INTERNATIONALIZATION

View online: https://www.construct.net/en/make-games/manuals/construct-3/plugin-reference/internationalization

The **Internationalization** plugin provides tools to manage the localization of a project. This includes facilitating translations, pluralization, formatting dates for the user's locale, and so on.

See the built-in example project for a thorough example showing the various internationalization features.

Internationalization is sometimes written as the shorthand i18n, referring to the fact the word starts with an I, ends with an N, and has 18 other letters in between.

Scripting

When using JavaScript or TypeScript coding, the string lookup features of this plugin can be accessed via the Internationalization script interface. Note however many other features of this plugin merely access the browser-provided Intl APIs, so those are not duplicated in the script interface, as you can access them directly instead.

Locales

Locales - which specify a region, language or dialect - are specified using standardized BCP 47 language tag, also known as just a *language tag*. For example `en-US` refers to US English, `en-GB` refers to British English, `pt-BR` refers to Brazilian Portuguese, and so on.

The translation file

The plugin can be loaded with translation information through event sheets, but the most common use case is to have all the strings for a given language in a separate file and load that using the *Load from JSON* action. A bare-bones example of such file would be as follows:

```
{
	"locale": "en-GB",
	"strings": {
		"foo": "example localized string",
		"bar": {
			"baz": "more localized text"
		}
	}
}
```

A language file is a JSON file, with two distinct keys.

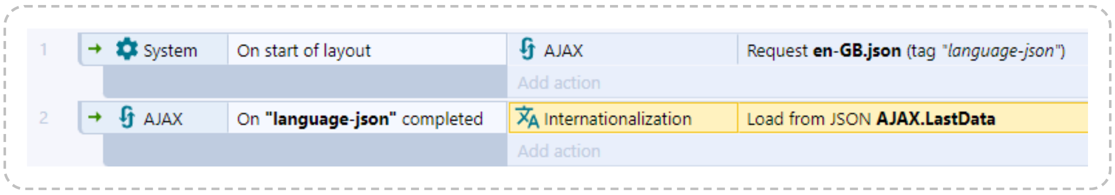
The `"strings"` key is mandatory, all the localized text should be nested inside it. It can contain any valid JSON - how it is organized is entirely up to the user to decide what is more convenient. Trying to load a file without this key will result in an error.

The `"locale"` key is optional and is used when loading to indicate to what language the file corresponds to. If the key is not present then the content of the file will be associated with the locale the Internationalization plugin is currently set to. If the key contains a value which can not be parsed into a valid locale, an error will be thrown when loading.

Loading an Internationalization file

To load an internationalization file do the following:

1. Use the AJAX plugin to request the file.
2. In the AJAX *On Complete* trigger, use the *Load from JSON* action of the Internationalization plugin to load the AJAX data.



1 System | On start of layout | AJAX | Request **en-GB.json** (tag *"language-json"*)
Add action

2 AJAX | On **"language-json"** completed | Internationalization | Load from JSON **AJAX.LastData**
Add action

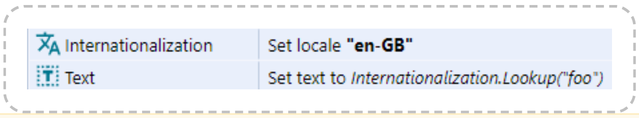
Following those steps it is possible to load a localization file for each locale that needs it.

*When loading files like in the example above, the file needs to have the **"locale"** key so the plugin can know to which locale the data belongs to without any further event blocks.*

Looking up localized strings

Once the plugin has the data for the required locales, the next step is to look up the strings to use them where needed. To do that do the following:

1. Use the *Set locale* action
2. Use the *Lookup* expression with the appropriate path



Internationalization | Set locale **"en-GB"**
Text | Set text to *Internationalization.Lookup("foo")*

*If the example localization file at the beginning of this document was loaded, that look up would yield the string **"example localized string"**.*

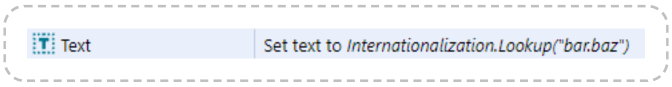
Nesting, Push and Pop

In the case of having a few strings that need to be localized it can be easy enough to just have all of them at the root of the **"strings"** key in the localization file. As the amount of text that needs localization increases it can be useful to group related strings together to keep the localization files tidy.

To make the lookup for a nested string you can do two things:

Absolute path

Include the absolute path to the string you are looking for in the argument of the *Lookup* expression. Dots can be used to navigate to nested JSON keys.

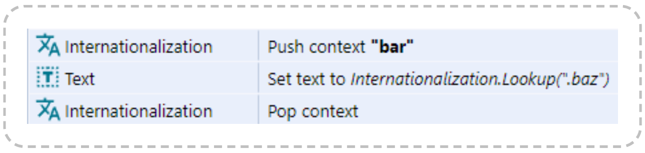


Text	Set text to *Internationalization.Lookup("bar.baz")*

This option is straight forward and can be convenient in isolated cases.

Relative path

Push the context where the strings you are looking for are in the localization file and then use relative paths when using the *Lookup* expression. A relative path begins with a dot, and is appended to the current context.



Internationalization	Push context **"bar"**
Text	Set text to *Internationalization.Lookup(".baz")*
Internationalization	Pop context

In the case of having many strings in the same context it can be useful to use relative paths to avoid duplication.

When using relative paths always remember to have a corresponding *Pop context* action for each *Push context* action.

The previous examples assume the example localization file at the beginning of the document has been loaded.

Note that absolute paths never have a preceding "." and that relative paths always have a preceding "."

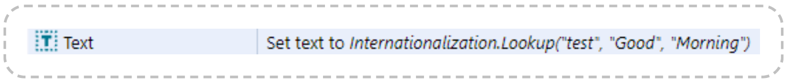
Substitutions

By using substitutions it is possible to produce dynamic results at runtime, take the following example:

```
{
        "locale": "en-GB",
        "strings": {
                "test": "Hello {0} {1}"
        }
}
```

The string has two placeholders that can be replaced at runtime if additional values are provided when using the *Lookup* expression. There can be any amount of placeholders and the number in between the curly braces refers to the order of the arguments passed to *Lookup*.

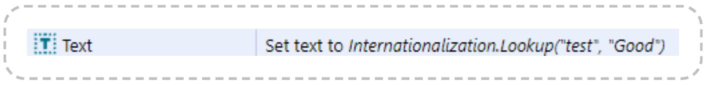
Example 1


Text | Set text to *Internationalization.Lookup("test", "Good", "Morning")*

The output of that example is the string "**Hello Good Morning**".

If there are more placeholders than arguments, like in the example bellow, then the placeholders with no matching argument are left unchanged in the final result

Example 2


Text | Set text to *Internationalization.Lookup("test", "Good")*

The output of that example is the string *"**Hello Good {1}**"*.

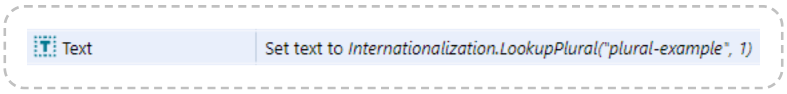
Plural rules

Following is a simple example of a localization file set up with a plural rule for British English.

```
{
        "locale": "en-GB",
        "strings": {
                "plural-example": {
                        "one": "{0} Pig",
                        "other": "{0} Pigs"
                }
        }
}
```

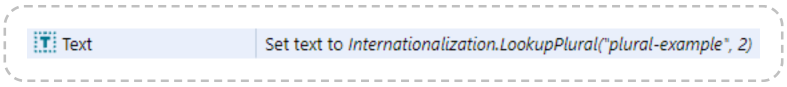
In order to look up the plural form the *LookupPlural* expression is used passing in an additional value which is used to decide which plural form will be used.

Example 1


Text | Set text to *Internationalization.LookupPlural("plural-example", 1)*

That example would yield the string **"1 Pig"**.

Example 2



Text | Set text to *Internationalization.LookupPlural("plural-example", 2)*

That example would yield the string **"2 Pigs"**.

*The placeholder **{0}** in plural strings is always substituted by the second argument of the expression. The placeholder is optional - if it is removed from the localization string the correct plural form will still be picked.*

LookupPlural can also have any amount of additional placeholders for substitution just like Lookup

Pluralization in other languages other than English

English is a relatively simple language when it comes to pluralization, as there are only two plural categories: **"one"** and **"other"** which refer to single units and more than one (or zero) respectively.

Other languages might be more complex in this regard. To find out how many categories and the name each category has, use the following expressions to print out the values and see what keys your localization files need.

PluralCategoryCount will return the total amount of categories for the current locale.

PluralCategoryAt(index) will return the name of the plural category with the provided zero-based index.

Make sure to use the Set locale action with the locale you want that information before using the expressions.

As an example, the locale **"ar-EG"** (Egypt Arabic) has six different plural categories, "few", "many", "one", "two", "zero" and "other". So a localization file for this language which uses plural rules should provide translations for each category.

Internationalization conditions

Compare to current locale

Test the passed in locale value against the locale the plugin is currently configured with.

Internationalization actions

Set locale

Set the locale the plugin will use.

Set context

Set a context to get strings from. An absolute context overwrites any existing context and a relative context is appended to the existing one.

Push context

Push a context to get strings from. This helps to avoid duplication in the case of having to look up multiple strings in the same context.

Pop context

Remove the existing context from the stack.

Add string

Dynamically add a localized string to the provided context in the localization data for the current locale.

Load from JSON

Load the plugin with localization data for the current locale or the locale defined in the passed in JSON.

Internationalization expressions

Locale

Returns the locale string the plugin is currently using.

DefaultLocale

Return the default locale for the current system. This can be used to select the default language.

Lookup(Context [, ...])

Looks up a localized string based on the current locale and a passed in context. Supports a variable list of arguments for substitution.

LookupPlural(Context, Count [, ...])

Looks up the plural form of a localized string based on the current locale a passed in context and a number to decide which plural form to return. Supports a variable list of arguments for substitution.

CurrentContext

Return the current context of the plugin.

SelectPlural(number)

Returns the name of the plural form based on the current locale and the passed in number.

PluralCategoryCount

Returns the total amount of plural categories for the current locale.

PluralCategoryAt(index)

Returns the plural category name based on the current locale and the provided zero-based index.

SaveToJSON

Returns a JSON representation of the plugin internal state.

FormatNumberAsDecimal(number)

Format the passed in number as a decimal based on the current locale.

FormatNumberAsPercent(number)

Format the passed in number as a percent based on the current locale.

FormatNumberAsCurrency(Number, Currency, CurrencyDisplay)

Format the passed in number as a currency in the current locale. The **currency** argument must be a valid 3 letter ISO currency code - see this table for supported currencies. If an unsupported currency is passed in, a warning will be printed to console indicating all the supported currencies. The **currencyDisplay** argument can be any of *"symbol"*, *"narrowSymbol"*, *"code"*, *"name"*. Using an unsupported value will default to *"symbol"*.

FormatNumberWithUnit(Number, Unit, UnitDisplay)

Format the passed number as a unit in the current locale. The **unit** argument can take any of the values defined in this table. The **unitDisplay** argument can be any of *"long"*, *"short"* or *"narrow"*. Using an unsupported value will default to *"short"*.

RegionName(RegionLocale)

Returns the region name of the passed in locale in the language of the current locale of the plugin.

LanguageName(LanguageLocale)

Returns the language name of the passed in locale in the language of the current locale of the plugin.

CurrencyName(CurrencyCode)

Returns the currency name of the passed 3 letter ISO currency code, in the language of the current locale of the plugin.