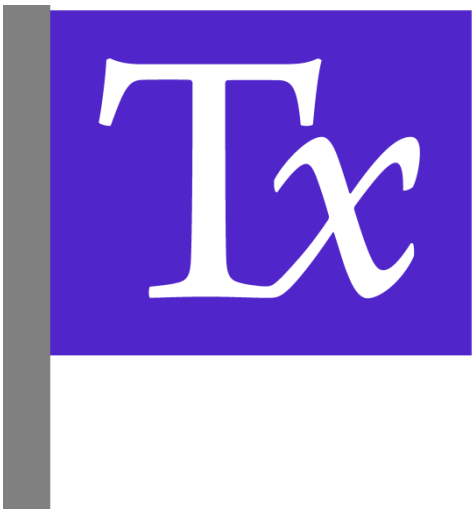

Tx Translation & Localisation for .NET and WPF

Overview and Documentation

Designed and written by Yves Goergen
Last updated on 2013-09-01 [DRAFT VERSION]
Contact: yves@unclassified.de
Project website: dev.unclassified.de/source/txlib

Contents

1. The Tx concept

Tx is a simple yet powerful translation and localisation library for .NET applications. It manages a dictionary containing all the text snippets and phrases you need, for multiple translations. If a translation is not available in your preferred language, it can be looked up from other languages. Texts can also contain named place-holders that are filled with your data at runtime so that you don't need to concatenate all the parts yourself in the right order. Localisation tasks like typography, number or time formatting according to the local standards are also covered.

A special feature of Tx dictionaries is that each text can have different translations depending on the subject count you're talking about. One *day* is a different word than two *days*. This allows you to speak to your users in the most natural way, avoiding ugly parentheses or alternatives for plural words in the user interface.

All texts and translations are stored in an XML dictionary file, one per project. This XML file usually contains all languages. The dictionary can be installed with your application, or compiled right into it as an embedded resource.

The TxLib library can be used from any .NET application and is the part you'll distribute with your application. For more portable applications, you could also directly copy the class files into your project – it's only a few files. Its main class, Tx, is further described in the next chapter.

1.1. Text keys

All texts are associated with text keys. A text key is a short string that uniquely identifies a text or phrase and is usually used in the application source code to refer to a specific text. Text keys can be freely assigned by the developer, but there are some structuring guidelines described at the end of this document and a few additional requirements when using the TxEditor tool.

1.2. Cultures

The .NET framework uses the concept of cultures to specify a combination of language and data formatting rules. Each culture has a code that describes the language and optionally a region. There are more variants but they are not supported by TxLib. Examples are "en" for the English language, "de-DE" for the German language in Germany or "pt-BR" for the Portuguese language in Brasil (as opposed to Portugal which many would probably first think of). TxLib uses these cultures to identify translations and both, region-specific and language-only codes, are used where the latter ones always serve as fallback if a translation is not found for a specific culture.

2. Using the Tx class

The Tx class is the heart of the TxLib solution. It provides the dictionary lookups and does all the translation and formatting work. It is designed as a static class so that it can easily be used everywhere in your application without creating or finding an instance of it first. The dictionary is loaded into memory which is usually done at application initialisation time. The dictionary is read from an XML file on disk or an embedded resource. When reading from files, they can optionally be monitored for changes so that the dictionary is immediately updated.

The main method of the Tx class is the shortcut method simply named **T**. It accepts the text key and has multiple overloads for additional parameters like a count or placeholder data. It returns the translated text as a string. Its use is as easy as:

```
Tx.T("my text key")
```

This will look up the text key "my text key" in the dictionary for the current culture which is provided by the .NET framework. In order to use a different translation, you can set the current thread culture to a different value, resulting in a consistent switch for both TxLib and the rest of the framework (like for instance number parsing).

2.1. C#

To access translations in your application or library code, you'll normally use the Tx class and its methods. TxLib is written in C# so you can only copy the code in C# projects, but being a CLR library, you can of course use it from any other CLR language like VB.NET, Managed C++, F# or whatever you have.

2.1.1. Initialisation

Before you can access any translated texts, you must load a dictionary. This is typically done at application initialisation time. You have several options here:

1. To load all files in a directory, you can use the **LoadDirectory** method. It finds all dictionary files in a given directory, optionally matching a specified file name prefix.
2. Loading a single dictionary file is done with the **LoadFromXmlFile** method. It reads the specified file and loads it into the global dictionary.
3. If you're not deploying the dictionary as a separate file on disk but instead as an embedded resource in the application assembly, you can load it with the **LoadFromEmbeddedResource** method.

When files were loaded from disk, you can later check these files for modifications and have them reloaded if they're changed, using the **CheckReloadFiles** method. Alternatively the **UseFileSystemWatcher** property can be set to automatically monitor all loaded files. See the ASP.NET section below for additional notes.

You can also add text keys in your own code, but this is more suitable for small dynamic additions rather than complete dictionaries. Use an overload of the **AddText** method for that.

2.1.2. Culture control

There are multiple methods and properties to retrieve information about the currently selected culture or all available cultures in the dictionary. The **AvailableCultureNames** property returns a string array of all culture names currently available in the global dictionary. The **AvailableCultures** property instead not only returns the names but the CultureInfo instances for them. The **CurrentThreadCulture**, **CurrentThreadLanguage** and

CurrentThreadCultureNativeName properties are only shortcuts to the framework's CultureInfo.CurrentCulture property.

The **PrimaryCulture** property gets or sets the primary culture name that serves as fallback for incomplete translations. If this is null, it will not be considered when translating texts. It is not guaranteed that this culture is available in the global dictionary.

Use the **SetCulture** method to change the current thread's culture. This only affects the currently executing thread and is also regarded by other .NET framework functions. The **GetCultureName** method returns the best supported current culture name for the thread. This can be CurrentThreadCulture, CurrentThreadLanguage, one of the browser's supported cultures (if specified), PrimaryCulture or, if none of them are available, any one of the available (loaded) cultures. This culture will be tried first to find translated texts.

2.1.3. Text translation

The main translation method is called **Text** and it always accepts a text key and has several overloads for combinations of additional data like a subject count or placeholder data. For less code to write, there is the T method that is basically just a pass-through method.

A special case is the **SafeText** method. It tries to find a translation for the specified text key but won't use fallback languages. If no translation was found in the dictionary, the key namespace part is stripped off the the remaining text is returned as the translation. You can use it to partially translate your data strings where most elements go untranslated. An example is city names where only some cities have translated names in certain languages. the text key would then be a namespace prefix like "city:" followed by the native name. The city of Prague is called *Praha* in Czech and *Prag* in German. All other cultures not defining their own translation will just get the native name. If your data is already in a consistent language like English, there would only ba a German translation for the city of Vienna, in this case *Wien*. This will be used for all variants of "de" cultures whereas everybody else gets the English name.

2.1.4. Counts and placeholders

Sometimes, a different word should be used depending on the number of subjects the message is about. In such situations, a **T** overload with a count parameter can be used to specify the number of subjects. TxLib will then use a more appropriate translation for this call if available.

To include runtime data in the translated texts, you can define placeholders in the texts. These placeholders need to be passed with the **T** method call. The easiest form is to use an overload that accepts placeholder names and their corresponding values directly after each other as a variable number of arguments. There is only very limited IntelliSense support in this case, like for any other method accepting a variable number of arguments, so you need to be a little more careful here. The syntax is this: You specify all placeholder values one after another, each by two arguments. The first argument is the placeholder name as a string, the second is its value which must be convertable to a string. Here's an example with multiple placeholders:

```
Tx.T("user info",                // The text key
    "name", userName,            // Pairs of placeholder names and values
    "location", User.Location,
    "status", statusValue);
```

Combinations of count and placeholders are also available.

2.1.5. Text conversion and decoration

Different languages have different typographical rules that should be regarded to provide the most comfortable translations to your users. There are methods available for quoting and nested quoting of text, placing it

in parentheses, adding a colon for tables or lists, and transforming the beginning to upper case. While the last method is only a shortcut to existing .NET functionality, all others use strings defined in a special area of the dictionary. This allows the translator to specify matching quotation marks and other characters for a language.

Quotation marks are different in almost every country and there are completely different styles of characters for this purpose. Also, parentheses look different in some Asian languages. The colon was added for the French who normally put a space in front of colons and a few other punctuation marks. And lastly, the upper case shortcut can be used to define a word in lower case to use it in any context but to transform it into upper case when used at the beginning of a line without maintaining two similar text keys for it.

There are also abbreviated alternatives for these methods, and what makes it even more useful, combinations of these abbreviated names. While the **T** method only performs a dictionary lookup, the **UT** method also transforms it into upper case and the **QTC** method puts the text in quotes and adds a colon. Please check the complete library reference for a list of method names.

2.1.6. Number, date and time formatting

While basic number formatting as well as date and time formatting is already provided by methods of the .NET framework, TxLib goes beyond that and adds formatting of data for typographical contexts and more closely to local standards. Most regional authorities, for example, encourage using a thin space as the thousands group separator to avoid any confusion with the decimal point or decimal comma. Additionally, typography disallows line breaks within such numbers. The .NET framework does not regard these recommendations and simply uses a comma respectively a point for this purpose. Also, in some regions like de-DE, thousands grouping is not used for numbers of four digits or less. The **NumberUnit** method combines a number and a short unit name with a non-breaking space where indicated. The **DataSize** method knows magnitudes of 1024 and the **Ordinal** method gives you ordinal numbers in your language.

There is also the **N** abbreviated method for number formatting.

The date and time formatting of TxLib surpass .NET's formatting by providing more combinations of fields to include. You could only express a day and month but no year, or just an hour but no minutes or seconds. Also, month and year, or quarter and year are supported in case you need those. All numeric, abbreviated or in full words. For a more individual touch, time spans can be expressed relatively, for different use cases. The file was updated "3 hours 20 minutes ago" (instead of 3:20:34 which may not always be very useful), but the session still lasts "for 8 minutes". Other languages than English require all the differentiation due to different grammatical contexts.

TxLib cannot parse numbers or other data from the strings it has generated. If you need to parse the strings you display, you should stay with the .NET methods for formatting and parsing. This is the case in two situations:

1. Pretty printing numbers and dates in the user interface where the user can edit the values and save them back. Here, you shouldn't use features like thousands group separators anyway because they're likely to be out of place after the user has edited a number.
2. Serialising data for storing or transmitting. In this situation, any localisation is out of place. Since you'll be transferring data between parties of different cultures, you really should define one universal format anyway.

2.2. WPF

WPF (Windows Presentation Foundation) applications have the great benefit of binding data to the user interface (the view). Any object that exposes public properties can be used as a source of data. Additionally,

you can use converters on the raw data and implement your own XAML markup extensions to add more functionality to the bindings. TxLib makes use of these features to provide a simple syntax for text translations directly within your XAML markup. It looks similar to what you know about the Tx.T method, but is regularly integrated in the XAML syntax.

First, to add Tx support to a XAML file, you need to add its namespace declaration. For a Window element, it looks like this:

```
<Window x:Class="WpfDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:Tx="clr-namespace:TxLib;assembly=TxLib" ...>
```

The namespace is now "Tx", just like the Tx class you already know. This namespace provides a number of markup extensions, again with similar names like the described methods. To insert the translation for a text key, use the {Tx:T} markup:

```
<TextBlock Text="{Tx:T my text key}"/>
```

It requires the text key as first argument and optionally accepts a subject count as second argument. The count can also be bound to another source value:

```
<TextBlock Text="{Tx:T n months, {Binding Value, ElementName=Counter}}"/>
```

Similar to the abbreviated method names, you can also use **Tx:UT** for upper-case text, **Tx:QT** for quoted text, **Tx:UTC** for upper-case text with a colon, and so on.

Besides the variants of **Tx:T** for text translation, you can also use **Tx:Number** for number and **Tx:DataSize** for data size formatting as well as **Tx:Time** for date and time formatting:

```
<TextBlock Text="{Tx:Number {Binding Power}, 3, Unit=kW}"/>
<TextBlock Text="{Tx:DataSize {Binding FileSize}}"/>
<TextBlock Text="{Tx:Time {Binding LastUpdate}, Details='DowLong,YearMonthDay'}"/>
```

Relative time specifications are also supported and come with the additional bonus of keeping themselves up-to-date as the time passes. Here are two alternative markup variants for the same result:

```
<TextBlock Text="{Tx:RelativeTime {Binding SelectedDate, ElementName=DatePicker1},
                 UpperCase=True}"/>

<TextBlock>
    <TextBlock.Text>
        <Tx:RelativeTime TimeBinding="{Binding SelectedDate, ElementName=DatePicker1}"
                         UpperCase="True"/>
    </TextBlock.Text>
</TextBlock>
```

To select from one of the grammatical contexts of the relative time, set the **RelativeTimeKind** parameter to the desired value. (**None** displays an absolute time value, **PointInTime** and **CurrentTimeSpan** work on a DateTime type value and **TimeSpan** requires a TimeSpan type value to display.)

2.3. Windows Forms

Much of the text to translate usually lies in the user interface definitions of a GUI application. While Windows Forms doesn't provide the same level of binding comfort as WPF, TxLib still makes use of every supported feature to ease the most common translation tasks. It provides the **TxDictionaryBinding** class that handles the Windows Forms type of data binding to the Tx dictionary. Simply call

```
TxDictionaryBinding.AddTextBindings(this);
```

in the Form's constructor (after InitializeComponent) to let it find all bindable controls that have a text key in their visible text at design time. So all you need to do is place your Label controls as usual and set a text like "[my text key]" (without the quotes) on them. Put the actual text key in square brackets.

To use a dictionary text in another property of a control, you can add that binding with the following method call:

```
TxDictionaryBinding.AddBinding(IntroLabel, "Text", "intro");
```

And finally you can always assign texts the manual way, without bindings and automatic updating:

```
IntroLabel.Text = Tx.T("intro");
```

The Windows Forms bindings are only updated when the primary culture is changed (for example through a list selection control) or when monitored dictionary files were modified. Additional features like subject counts, placeholder data and number or time formatting from other bound sources are not supported for Windows Forms.

2.4. ASP.NET

Web applications can be used by multiple users at the same time. Each request is handled in a separate thread. To set different languages for each user, the current thread culture must be set at the beginning of each page request. This per-request initialisation could be done in the Global.asax.cs file in the Application_BeginRequest or Application_PostAuthenticateRequest method. This is also where the checking for modified dictionary files with the **CheckReloadFiles** method can be added.

The **SetWebCulture** method is useful for web contexts where the browser indicates a number of language preferences in the HTTP request header. This ordered preference can be used by TxLib to provide better fallback translations as understood by the user.

Technical note: Since referencing the ASP.NET classes requires the full .NET framework and is not covered by the client profile, this is not directly included in TxLib. So you still have to provide the connection and pass the request header data to this method for it to do all the rest:

```
Tx.SetWebCulture(HttpContext.Current.Request.ServerVariables["HTTP_ACCEPT_LANGUAGE"] as string);
```

3. TxEditor application

The TxLib solution also comes with a graphical translation file editor that provides all the functionality for translating personnel that don't necessarily have any programming experience. The TxEditor application can load a dictionary file, list all the text keys that it defines and allows viewing and editing the translation texts. It performs consistency checks to point the user to potential errors and highlight missing translations. TxEditor also makes use of TxLib itself so it is fully localisable and can be used to translate itself into other languages.

3.1. Dictionary files

TxEditor loads a dictionary file into memory so that the user can edit its contents. This works just like with any normal document or picture editor. There can only be one dictionary loaded at a time. In the current file format (version 2), dictionaries are stored as a single file for all text keys and cultures. In the older file format, a separate file was used for each culture. This is what the **Load Directory** button is used for. All file-related buttons are in the File section of the toolbar.

Import and Export

3.2. Cultures

A culture is the .NET framework concept of a language and optional region to support localisation. A dictionary contains a set of cultures for which translations can be added. Cultures are selected on the dictionary level, so for all text keys within a dictionary, a culture is either available or not. Cultures without a region specification are always used as fallback if a text wasn't found in a region-specific culture.

To add a new culture to the dictionary, press the **New culture** button in the Culture section in the toolbar. In the dialog window, you can browse and filter all cultures that are supported by the .NET framework installation. After a while getting used to the culture codes, you may already know the code you need and can type it directly in the text box at the top. Click the **Add culture** button to add the selected culture.

When a text key is selected, an input field for each available culture is displayed in the text editing area of the application window. The cultures are sorted by primary first, then alphabetically by language code, each followed by the specific regions. Once a culture is added to the dictionary, it is available for all text keys automatically.

Cultures can also be deleted from the dictionary by first focusing a text input field of the culture to delete (any selected text key will do) and then pressing the **Delete** button in the Culture section.

With the **Set primary** button, you can specify the selected culture as primary culture for the dictionary. Again, focus any key's input field of the desired culture first. The primary culture is normally the one your development team is communicating in or the culture of your largest user group.

The **Tools** button opens a menu of additional commands. You can replace one culture by another to effectively move all present translations from one culture code to another. This may be useful when restructuring regional-specific cultures. You can also insert the internal text keys for formatting functions to **TODO** if it is supported by TxEditor. Lastly, there is a dialog available to browse .NET's own formatting of numbers and times which can give an impression of how data will look in a certain culture, and serve as basis for additional time formats in the Tx namespace.

3.3. Text keys

Text keys are the main part of a dictionary. All currently defined text keys are displayed in the tree view that makes the left part of the main window. It can be navigated just like a folder view in the Windows Explorer. To view and edit the translations for a text key, select it from the tree. The editor area at the right side of the main window always shows the texts for the currently selected key (see next section).

To add a new text key, press the **New key** button in the Text key section of the toolbar. A dialog window opens to let you enter the name of the text key to add. After you have added some keys, you will notice that the text field already contains the currently selected text key with parts of it preselected. Only the last segment of the key is selected so that you can quickly go three different ways:

1. Just start typing to replace the selected text and add a new key on the same level as the selected key.
2. Press the Right arrow or the End key to continue typing after the preset text and add a new key one level below the selected key.
3. Press Ctrl+A and type a whole new name to add a key with no relation to the selected key.

This presetting and selection of the text is only a comfort feature to give you a good start to type a key name. After all, everything that matters is the full key name in the text field when you confirm the dialog by pressing the **Add text key** button. The new key will be selected in the tree view and you can directly continue typing the first translation.

While the Tx library can handle any text key string in the dictionary, the TxEditor tool imposes a few restrictions on the text key syntax due to its hierarchical approach to text key management:

- The key name cannot contain multiple consecutive points (**.**).
- The key name cannot contain multiple colons (**:**).
- The key cannot be empty or contain only white-space.
- The key cannot start or end with a colon (:) or point (**.**).

Selected text keys can also be deleted with the **Delete** button, renamed with the **Rename** button and duplicated with the **Duplicate** button. Deleting works with multiple selected keys, but renaming and duplicating only supports a single selected key because you need to enter a new key name in a dialog window. In any case, these three operations optionally include all text keys down the tree hierarchy. You will be given appropriate check fields or confirmation dialogs when applicable.

3.4. Text editing

When a text key is selected from the tree, its translations for all available cultures can be viewed and edited in the editor area at the right side of the main window. It prominently shows the full selected text key at the top, followed by an optional comment field and then all culture texts.

3.4.1. Comments

Each text key can have a comment associated with it that purely serves documentation purposes to explain the usage context of a text key to a translating person who is not part of the development team. Since the text keys in TxEditor are not viewed in the context of their use in the target application, such explanations may be helpful. There is no restriction on the contents of that field, but be aware that it is part of the dictionary file and as such being deployed with the application. Don't include any internal material in the comments and don't write too long stories. This input field can be toggled with the **Comment** button in the View section of the toolbar.

3.4.2. Text input

The rest of the editor view contains the input fields for each culture. The primary culture, indicated by the green culture name, is always at the very first position. This is the fallback culture that will be used at runtime if nothing else works. Then, in the order described in the Cultures section above, all other cultures are listed. Each culture text has a default unquantified translation text. Just click on the "Enter text" placeholder and start typing to enter the translation. The text field height will extend to its contents.

3.4.3. Quantified texts

Additionally, quantified texts can be added for each text key/culture combination. These texts will be used when the application indicates a matching subject count. This allows using a different translation for singular and plural forms, or in a more creative way to use different texts for multiple count values. Any integer count value can be used here. To add a quantified text, press one of the buttons at the right side, labelled **0**, **1**, or **+**. The third button just adds an empty item and the other two preset the count value with 0 and 1, respectively. The count value can be entered in the first text field of that row, right after the "=" sign. To delete a quantified text, press the × button to the right of it. Some languages like for example Polish require additional tests for the count value modulo 10 or 100 to select the correct word form. If such a modulo test shall be used, the number can be entered in the second field, after the word "mod". To refresh the sorting of all count entries after editing them, press the ↻ button, the fourth in the row.

3.4.4. Placeholders

If you need to include user input, dynamic data or other information within your translated texts, you can add placeholders for them which will be resolved at runtime. Each placeholder has a unique name an is specified inside braces in the translation text. A placeholder may be used multiple times in a text. The actual values for these placeholders are passed to the `Tx.T` function at runtime. An example for a simple placeholder is this:

Are you sure to switch the primary culture to {name}?

TxEditor will highlight placeholders with a beige background colour. The placeholder is called *name* here and its value must be passed at runtime. A special placeholder is the hash sign (#), which is resolved to the specified count value. With this, you can use different translations for each count value and display the count value at the same time, without passing it twice to the function. Examples:

{#} files loaded	(Generic form)
File loaded	(For count 1)

3.4.5. Key references

As a more advanced feature, you can add references to other text keys in your translations. This may be useful if you want to include similar texts or text snippets defined elsewhere (under a different text key) instead of copying their content verbatim which may become outdated sometime. To add a reference, place the other text key in braces, followed by an equals sign (=), like this:

{=error}: The file cannot be opened.

Key references are highlighted with a blue background colour. You may even pass along a count value to the other text key in case it's necessary or useful, by appending a hash sign (#), or pass an integer placeholder value as count value to the subkey, by additionally appending the placeholder name:

Translated text. {=other key#}
Translated text. {=other key#mycount}

TxEditor checks these references and marks invalid references with a red underline.

3.5. View options and filters

There are several options to customise the editor view available directly in the toolbar. First there are the navigation commands **Back**, **Forward** and **Definition** that allow you to quickly navigate through the previously selected text keys and the text key referenced under the cursor in a translation input field, if available.

The other buttons in the View group of the toolbar change the text appearance and toggle additional helper views. The **Monospace** button switches to a fixed-width font for all translation texts which may be easier to read when many narrow characters or signs a used. The **Hidden chars** button visualises normally invisible characters in the translation texts like spaces or line breaks. Different types of spaces are partly marked with different symbols to help distinguishing them.

The **Character map** button shows or hides a list of characters under the toolbar that you cannot type on your keyboard. When any of those characters is clicked on, it will be inserted at the current cursor position in the translation input field. This allows you to enter special characters for foreign languages, typographical punctuation like quotation marks or diverse spacings, or other decorative characters. Each button shows its codepoint and Unicode character name in its tooltip. The character list can be specified in the application settings.

The **Suggestions** button shows or hides the list of similar translation texts as described in the Suggestions section below. The **Comment** button shows or hides the comment input field for all text keys. The text key comment can explain a text key's usage and help in choosing a reasonable translation. See the Text editing section above.

To find text key names or occurences of certain words, type the search terms into the input field next to the **Search term** label. Matching text keys will be displayed in the tree view as you type your query. **Options**

The **problems** filter button shows the current number of text keys with validation errors and switches the filter to only show these text keys. You can use it to quickly find and handle all problems with translation texts. Text keys do not disappear from the filter view when the problem is resolved until you modify the filter settings again.

Culture filter

3.6. Application settings

TODO

3.7. Validation

To support the user to create high quality and consistent translations, the editor performs a number of tests on the entered translation texts. If any problem was found, the text key icon in the tree changes its colour from black to red and a short description of the problem is displayed next to the key. All parent tree nodes of a key with a problem are also red so that nothing goes unnoticed. You can also use the filter option to quickly find and handle all problems, as described above. Each message is described below:

Additional placeholder. A translation text contains a placeholder that does not occur in the text of the primary culture. This placeholder will likely not get any data at runtime.

Duplicate count/modulo. There are multiple quantified text items for the same combination of count and modulo. They must be unique or an arbitrary instance will be selected at runtime.

Inconsistent punctuation. A translation text has a different spacing at the beginning or end, or a different punctuation at the end than the text of the primary culture. This may lead to inconsistent layout or misunderstandings in the user interface.

Invalid count. A negative count value was entered or the count value for a quantified text is missing. Don't use quantified texts if there's nothing to quantify.

Invalid modulo. A negative modulo or a modulo of 0 or 1 or greater than 1000 was entered.

Missing placeholder. In any translation text, at least one of the placeholders used in the primary culture is missing. The user will probably see less information when using that language. Each placeholder name is counted only once, no matter how often it occurs in a single text.

Missing referenced key. A translation text is referencing another text key that does not exist in the dictionary.

Missing translation. For at least one language, no translation text was entered. If a non-region-specific culture is available, this is what is checked. Additional region-specific cultures are optional in this case. If only region-specific cultures are available for a language, all of them must have a translation text set.

Referenced key loop. A translation text is referencing its own text key or another text key that eventually leads to a referencing loop. The actual translation text for this key (or any other key along the loop) cannot be correctly determined at runtime.

3.8. Suggestions

Another feature to support the user in creating more consistent translations is suggestions. Based on the text of the primary culture, other texts with similar words are searched and listed along with their translation for the currently selected culture. This allows you to easily find other instances of the same word and use the same translation for it in the entire dictionary. The suggestions are displayed in a separate area below or next to the editor area, depending on your settings. It can be displayed and hidden with the **Suggestions** button in the View section of the toolbar.

Translations for the suggested similar text keys are only displayed if you focus an input field of another culture than the primary culture. In that case, only similar texts are displayed but no translations because they are obviously the same anyway.

All results are sorted by relevance, beginning with the most relevant match. The relevance of an item is computed by finding other texts that contain the same words, case-insensitive but otherwise as-spelled. Common stop words for a number of built-in languages are ignored. The longer the text, the less is the value of a single matched word. This is a rather basic approach to find similar texts, don't expect recognition of slightly different word forms and synonyms or any further intelligence like on popular web search engines for now.

3.9. Keyboard shortcuts

For improved productivity, the most important functions can be quickly accessed by keyboard shortcuts (hotkeys). The shortcuts for functions represented by a toolbar button are displayed in their tooltip. Other shortcuts may be less obvious.

3.9.1. Global shortcuts

Ctrl+F	Select the search filter input field

Ctrl+N	Add a new text key
Ctrl+O	Open another dictionary file
Ctrl+Shift+O	Open dictionary files from a folder
Ctrl+S	Save the currently loaded dictionary
Ctrl+Shift+T	Open the text key wizard dialog (only when in a Visual Studio window)
Ctrl+Up	Select the previous text key from the tree, keeping input focus
Ctrl+Down	Select the next text key from the tree, keeping input focus
Alt+Left	Navigate back to the previous view in history
Alt+Right	Navigate forward to the next view in history
Alt+Up	Navigate to the previous visible text key in the tree (keeping culture focus)
Alt+Down	Navigate to the next visible text key in the tree (keeping culture focus)

3.9.2. In the text key tree

Delete	Delete the selected text key(s)
F2	Rename the selected text key
Ctrl+C	Copy the selected text key name(s) to the clipboard
Ctrl+D	Duplicate the selected text key

3.9.3. In the search filter input field

Escape	Clear the search filter and select the tree view

3.9.4. In a translation input field

F12	Navigate to the definition of the text key under the cursor
Ctrl+0	Add a quantified text with the count 0
Ctrl+1	Add a quantified text with the count 1
Ctrl++ (Plus)	Add a quantified text with no count preset
Ctrl+- (Minus)	Delete the selected quantified text

Text key wizard

Often you need to translate many texts in an already existing application, when localisation must later be added to an application. It can be a lot of work to copy all the texts, add a new text key for each, paste the original text into TxEditor and finally write the code to access the new text key. This process is largely automated by the text key wizard. It is currently targeted to Microsoft Visual Studio only and sits in the background until you press a hotkey. The workflow is the following:

1. Start TxEditor and open the application's dictionary to work on.
2. Select the literal string in your source code in Visual Studio.
   - In C# files, include the double quotes around the string because you need to remove them as well when inserting the `Tx.T` method call. If the string is assembled from multiple strings containing data from variables or other functions, try to separate them first.
   - In XAML files, you first need to move any text from the element content into the corresponding property attribute. I. e. write `<TextBlock Text="abc"/>` instead of `<TextBlock>abc</TextBlock>`. then do not include the double quotes around the string because you need to keep them when inserting the `Tx:T` binding.
3. Press the hotkey **Ctrl+Shift+T**.
4. The TxEditor text key wizard dialog appears in the bottom right corner of the screen (configurable). It allows you to select the source file language (selection is remembered but not automatically recog-

nised) and to enter a new text key for the text you just selected. The selected text is also copied to the clipboard and already set in the text field where you can further edit it to clean up any source remains. If the selected text is surrounded by double quotes, these are removed and backslash escape sequences are automatically resolved.

- If the selected text already appears in an existing text key, the text key is preset and selected. Press the backspace or delete key to clear it and see the next suggested text key. Start typing something else to discard any key name suggestions and define your text key.
- If the wizard was used before, the previously defined text key is preset and the last segment is selected just like when renaming a text key. This allows you to either overwrite the last segment and create a new text key on the same level, or move the cursor to the end and create a new text key one level below.

5. For XAML files, you have the option to set the selected text as Default attribute in the Tx binding so that you still see the original text at design time. Otherwise, the text key will be displayed instead.
6. Confirm the dialog by pressing the **Save text** button or the Enter key (not in the translation text input field). The new text item is saved to the dictionary for the primary culture. No other cultures are supported in this workflow, you need to add translations afterwards back in TxEditor.
7. The required C# or XAML code is put in the clipboard which is automatically pasted back in the Visual Studio window, replacing the text you've selected before. You can now modify the generated code to set the placeholder data or count value, if necessary.

Don't forget to **save both**, your source code files and the dictionary, from time to time to avoid data loss in case anything goes wrong.

Copying and pasting the selected text in Visual Studio by TxEditor works by sending generated keystrokes to the other window. This fails if Visual Studio runs in a higher isolation level than TxEditor. Also, the wizard hotkey won't work from within Visual Studio in this situation. In the rare case that you started Visual Studio as administrator, you need to either start it as a regular user this time, or start TxEditor as administrator as well.

4. Advanced topics

4.1. Multithreading

All TxLib methods are threadsafe. The Tx class, which, at the core of TxLib, is used by all other supporting classes, maintains a static dictionary with all texts for all loaded cultures. Any reading access to it uses a reader lock. Whenever the dictionary itself is modified, a writer lock is acquired to lock out all other reading and writing access to it.

The culture selection always depends on the current thread culture as already provided by the .NET framework, so multiple threads are independent of each other.

4.2. Logging

Tx has some logging capabilities to track any issues with text key lookups, placeholder resolution or key usage. The logging is generally controlled by the static **LogFileName** property of the Tx class. It is set to null by default, which means that logging is entirely disabled. This is the recommended setting for production code because it doesn't generate any data or decrease performance. Setting this property to a full file name (or anything that can be resolved from the current working directory), all subsequent log events will be written to that file. The special value "" (an empty string) enables logging to the Trace output which can be monitored with the Visual Studio debugger or other tools that can read data written by the OutputDebugString function. Existing log files will be appended to. The directory must already exist. Some events are logged with a stack trace that helps you find the place in your part of the source code that caused the problem.

The logging behaviour can also be controlled with environment variables. The LogFileName property can be initialised to a different value to activate Tx logging in the target environment without modifying the source code or other configuration files. The environment variable for this is named **TX_LOG_FILE** and must be defined in the current user profile. Both, a regular file name and an empty string as described above, are supported.

The additional environment variable **TX_LOG_UNUSED** controls writing a list of all unused text keys when the application ends. This helps in finding unused text keys that may be left over from refactorings. This is only enabled if logging is enabled in general.

These environment variables are evaluated in the static class constructor of the Tx class. That means that they are evaluated before any other application code is executed and changes to these variables from the application itself will only be considered the next time the application starts.

5. Text key best practices

Basically you are entirely free to use whatever text keys you like in your dictionaries. But to keep some order and to guide new users to a more structured approach, there are some guidelines and best practices for text key management.

5.1. Structure

You should not see the text keys as a flat list of entries with a more or less common naming scheme, but instead consider structuring the keys according to their use in the application. Only the most common and general texts and phrases can be stored with a short key in the "root" scope. Anything more special should have a context to group the entries in. Think of the text keys as a folders and files hierarchy like the one you use to organise your documents or pictures.

TxEditor supports this structure by displaying it in a tree view. At first, only the top-level "folders" are visible and you can then dive in through the contexts to the actual text key by expanding the tree nodes.

By convention, the scope separator character is a **point** (.). Use it to separate the different scope segments in a text key.

Here are some examples for an application that has a toolbar with different sections, a status bar displaying the current status and some message dialogs for user input.

toolbar.file.new	New
toolbar.file.open	Open
toolbar.file.save	Save
toolbar.edit.copy	Copy
toolbar.edit.paste	Paste
statusbar.file saved	File saved.
statusbar.no file loaded	No file loaded.
statusbar.error loading file	There was an error loading the specified file. {details}
message.user name.caption	User name
message.user name.enter user name	Please enter your name:
buttons.ok	OK
buttons.cancel	Cancel

After all, even if TxEditor displays the keys in a tree, a text key is always the full string as it's displayed at the right side of the editor. This full key is also the string that you need to use in your code to refer to a specific text. Knowing this, a part of a text key can also be a text key, so "toolbar.file" and "toolbar.file.new" can perfectly coexist. Sometimes it's even useful to do that, like when defining an explanatory description for some text key by appending ".desc" to it.

A special case of a scope is a namespace. It can be used together with the `Tx.SafeText` method. Namespaces use the **colon** (:) as separator character and cannot be further structured. A special namespace called "Tx" is also used by TxLib formatting methods to define number or time formats, quotation marks etc. Within a namespace, the normal structuring with points can be used.

5.2. Naming text keys

In general you can use any character you want to when naming text keys. For practical reasons, and by the structuring conventions described above, you should however comply with some guidelines for text key names.

Prefer lower case. In some situations, upper-case characters may help to understand text keys, and text keys are generally case-sensitive. But for the most texts, it should be easiest to just stick to lower-case characters for the entire text key. It's also easier to type and remember.

Points and colons have structuring meaning. As described in the previous section, the use of points (.) and colons (:) adds structuring meaning to the text keys and is used in some methods as well as the TxEditor tool.

Do not use quotation marks. You'll be using the text keys in different source code languages and contexts and usually need to specify them as a string. Quotation marks within the text key only make it a lot harder to use the keys later and reduce code readability. So better don't use any kind of single or double quotation marks or their typographical equivalents.

Do not use certain special characters. In XAML files, the characters ampersand (&), less-than (<) and greater-than (>) need to be replaced by an escape sequence. In C# code, the backslash (\) character is an escape character. In Tx placeholder and reference resolution, the characters hash (#), equals (=) and curly-braces ({ and }) have a special meaning and cannot be used. Any type of control character should be avoided as well.

Use spaces. The plain space character is explicitly allowed and in fact preferred over the common underline (_) when you want to express multiple words in a key. Spaces are more natural and can more easily be read. Spaces at the beginning and end of a key will be trimmed though.

Try to use letters, digits and spaces only. Considering the previous points, try to restrict your key names to letters, digits and spaces only. If it doesn't cause problems with your development team, non-ASCII letters like accented, decorated, Cyrillic, Greek or other characters of your familiar writing system are just fine.

Use a simplified form of the primary culture text. One of your main tasks using TxLib will be maintaining, remembering and using text keys. To make it easier, you should use key names that you can easily remember or from which you can easily infer what they're supposed to used for. When you read your application code, you mostly see text keys instead of translated texts, so you should be able to know what a text key basically stands for. Try to use a simplified form of the text in the primary culture as the text key, like in the example above. Using multiple words is fine (use spaces as usual), as long as the key doesn't get too long. Try to find an abstract alternative in this case.

Use the unquantified word form. When defining translations for subject counts, the easiest method is to define a text for the most general form of all plural counts, and add exceptions for those counts that need them in your language. Most of the time, you'll probably only adding exceptions for a count of 1 or maybe also 0. Give the text key a name of the general plural form then. Example: time.days is generally translated as "days", except for 1 where it's "day".