project you can be connected to more than one instance) you can change the parameters and they will apply to this instance only (however, they will only apply after you restarted Studio!)

Please note that, as any Windows ini file, these files are totally case insensitive, for the section and properties names, and *also for interpolation variable names*. But for the value of variables or parameters in the ini file, it may not be true!

Now let's describe the sections of Plugin-config.ini :

- describes the text which will appear in the Trados user interface. This enables to replace the name Cyclotis to something else.
- enables to configure what the autosuggestion provider does with the data it receives: how many suggestions will be retained, what is considered as a letter, etc. Please note that Instances-Config.ini contains also a section which indicates which data the provider will receive: if you want to make it locally inactive, it is better to do it in the local section. A more detailed description of the two sections comes later in the document.
- enables to pre-fill the user interface used to add a new instance of the plugin in your project : if you always connect to the same server, but not the same instance, you may force some parameters here
- enables to create configuration windows with user-friendly criteria (while default ones use technical criteria)
- enables to define some variants for the following file (Instances-config.ini) : they will appear when the user asks for Advanced Configuration.
- contains the common part of logging system. Description comes later in this document.

Now let's describe Instances-config.ini:

- contains the part of the logging system which can be different from one instance to another. The parameters are the same as Logging.Common but of course, values can differ. A more detailed description of the two sections comes later in the document.
- enables to configure the relation between autosuggest provider (which is configured in common part) and the current database connection. Here you can configure in which case the autosuggest provider will receive the segments from the database or not. A more detailed description of autosuggest configuration comes later in the document.
- `Limits` enables to configure numeric restrictions applied to the results, either for personal preference or to keep better performances;

- Npgsql enables to configure the database connector. Here you can for example increase timeouts or activate SSL. All options accepted by the driver are described here : http://www.connectionstrings.com/npgsql/ (do not specify here server, database and port: they are in the dialog box instead!)

  Note: for these two sections (`Limits` and Npgsql) you also have the possibility to have several copies depending on IP address : if you create for example section “Limits| 200.10.0.0/8”, this section will replace “Limits” section if and only if one of the IP addresses your machine has is conform to the specified mask. This is useful, for example, to have longer delays when you take your laptop at home but still want to access a database which is at your office : you do not need to have two config files for this case. Be careful that the specific section *completely* overrides the default one: do not expect that a parameter which is not in the specific section is taken from the default one, please repeat it instead.

- Sdltm.Proxy enables to configure the use of sdltm files to have features not available in the API

- `QueryDefs` enables to configure the behaviour of the plugin compared to Studio:

  - Whenever a "normal search" means "exact search, then fuzzy" (what is the case in sdltm by default) or if you simply want to do fuzzy searches to go faster;

  - Whenever the rules deciding between insert and update are implemented at client or server side. As the current server implements the rule, you probably won't have any reason to change this.

- enables to decide what should be stored in the database's 1st-level fields like author, and which score algorithm to use. Properties are in a separate section

- describes what you put in the database. describes which properties are visible in Trados interface. This is not necessarily the same as before: you can use a complex system in the database and a simple one for display, for example.
  The description of these sections comes later in this document.

Dialogs configuration

You have two possibilities to define the dialogs which appear when the user asks to create a new instance or to modify an existing one. Of course, both are in Plugin-config.ini, as you cannot define one dialog for each database!

Standard dialogs

The first possibility is to pre-fill the existing dialogs. There is one dialog per connection type, so one for direct and one for REST/HTTP connections. The possibilities are the following:

- direct.xxx or rest.xxx enables to pre-fill the corresponding value in the dialog. This, of course, is only used to create a database – to modify an existing one, the already existing

values will be used

- forced : if one of the previous variables is here, the corresponding field will be visible but grayed, so not modifiable. Be careful not to force a field whose value is not defined, this is a very common error!
- Hidden : if a variable is here, the corresponding field will not be visible at all to the user. The same remark as for forced variable is valid here.
- Build.url : if this field is defined, in the REST window a button named "…" makes enable to build the "URL" field using a window similar to "Direct".
- [rest or direct or all].validateOnCreation says that when you press "OK" on one of the windows to create a new connection, Trados will immediately try to open it and refuse to create the connector if it fails.
- [rest or direct or all].validateOnEdit is similar to previous variable except that it applies when you modify an existing connection.
- [rest or direct or all].check indicates whenever we want to gray OK button when parameters are missing (check = filled), or have a bad format (checked = strict). Value ‘check=no’ makes the button always active.
  When the button is grayed, a red label indicates almost one reason why you cannot confirm: each time you change the value of one field, a new check is done and the red label is changed or deleted, in which case the OK button is enabled again.
  Note that here we do a syntaxic check only : on the contrary, the two previous variables act after pressing OK and check that the resource exists or not.
- [rest or direct or all].Editor enables to choose which text editor is used when the user wants to use a modified version of Instances-Config.ini

Dynamic dialogs

Dialog Alternatives are used inside a corporation where all databases are hosted a similar way : instead of specifying technical parameters like computer name or port, the user may prefer to use criteria he better understands, depending on the nomenclature used for internal projects.

First, create the section [DialogAlternatives] and give list of profiles you will create : this is simply a comma-separated list of names which will correspond to variables created later in the same section. Then, use Alt.Name and Generic.Name to define the text which will appear to refer to "alternatives" (the profiles you defined) and "Generics" (the two standard profiles). Also define Generic.Pos to say whenever you want to see the generics before, after, or not at all.

Now, for each name defined in the List, you will create variables. For example if List=A,B,C you will have data prefixed by A. then others prefixed by B. and so on. You can also have values prefixed by Common.xxx : this does not mean that they are in all windows, but that they can be imported without recopying their contents. For example if you create a type called Lang, not all windows will use it but the ones who want to do probably not want to manually recopy the

definition each time they use it!

You can define three kinds of entities here: Types, Fields and constants. So, with the previous samples, you will have some Common.Type.XXX, Common.Field.XXX, Commons.Consts.XXX, A.Type.XXX, A.Field.XXX, A.Consts.XXX etc.

Actually, Type can be:

- A list: for example "Common.Type.A=list:A B C" will generate combo-boxes with all values of the list (separated with spaces or commas). Other types may be defined in the future.
- The value 'int', meaning that it must be an integer
- A boolean : type=bool followed by one or two values separated by ',' : the first value is what is considered as true, the second one, optional, what is considered as false. Note that these values can contain variable interpolation, if you want
- String format : type=string @@ 00000 indicates that this is a five-digit string. Today digits are the only values supported for this format.
- Regular expression : type=string =~ \d{5} is equivalent to previous form, but gives you more flexible format if regular expressions are clear for you.

For each field, let's speak about A.Field.X, you will have to define:

- A.Field.Title : what will be shown as a title for the user. Default is to use the field name (non-prefixed), which is often enough but not when you have a multi-word title.
- A.Field.Type : either a standard type as string (default), int, bool (will generate a checkbox), or one of the types defined before. You can also define a formatted string, "Type=string 0XXX" where 0 means "any digit" and other characters are taken literally.
- A.Field.Default : the value which is put in the field when you open the dialog for a new database (not for modifying an existing one). Here you can use standard variables (have a look at page 12). Preferably use the variables from projects view, because you had access to this window starting from it, not from the editor.

Finally, you want to configure the window with the fields you just defined. So, for window A you will have to define:

- A.Name : what will appear in the profile selection window button – this is a free text. Default is to take the term A itself, which is enough when it does not contain spaces.
- A.Dialog.Line1, A. Dialog.Line2, etc: decides in which order and line organisation the fields will appear in the dialog box. Fields can be taken either from the same prefix (non-prefixed, so to refer to field A.Field.F1 only tell "F1"), or from Common (so F1 means either A.Field.F1, or if not exists, Common.Field.F1)
- Finally, you have to define the URI which will be given to Trados once the user confirms the creation. This URI must have the same form as it should be when we make use of the standard Direct/REST windows. In this specification, make use of ${Field.XXX} to make

reference to one field of the current window.

Constants are useful to repeat a value in several URI (in the URI you will refer to it using ${Const.XXX}, *without the prefix*. Also, using recursive interpolation you can have more constants with a common schema and use them with a field, for example:

${Const.Port.${Field.XXX}} : first ${Field.XXX} will be replaced by its value, then the corresponding constant (including port) will be replaced.

SDLTM proxy

Some feature from Trados Studio, and especially from SDLTM files, are not open to the "Translation providers" API. When we asked SDL how to implement them, they suggested to create a sdltm file, import the segments retrieved by our plugin and to export them back, then they are enriched with the features from SDLTM file. We call this mechanism the sdltm "proxy". In the plugins directory you will also find some plugins doing the contrary : they use feature from Cyclotis to be added to standard sdltm files – then we call them reverse proxies.

In the section [Sdltm.proxy] you can configure

- Whenever you want to use it or not (removing the section or the "location" line)
- To use a sdltm file or to keep all in memory. Using a file may be necessary when you don't have enough memory, or if you want to trace for debugging purpose
- When it has to be cleaned (maximum size). It is generally not useful to keep results after they have been used, except of course if you use a file for debug purpose.
- What to do if the file already exists when we open Studio
- Finally, parameters starting with "Search.", "Import." or "Sdltm." are the ones which you usually configure at project level : maximal number of segments, minimal score, etc.

SDLTM cache

Trados's "task" named "Analyze files" generates a lot of queries in a short amount of time. In the case of , it is very likely to have a very short amount of results, because the database is supposed to be empty when you begin the project. And even if it is not the case (because you join a project which is already started by another user some minutes before you) it is very unlikely that the database changes during the small seconds the whole "analyse files" task runs. All this means that these queries are mostly useless but except if you explicitly disable before "analyse files" and reactivate it later, Trados gives us no possibility to exclude our plugin from the process, so we have to find a way to reduce the number of queries sent to the server.

To do this, we use exactly the same technique as before: using a sdltm file or its equivalent in memory. That's the reason why these two sections have exactly the same structure. The difference is not how we use it, but *why* and *when*: proxy is used *after* a query has given results, in order to add properties before showing to the user. Cache is used *before* a query is sent to the server, and enables to avoid them if we know from a previous query that the database is (quite) empty.

Logging configuration

The version 3 of the plugin (for Studio 2014 or later) enables to make the distinction between global log messages (which appear before an instance is working or are not related to an instance) and local (which are specific to an instance, and can also be reported in the project directory instead). But sometimes an instance message can appear in the common part due to the fact that the instance

is not ready (for example during its configuration).

Note : in this section, "common part" means the section [Logging.Common] in Plugin-config.ini, while "local part" means [Logging.Specific] in Instances-config.ini.

First of all, define whenever you will use specific logging or not, in the common part using the parameter SpecificLogging.Use : possible values include

- NULL : specific messages will be lost;
- NO : specific messages comes to common log file, but using the specifications of the common log file
- MERGE : specific messages comes to common log file, but you keep the possibility to configure the messages in the local section
- YES : specific messages come to another file (configured in the local section)
- BOTH : specific messages come to common log file and to specific. This is a combination of YES+NO (and not YES+MERGE)
- CONFIG : if you use this option, the correct value will be read in the local part instead. This enables to take a different decision from one instance to another.

Now you can define the file name(s). Both common and local sections have a parameter named FileName which supports variable interpolation: you will probably find an interest to add user or machine name in common file name, or instance name for local one. Look in section *Environment* for more details.

Unless you specify a full path, the common file will be in the "unpacked" directory of the plugin, and the local file inside the project itself (but will not be included if you use Trados's "create package" menu). You can have a look at the samples in the file itself to change this behaviour.

Now you can configure what should be logged or not. In the files delivered with the plugin, only the most common messages are activated, except in test environments, where everything is activated. A list of all possible options is in a comment inside the file.

By default, log files are not deleted when you start again Studio. This can cause the files to grow until the infinite. To avoid that, we also implement an archive mechanism: you can decide after which size (in mega or giga-octets) the file has to be archived, and whenever you want to zip, delete, or rotate (which means: keep only 5 zip files, then delete the oldest one)

Autosuggest provider

The version 3 of the plugin (for Studio 2014 or later) also comes with an autosuggest provider: when results are returned by a fuzzy search, until you go to another segment in the editor, if you begin to type a translation it can suggest to you words which come from the translations just extracted from the : even if it is not a 100% match it can be useful.

This provider can be seen as a client/server: after each search and before to return the segments to

the user, the plugin asks the server to reset (so that it will give you only suggestions related to current segment), then each running instance sends the translations it has found; when you are editing the translation, Studio calls the server (not anymore) to get the suggestions.

Due to this 2-level architecture, there are two parts in the configuration: in Plugin-config.ini you configure the server part, while in Instances-config.ini you configure the interaction between the current instance (which can be different from one instance to another) and the server: even if you are connected to more than one instance of , there is still one and only one autosuggest server.

Server/global configuration

The autosuggest provider receives complete segments, except that tags have been removed. The role of the server configuration is to decide how it should convert it to subphrases and prepare suggestions.

The first thing to do is to answer the questions: what is a letter, and what is a word? It sounds simple for alphabetic writing systems using spaces, like those used in Europe. But you may want to be more specific: how would you deal with dash or apostrophe?

Once the phrase is divided in words, for the word the user is typing the system will suggest not only words, but small segments containing more than one word. We assume (but you can configure it) that commas and final dots mark the end of a subsegment. For example in the phrase "this is a long phrase, and it contains a comma", when you are typing "this" the system should suggest you nothing longer than "this is a long phrase". But what if only "this is" is useful for you? To avoid problems, the system will suggest "this is a long phrase", "this is a long", "this is a", "this is" and "this". You can limit the number and quality of suggestions if it is too much for you.

Client/Instance configuration

In [Autosuggest.Link] section you simply define whenever you want to send data to the server or not, and if true, a minimal score for a segment to be sent or not.

Properties specification

The two sections work the following way:

1. A dictionary is created in which all line is replaced by the interpolated values. If the property name starts with '_' it is a meta-property, used for the following steps but not to appear in the final result.
2. In case you define property, each subfield (separated by commas) is really created (in the database or in the display). For the display, this also means that they will appear in this order. In the database the order may not be important (in case the database uses a generic properties type, which is generally transparent for the user)
3. In case you define , fields which are generated in step 1 but who don't appear in will be generated as well. indicates whenever you want to have them before or after the fields

4. If you define , fields listed here are not in the output. This is useful either if you have defined, or if you did not define at all (in which case all non-excluded fields are in the output, but in a random order)

You can define several sections if you made use of complex types in the database : for example I defined the type as . So, if I know that I have some tables using it and some others who don't, I can have two sections [Propertiesand the plugin will use the correct one after it detected the type for the current table.

In you also have the added possibilities:

- : if you use this, all properties extracted from are available with the prefix added. For example a property named 'X' is usually available as ${X}, but if it conflicts with another property, you shoud add so that you read it the property with name ${DB.X}, instead. The prefix is also used in the previously defined _Exclude variable.
- indicates what to do if property xxx is empty. The possibilities are:
  - : generate an empty text
  - : if the property is empty, do not generate the variable at all even if it contains other characters than the variable
  - Any free text. In this text you also have the variable ${err} which will be replaced by the name of the missing variable.

Pre-defined environment variables

Properties are extracted from Trados environment (some of them only in version 2014 and thanks to autosuggest, simply because the corresponding API was not available in older version).

Available properties :

- CurrentFile : (Studio 2014 or later) the file actually opened in the editor. You can use:
  - ${CurrentFile.Name} : only the file name, without any directory
  - ${CurrentFile.Path} : full path, starting from C:\
  - ${CurrentFile.Folder} : the folder inside the project. Usually blank, unless you create folders in Trados interface
  - CurrentFile.Ori : with the same subvariables (name, path,folder) but with the file without sdlxliff extension
- (Studio 2014 or later)
  - : the segment number
  - Other properties are available but seem to be always blank, so not used

- (Studio 2014 or later)
    - First of all, you must prefix using the correct context:
        - ProjectsView.CurrentProject in case you are in the view used to create/delete/load projects. Note that in case you are creating a new project, these variables may have a wrong value – do not expect this to be always correct.
        - EditorView.CurrentProject if you mean the project of the file currently loaded in the editor
    - Then you have the same variables:
        - : the full path of the project, starting from C:\
        - : project name, without path
        - Lang.src.iso2 or 3 : the iso code of the project's source language
        - Lang.tra.iso2 or 3 : the iso code of the project's target language (avoid to use this if your project is multilingual)
- : the Windows system environment
    - : user login; mainly useful for "author" property
    - : user login with domain name
    - MachineName : the computer domain name
    - Environment.Var.* : with * replaced by a variable, will contain the corresponding Windows environment variable. You can also prefix it with User, Machine or Process to be more precise.
- : info about the current database. Subproperties are the ones used in the code (depends on whenever you do a direct or REST connection), plus two standard properties named Short

Interpolation capabilities

When you define a property in the config file, each instance of ${...} will be replaced by the corresponding value if Trados is enable to build it. This mechanism can appear more than once in a parameter (you can define parameter Sample=${var1};${var2} for example) but is not recursive (you cannot define Sample=${${Var1}} and hope that it will do two-step evaluation).

Please first be conscient that it is replaced by the value ***at the time we ask for it***: for example it is probably an error to use ${CurrentFile...} in Properties.TradosDisplay because it would be replaced when you are editing the segment, and then you know where you are! On the contrary, put it in Properties.DatabaseStore is a good idea: this will be used to fill the database, and when a new user does a fuzzy search returning this segment, his "TradosDisplay" section will read from the database

instead of from his current file, which is not the same as yours…

In the ${...} you initially put variable names, whose list I just gave in previous section. The section Properties.TradosDisplay also understands all variables defined inside Properties.DatabaseStore. Also any property defined in a section is available in the same section.

As this syntax is known to be very complex, let's define it progressively with samples.

Basic sample

Let's imagine we have a set of environment variables A="01", B="2" and C="3" (the contents is a string, not a number). Now we can use them to define other variables.

Sample1=${a} - ${b}:${c}

In this sample, we will create a variable named Sample1 which contains the string "01 – 2;3" : we simply replace every occurrence of ${…} by the evaluated expression.

Splitting a variable

In DGT, document file names usually make use of a nomenclature with the form "requester-year-number", with dash as separator. If the variable FileName contains such a name, we could define that

Requester=${FileName|-|0}

The symbol | indicates that we want to split the variable; in the middle, we tell each symbol used as a separator (in our case only dash, but we could accept more than one symbol:

Requester=${FileName|-|0}

Requester=${FileName|-._|0}

In this sample, all three symbols - . and _ are accepted as separators. Note that you lose the information about which symbol indeed was recognized as a separator during the parsing of the name.

Finally, the right number is the position in the array you want to take. As in Perl, 0 is the first value, 1 the 2nd one, 2 the 3rd one, etc. and -1 is the last one, -2 the pre-last one, etc.

For example if FileName = DGT-2014-12345.doc then

${FileName|-|0} = ${FileName|-.|0} =DGT

${FileName|-|1} = ${FileName|-.|1} =2014

${FileName|-|2} = 12345.doc (yes, the dot is not a separator here)

${FileName|-.|2} = 12345 (now yes, it is)

${FileName|-.|3} = ${FileName|-.|-1} = doc

${FileName|-.|-2} = 12345

Etc.

If we were sure all requester names have the same size, we could use the position in the name itself. In the hypothesis they are 3 letter long, we could also write:

${FileName:9,5} = 12345 (starting from position 9 – don't forget we start from 0; taking exactly five characters)

${FileName:4..7} = 2014 (starting from position 4, ending with position 7)

Alternative variables

Now imagine that some users enter the requester in the database, while others prefer to enter the file name and let Studio rebuild the file parts. So, sometimes the variable DB.Reqester is defined, sometimes not. So we want to look at variable FileName only if DB.Requester is empty. Let's do so:

Requester=${DB.Requester,FileName|-|0;Table.Name|._|1}

Here is what will happen:

1. the system will search variable DB.Requester. If found, go to step 3
2. the system will search variable Filename,
3. Split the first value found using dashes and get first field. If not (either DB.Requester and Filename were empty *or the first non-empty one did not contain dashes*) we have to continue
4. then try to read Table.Name, split using '.' and '_' and get 2nd field.

Please note that we made use of two sperating operators : colon ',' and semi-colon ';'. The difference is the priority : colon makes separation between variable names only, and splitting operators are applied on the result; on the contrary, ';' separates full expressions, including splitters. Now let's consider the following expression:

Requester=${DB.Requester;FileName,Database.Name|-|0}

Here the operator |-|0 applies to the expression FileName,Database.Name (and not DB.Requester which is in another expression). Again let's detail what happens here :

1. the system will search variable DB.Requester. If it is found, the search is finished
2. the system will search variable Filename
    a. if it finds it, it tries to split it and return first field
    b. if it is not found or could not be splitted, it searches for variable Database.Name
        i. if it finds it, it tries to split it and return first field

Finally, have a look at the following expression:

Requester=${DB.Requester|-|0;FileName|-|0;Table.Name|._|1}

This sounds equivalent to the first expression, as if comma was only here to factorize the splitting operators, but it is not strictly true. Here is what will happen:

1. the system will search variable DB.Requester. If found, split it to dashes and takes 1st field. If not, continue to step 2. If non-empty but not splittable, go to step 2.
2. the system will search variable Filename, if found split it to dashes and takes 1st field. If not, go to step 3
3. then try to read Table.Name, split using '.' and '_' and get 2nd field.

So, as you see there is a difference in case DB.Requester is not empty but contains a non-splittable value: with the comma, Filename will not even be parsed, we directly go to Table.Name. If it is not what you want, use ';' instead!

Missing variables

But what if even Table.Name or its splitting did not succeed in the previous expressions?

First of all, we must say here that at this step, the system only has in memory the last error it encountered: it does not remember about DB.Requester or FileName anymore, it only remembers that it encountered an error for Table.Name. Oh, well, it is not absolutely true: it also remembers that it was trying to generate Requester.

The default behaviour is to provide the property Requester with 'Table.Name'? as a value. This should sound strange for a user but it is very useful for the person who writes the configuration, as it can indicate an orthographic error, for example. But of course, once we checked that this is correct in normal case (i.e. the variable name is correct, it should exist) then it means that we have to find a more user-friendly solution.

Each error is reported in the logging system if you activated the condition var-errors. Also if you activated var-errors-detailed, you could also see errors related to DB.Requester and FileName, as the error will be generated before we go to next variable and loose the information. But maybe it is not a good idea to tell it to the user…

The first possibility to build a more user-friendly value is to use _Missing variables. In our example, the system will look for a variable _Missing_Database.Name (remember that in the properties, data prefixed with _ are meta-information instead of real property specifications) and if not found, will use the variable _Missing as default. And the default value is '${err}'? which means that it should be replaced by the name of the variable which caused the problems, between quotes and followed by a question mark : this is exactly what you saw two paragraphs ago.

Another possibility is to generate an empty text, using _Missing=#Empty. Note that it happens before we generate the key/value pair. Let's go back to our very first sample, remember:

Sample1=${a} - ${b}:${c}

Imagine that a = 01, c = 3 but b contains nothing. Without any _Missing or _Missing_b variable, the result would look like this

Sample1=01 – 'b'?:3

While with _Missing_b=#Empty or _Missing=#Empty you will see

Sample1=01 –:3

Another possibility is to use #Cancel in the value of _Missing or _Missing_b : in this case, if any of the contained variables is empty or contains an error, Sample1 will not be generated at all: the errors only appear in the logs.

Maybe you don't want to take the decision at the level of variable "b", but at the level of variable Sample1? In this case use

_Error_Sample1=#Cancel

_Error accepts only this value, of course not defined by default. The difference with _Missing is that it is related to the left part of the expression (before = sign), while _Missing variables are related to the right part.

Sometimes you would like to generate a variable (the left part) only if another one could not be correctly set. For example, if I cannot generate (requester, year and number), I would like the user to see the filename – and only in this case. Alternatives do not help me, they affect only the right part of the expressions. Then we can define

_Instead_Requester=File

Like _Error variables, _Instead variable**s** (note that there is no global _Instead variable) take care of the key name, left part of the expression. The contents is a list of variables, either with a value or not, for example:

_Instead_Sample1=Key1=${FileName|-|0},Key2,Key3

If a value is given, it will be interpolated (avoid to use commas in it, it may confuse with global separator). But it is better not to define a value here, or a very simple one: normally it will use the variable you defined elsewhere in the document.

Well, a last point here. Don’t forget that in the properties part, the only properties which will be visible to the user are the ones defined in _Order (or if not defined, what is not in _Exclude). In this sample, variable File must ***not*** be in the order list: in case of error it temporarily, locally, takes the place of Requester in the chain. But if it is already in the chain, it could appear twice in the result!

Typed variables

Most often a variable may only be in a specific range of values.

You can specify this type using meta-variable _Valid_XX, where XX is the name of the real variable you want to describe. The type specification is exactly the same as in dynamic forms.

This definition will be used to know if _Missing or _Instead variables should be activated or not. This prevents from displaying values which are not empty but do not reflect your specification, due to a wrong database or file name. But this works only for variables which appear in the left of an expression (i.e. as a key in the ini file). What if we want to use it inside ${…} ?

To do this the syntax is ${@Type#...} : do not forget @ nor #. Then the expression in '…' is the same as what we learned before. The type will be checked for all the expressions **separated by ';'** - not by ',', because the type checking occurs to the finished expression, after execution of '|' or ':'

subexpression. Type checking for subexpressions is not implemented yet.