Kentico 8.1

Macro expressions

Macros are text expressions that the system evaluates and converts into values. By using macros, you can create dynamic website content and in general configure the application to perform different actions under different conditions. Macros range from simple expressions for loading values from data in the system, to complex sets of instructions that lead to a certain result.

Learn the macro syntax

Kentico uses a macro language (K#) with a wide range of syntax options and features.

Enter macro expressions in the system

Learn where in the Kentico user interface you can enter macro expressions, and about the available assistance features.

Create macro conditions

One of the most common ways to leverage macros is to define dynamic conditions for various types of functionality.

- Build conditions using a text-based interface (macro rules)
- Create custom macro rules

Find information about macro methods

The system provides a large number of methods that you can call in macro expressions. View this reference to learn about the available methods.

Troubleshoot macros

If you encounter problems when working with macro expressions, the system provides several tools that can help identify and fix the issues:

- Search the system for macro expressions
- Debug macros
- Solve problems with macro signatures

Extend the macro engine

If the default functionality of the macro engine does not cover all of your requirements, developers can extend the macro engine.

Learn to create custom:

- Macro methods
- Macro fields (properties)
- Macro namespaces

Macro syntax

The system identifies macros using special parentheses. You need to enclose macro expressions into curly brackets and the percentage symbol: ***{% expression %}***

Kentico provides an object-oriented language named **K#**, which defines the syntax of macro expressions. The macro language is very similar to the C# programming language. This page provides an overview of the features available in K# and describes the differences from C#.

The K# syntax is ***case-insensitive*** — the system does not differentiate between upper and lower case letters in commands. However, letter case may have an effect in the values of constants and variables (for the purposes of comparisons and similar).

What is the # character at the end of macros?

When you save a macro expression, the system automatically adds the # character before the closing parentheses. The character indicates that the macro is signed.

See Working with macro signatures for more information.

In addition to the primary macro syntax, the system also supports several special macro types:

Special macro type	Description
Query string macros	To load the values of query string parameters from the URL, use macros in format: ***{? parameter ?}*** Query string macros support all K# syntax. The names of all available query string parameters automatically work as variables that store the value of the corresponding parameter. For example, on pages with URLs like */Home.aspx?nodeid=10*, *{? nodeid ?}* resolves into 10. Alternatively, you can get the values of query string parameters inside standard macros: *{% QueryString.parameter %}*
Localization strings	To add localization strings into text values, use macros in format: ***{ $ ResourceStringKey $}*** The localization macro resolves into the string version that matches the currently selected language. See also: Localizing text fields Localization macros do NOT support K# syntax, or any expressions except for resource string keys. To localize strings inside standard macros: *{% GetResourceString("ResourceStringKey") %}*
Cookie macros (standardized)	To load values from the current user's browser cookies, use the following standard macro: *{% Cookies["CookieName"] %}* For example: *The code of the currently selected language is: {% Cookies["CMSPreferredCulture"] %}*
Path expression macros (standardized)	To resolve page path expressions, use the following standard macro: *{% Path["path"] %}* For example, you can enter the following into an SQL where condition field: *NodeAliasPath LIKE {% Path["../%"] %}* (the macro's result matches the path of all pages under the current page's parent)

The macro engine no longer supports custom macros in format *{# expression #}*. If you are upgrading from an older version of Kentico, and your data contains custom macros, the system automatically converts all occurrences to *{% ProcessCustomMacro("expression", "parameters") %}* to ensure backward compatibility.

We strongly recommend implementing all custom macro functionality using the approaches described in Extending the macro engine.

Values and objects

When the system resolves a macro expression, the result is an object (value). Macros support the following types of objects:

- Standard scalar C# types (int, double, string, DateTime etc.).
- Kentico system types (objects representing pages, users etc.).
- Collections based on the *IEnumerable* interface, which contain various types of objects. For example, *InfoObjectCollection* stores sets of Kentico objects.
- Context objects containing data related to the currently processed request (information about the current user, the currently viewed page, etc.). For example: *CMSContext*, *SiteContext*, *DocumentContext*, *ECommerceContext*.
- Macro namespaces that allow you to access other properties or methods (a namespace object itself does not return any data).

To work with fixed values (literals) in macro expressions:

- **numbers** - type the numbers directly. You can use integers or numbers with decimal points (double type).
- **text** - enclose text (string) values into quotes, for example: *"administrator"*
- **booleans** - use the *true* and *false* keywords.

Operators

K# uses the same basic operators as C#. See the Reference of C# operators to learn more.

The following table summarizes the differences in operator behavior that you need to keep in mind when writing macros.

Operator	Examples	Description
x.y	CurrentUser.UserName	Accesses the members (methods, properties) of macro objects or namespaces.
+	10+5 CurrentPageInfo.DocumentPageTitle + " | suffix"	Adds two operands together. Adding is supported for numbers, and the following types: • **String + String**: Returns the concatenation as a *String* • **DateTime + TimeSpan**: Adds the *TimeSpan* to the *DateTime* and returns the result as *DateTime* • **TimeSpan + TimeSpan**: Returns the sum as a *TimeSpan* String concatenation is the default option if none of the combinations above are detected — the operator returns the concatenation of the operands' *String* representations. For example, *{% "string" + 5 %}* returns **string5**.
-	-10 10-5	Unary - returns the numeric negation of a number. Binary - subtracts the second operand from the first. In addition to numeric types, you can subtract objects of the following types: • **DateTime - TimeSpan**: Subtracts the *TimeSpan* from the *DateTime* and returns the result as *DateTime* • **DateTime - DateTime**: Subtracts the second *DateTime* from the first, and returns the difference as a *TimeSpan* • **TimeSpan - TimeSpan**: Returns the difference as a *TimeSpan*
== !=	50 == 5*10 CurrentUser.UserName != "administrator" CurrentDocument == Documents["/Services/WebDesign"]	Operators that check for equality or inequality of the operands. Return a boolean value. Equality checks support all available object types, with the following special rules: • Empty strings are equal to *null* • Simple data types are equal to their string representation • *Info* objects are equal to string constants that match the object's display name or code name • Two *Info* objects are equal when they have the same object type and ID • GUID values are equal to the string representation of the GUID (always case insensitive) • Enumeration values are equal to constants of the enum's underlying type (integer or string) **Tip**: Use macro parameters to specify case sensitivity and the culture context in equality checks.

< <= > >=	CurrentPageInfo.DocumentPublishFrom <= DateTime.Now	Comparison operators that return boolean values: • **<** (true if the left operand is lesser than the right) • **<=** (true if the left operand is lesser than or equal to the right) • **>** (true if the left operand is greater than the right) • **>=** (true if the left operand is greater than or equal to the right) You can compare objects of the following types: • numeric (Int, Double) • String (comparison based on lexicographical order) • DateTime • TimeSpan **Note**: When comparing strings, use macro parameters to specify case sensitivity and the culture context.
mod	5 mod 2	Performs the modulo operation — computes the remainder after division of two integer operands.
%	30%	In K#, the % character represents percentage values. You **cannot** use the % operator for the modulo operation. Adding the percentage sign converts the preceding number to a double equivalent (multiplies the number by 0.01). For example, *{% 30% %}* returns **0.3**.
??	CurrentDocument.Children.FirstItem ?? "No child pages"	Null-coalescing operator. Returns the left operand if the operand is not *null*, otherwise returns the right operand. **Note**: Empty strings are not considered as null values by the operator.

Macro methods

Methods allow you to perform tasks (execute code) inside macro expressions. You typically call methods with at least one argument.

The recommended K# syntax for method calls is **infix** notation for the first argument. Prefix notation is also supported. For example:

```
// Returns "WORD"
{% "word".ToUpper() %}

// Returns "The sky is red on red planets"
{% "The sky is blue on blue planets".Replace("blue", "red") %}

// Alternative method calls with prefix notation (not supported by the autocomplete
help)
{% ToUpper("word") %}
{% Replace("The sky is blue on blue planets", "blue", "red") %}
```

Kentico provides an extensive set of default methods that you can use in macro expressions. See: Reference - Macro methods

Developers can also extend the macro engine and register custom macro methods.

Compound expressions and declaring variables

K# allows you to write compound expressions, containing any number of simple macro expressions. You need to terminate each expressions (except for the last) using a semicolon. The overall result of a compound expression is the result of the last expression.

Variables allow you to store and manipulate values inside macro expressions. You do not need to explicitly declare the type of variables.

Example

```
// returns "12"
{% x = 5; x + 7 %}

// returns "10"
{% x = 5; y = 3; x += 2; x + y %}
```

The scope of variables spans from the point of declaration to the end of the area containing the macro (such as text fields, e-mail templates or Text / XML transformations), including all separate macro expressions in the given area.

Conditional statements

Use the **if** command to create conditions in format: ***if (<condition>) {<expressions>}***. The condition statement returns the value of *<expressions>* if the condition is true, and a *null* value if false.

```
// returns "z is less than 3"
{% z = 1; if (z<3) {"z is less than 3"} %}
```

To create conditions with a result for the false branch, use the following syntax: ***if (<condition>) {<expressions 1>} else {<expressions 2>}***

```
// returns "z is greater than or equal to 3"
{% z = 5; if (z<3) {"z is less than 3"} else {"z is greater than or equal to 3"} %}
```

The **ternary operator** allows you to create compact conditions with a different result for each branch. Use the following syntax: ***<condition> ? <expressions 1> : <expressions 2>***

If the condition is true, the statement returns the value of *<expressions 1>*. In the case of a false condition, the return value is *<expressions 2>*.

```
// returns "The second parameter is greater"
{% x=1; y=2; x > y ? "The first parameter is greater" : "The second parameter is
greater" %}
```

Open conditions and loops

When defining conditions or loops, you can leave the body of the loop/condition open and close it later in another macro expression. This allows you to apply the command to text content or HTML code placed between the macro expressions. Open commands can be particularly useful in macro-based transformations or various types of HTML templates.

You can also nest additional macro expressions inside open loops or conditions.

Example

```
// Displays a message including the current date if the year is 2014 or more
{% date = CurrentDateTime; if (date.Year > 2013) { %}
The current date is: {% date %}. The registration period has ended.
{% } %}
```

Iteration (loops)

K# provides several types of loop commands:

- **while** (<condition>) {<executed expressions>}

- **for** (<init expression>; <condition>; <increment expression>) {<executed expressions>}
- **foreach** (<variable> in <enumerable object>) {<executed expressions>}

If a loop command is the last expression in a macro, the return value is a concatenated list containing the results of the sub-expressions executed by all iterations of the loop.

While loop example

```
// returns "10"
{% z = 1; while (z<10) {++z}; z %}

// returns "2 3 4 5 6 7 8 9 10"
{% z = 1; while (z<10) {++z} %}
```

For loop example

```
// returns "5"
{% z = 0; for (i = 0; i < 5; i++) { z += 1 }; z %}

// returns "1 2 3 4 5"
{% z = 0; for (i = 0; i < 5; i++) { z += 1 } %}
```

Foreach example

```
// returns "HELLO"
{% z = ""; foreach (x in "hello") {z += x.toupper()}; z %}

// returns "H HE HEL HELL HELLO"
{% z = ""; foreach (x in "hello") {z += x.toupper()}%}
```

Use the **break** command to terminate loops. For nested loops, the command closes the innermost loop.

```
// returns "1 2 3 4 5"
{% z = 0; while (z < 10) {if (z > 4) {break}; ++z} %}
```

The **continue** command skips to the next iteration of the current loop.

```
// returns "0 1 2 4 5"
{% for (i=0; i<=5 ; i++) {if (i == 3) {continue}; i} %}
```

Return command

Use the **return** command to terminate the processing of a macro, and set the attached expression as the macro's final result. You can use the return command inside loops, or anywhere in compound macro expressions.

Example

```
// returns green
{% "red"; "yellow"; return "green"; "blue" %}

// returns "ignore the loop"
{% z = ""; foreach (x in "hello") {return "ignore the loop"; z += x } %}
```

Console output

Console output allows you to build the results of macro expressions without declaring variables. Use the **print(<expressions>)** or **println(<expressions>)** syntax. Each console output expression adds to the macro's return value, and the system continues processing the macro.

Console output has higher priority than the standard result of macro expressions, but lower than the **return** command.

Example

```
// returns "123"
{% i = 1; while (i < 4) {print(i++)}; "ignored" %}

// returns "result"
{% i = 1; while (i < 4) {print(i++)}; return "result" %}
```

Tip: To explicitly set the current console output as the return value of a macro, use the plain **return** command without an attached expression.

Indexing

K# supports indexing for collections and objects that serve as containers for other data:

- **DataRow**, **DataRowView**, **DataRowContainer** (returns the value at the specified index)
- **DataTableContainer** (returns the row at the specified index)
- **DataSetContainer** (returns the table at the specified index)
- **String** (returns the character at the specified index)
- **IEnumerable** collections, such as *InfoObjectCollection* (returns the object at the specified index in the collection)

Example

```
// returns "e"
{% "hello"[1] %}

// returns the value of the FirstName column from the DataRow
{% dataRow["FirstName"] %}
```

Comments

To add explanatory text inside macro expressions, use one-line, multi-line or inline comments.

Example

```
{%
// This is a one-line comment. Initiated by two forward slashes, spans across one
full line.

/*
This is a multi-line comment.
Opened by a forward slash-asterisk sequence and closed with the asterisk-forward
slash sequence.
Can span across any number of lines.
*/

x = 5; y = 3; /* This is an inline comment nested in the middle of an expression.
*/ x+= 2; x + y
%}
```

Lambda expressions

Lambda expressions are ad-hoc declarations of inline functions that you can use inside macro expressions.

Example

```
// returns "4"
{% lambdaSucc = (x => x + 1); lambdaSucc(3) %}

// returns "6"
{% lambdaMultiply = ((x, y) => x * y); lambdaMultiply(2,3) %}
```

The scope of lambda expressions spans from the point of declaration to the end of the area containing the macro (such as text fields, e-mail templates or Text / XML transformations), including all separate macro expressions in the given area.

Macro parameters

Macro parameters allow you to modify how the system resolves individual expressions. To append parameters to macro expressions, use the following syntax:

```
{% ...|(<parameter>)<value> %}
```

You can add multiple parameters to a single expression. Use backslashes to escape the | separator in parameter values if necessary.

The following macro parameters are available:

Parameter	Example	Description
default	`{% SKUPrice|(default)N\\\|A %}`	Sets a default value that the macro returns if the result of the expression is an empty value.
encode	`{% NewsSummary|(encode)true %}`	Enables HTML encoding for the result of the macro (converts reserved HTML characters to equivalent character entities).
notrecursive	`{% NewsText|(encode)true|(notrecursive)true %}`	If true, the system does not recursively resolve macro expressions contained in the macro's result.
culture	`{% SKUPrice|(culture)en-us %}`	Sets the culture (language) used when formatting numbers and dates in the macro result.
casesensitive	`{% Contains("term", NewsText)|(casesensitive)true %}`	Determines whether string comparisons and other operations inside the macro expression are case sensitive. False by default. You can enable case sensitivity globally for all expressions by adding the following key to the *appSettings* section of your web.config file: `<add key="CMSMacrosCaseSensitiveComparison" value="true">`

timeout	`{% GetDocumentUrl()|(timeout)2000 %}`	Sets the maximum time allowed to resolve the expression (in milliseconds). If unspecified, the default timeout is 1000 ms. If the timeout is reached, the system aborts the resolving process. The macro returns a null value and an entry is logged in the event log.
handlesqlinjection	`{% QueryString.Param|(handlesqlinjection)true %}`	If true, the system replaces single quote characters (') in the macro result with two single quotes ('').
debug	`{% Documents["/News"].Children.WithAllData|(debug)true %}`	Enables detailed macro debugging (only for the given expression).

Entering macro expressions

The Kentico user interface provides several features that help you:

- Build the macro expressions that you need
- Enter macros into the system

Using the macro autocomplete help

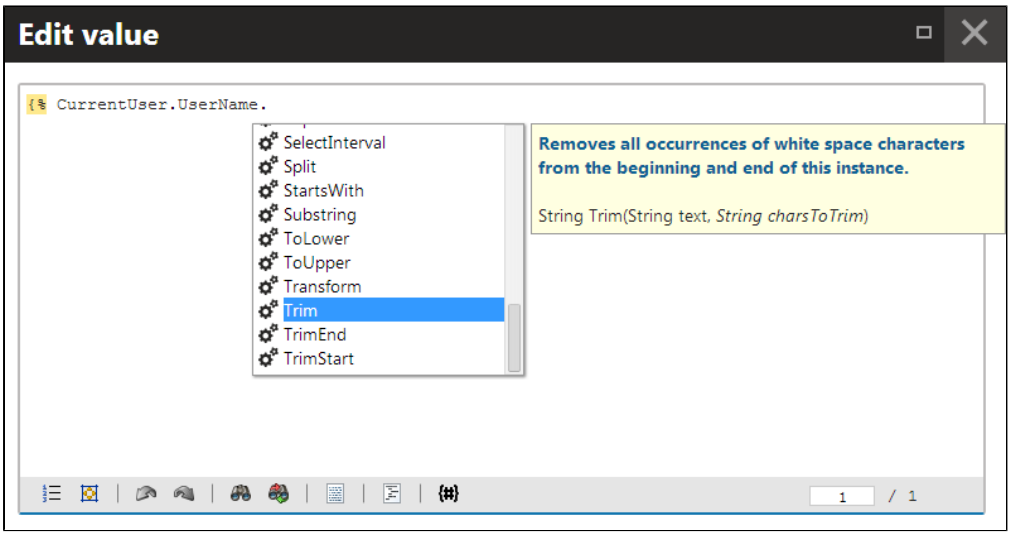
When typing in dedicated macro editing fields, the system offers an autocomplete assistance feature. The autocomplete also works if you write macro expressions in other locations, for example when editing e-mail templates or **Text / XML** transformations.

> **Note**: ASCX type transformations do NOT support macro expressions.

The autocomplete is similar to IntelliSense in Visual Studio — as you type, a box with the available macro methods and fields appears next to the cursor. If you continue typing, the autocomplete filters the list to include only items that contain the given text. You can navigate through the listed macros using the up and down arrow keys or the mouse. Once you select the appropriate item, press the **Enter**, **Space** or **Dot** key to insert the macro into the text.

When accessing members of macro objects using the dot operator, the autocomplete shows a list of available methods and properties.

If you select a macro method in the autocomplete list, the help displays the method's description and signature (the return type and parameters). *Italics* identify method parameters that are optional.



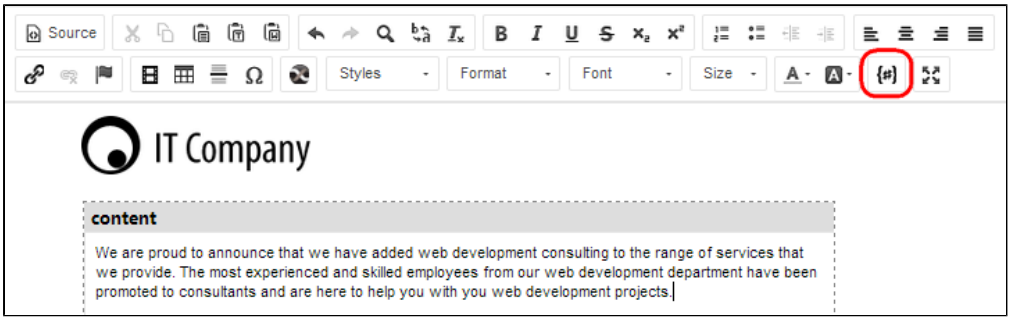
When creating macros within a specialized context, the autocomplete shows related macro fields in a high priority section. You can access prioritized macros at the top of the list, separated by a horizontal line.

> **Tip**: To manually open the macro autocomplete, press **Ctrl + Shift + Space**.

Inserting macros into text and code fields

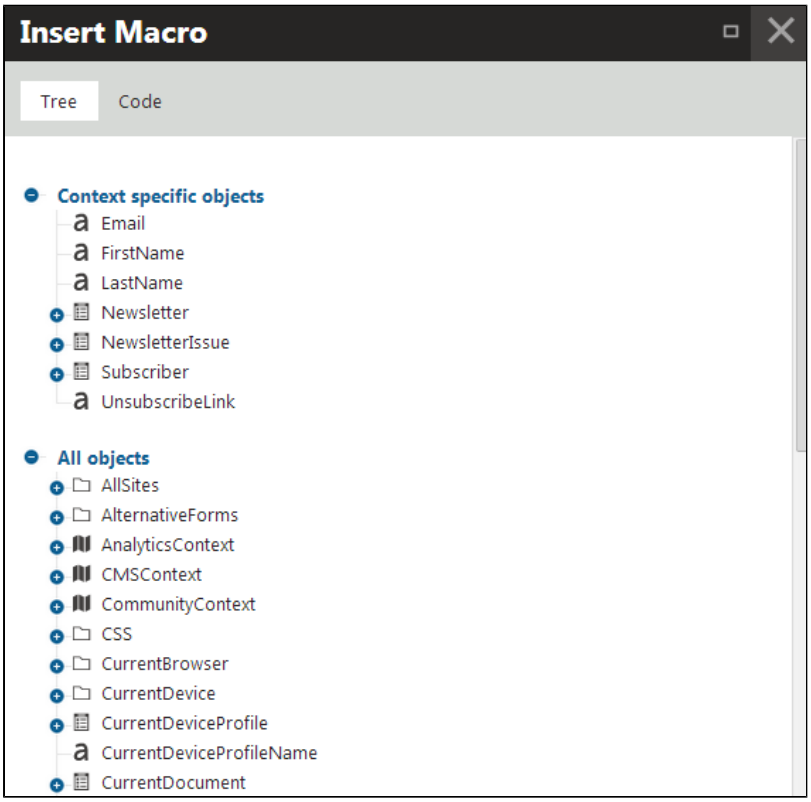
The system provides a shortcut for inserting macro expressions in the WYSIWYG editor, macro editors, and other types of code editors that support macros. For example, the feature is available for editable text regions of pages, or when editing e-mail templates and newsletter issues.

Click **Insert macro** ({#}) on the toolbar above or below the editing area.



The **Insert Macro** dialog opens. You can explore a tree of all available macro options. Click on an item in the tree to insert the given macro into the edited area.

If you wish to write the macro expression directly (with autocomplete support), switch to the **Code** tab, type the macro code and click **Insert**.

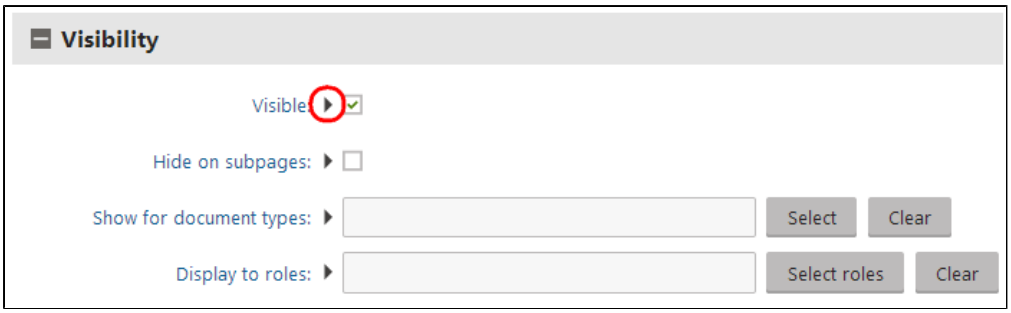


The macro appears at the current position of the cursor in the edited area, enclosed within the *{% %}* macro parentheses.

Adding macro values into web part properties

You can use macros to insert dynamic values into the properties of web part instances. The system evaluates the macros when rendering the page containing the web part.

All web part properties support macro expressions. For properties with text values, type the required macros directly into the value. To assign macro values to other properties (such as checkboxes or lists of options), click the ▶ icon next to the given property.



The **Edit value** dialog opens, where you can write the required macro.

For example, the *{% CurrentUser.CheckPrivilegeLevel(UserPrivilegeLevelEnum.GlobalAdmin) %}* macro returns a true or false value depending on if the current user has the Global administration privilege level. When you add the macro into the **Visible** property, only global administrators can see the web part's content on the live site.

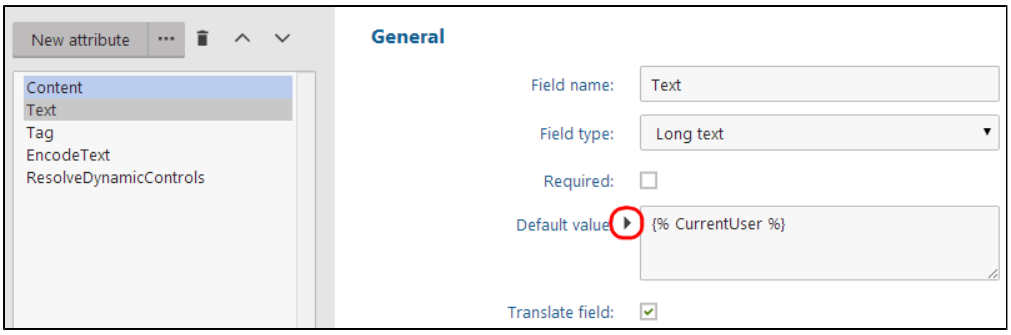
> **Note**
>
> By default, the **WhereCondition** and **OrderBy** properties of web parts are secured against SQL injection attacks. If you insert a macro that returns a string value into the property, the system escapes single quote characters ('), and replaces them by two single quotes (''). This may lead to SQL syntax errors.
>
> Use the **handlesqlinjection** macro parameter to control single quote escaping in macro results. See Macros and security - SQL injections for more information.

Setting default form field values through macros

When defining form fields through the field editor, you can use macros to set dynamic default values (for both visible and hidden form fields).

Use one of the following approaches, depending on when you want the system to resolve the macro:

- Place the macro directly into the **Default value** of text-based fields. When users view the resulting form, the macro appears ***unresolved*** in the field. If the user leaves the expression in the field's value, the system resolves the macro when processing the submitted form.
- Click **Edit value** (▶) next to the **Default value** and write the macro in the Edit value dialog. In this case, the system ***resolves the macro directly when users open the form***.



> **Note**
>
> When editing fields for web part properties, form control parameters, or the properties of UI page templates, the resolving of macros added into default field values via the **Edit value** (▶) dialog depends on the **Resolve default value** flag.

Using macros as default values for web part properties

Setting default values for web part properties is a common example of the macro functionality in form fields.

1. Open the **Web parts** application.
2. Use the tree to select the web part for which you want to set the default property value.
3. Open the **Properties** tab.
4. Select the required property and enter the macro into the **Default value**.
5. Click **Save**.

The system resolves the macro depending on how you entered the expression into the default value:

- If you placed the macro directly into the default value of a text property or have the **Resolve default value** check box disabled, the expression remains unresolved in the configuration dialog when users add new instances of the web part. The system resolves the

macro when rendering the page containing the web part instance (as long as the user leaves the expression in the property's value).

- If you added the macro through the **Edit value** ( ) dialog and enable **Resolve default value**, the system resolves the macro directly when opening the web part properties dialog.

Writing macro conditions

Macro conditions greatly increase the flexibility of certain features. By preparing conditions, you allow the system to dynamically make decisions. For example, when to perform an action, under which circumstances to make an item visible, and so on.

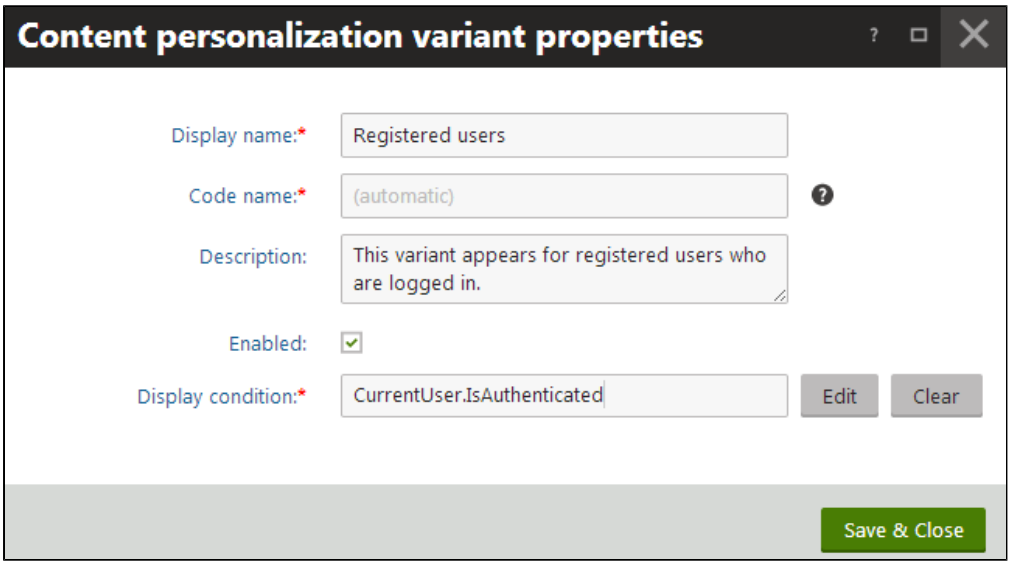
Each condition is defined by a macro expression that returns a true or false value. When the system evaluates the condition, the macro is resolved according to the currently available context data, and the result determines whether the condition is fulfilled or not.

For illustration, the following are examples of functionality that uses macro conditions:

- Content personalization variants (display conditions)
- Form fields (visibility and enabled conditions)
- Marketing automation processes (triggers and step transition conditions)
- Advanced workflow steps (step transition and branching conditions)

Entering macro conditions

The user interface provides dedicated fields for entering macro conditions. For example, content personalization variants have a **Display condition** field.



You can choose between two ways of creating macro conditions:

a) Build the condition out of macro rules

Macro rules are predefined, text-based clauses that define certain requirements. You form the overall condition by combining any number of macro rules. Using macro rules does not require knowledge of K# syntax.

To work with macro rules, click **Edit** next to the condition field. See Building conditions using macro rules for more information.

b) Write the code of the condition directly

Write a K# macro expression into the condition field. The result must be a boolean value (true/false). Use macro methods with a boolean return type, or comparison and equality operators.

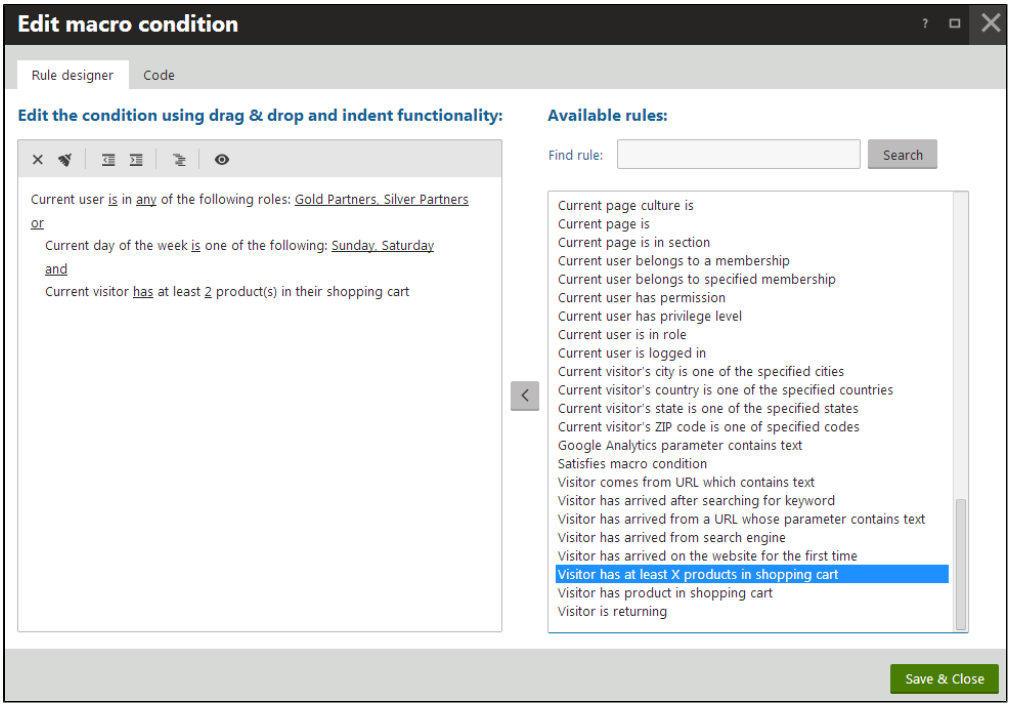
Do not enclose the expression into the standard macro parentheses — the system automatically processes the content of the condition as a K# macro. You can leverage the macro autocomplete help when typing in condition fields.

Tip: If you wish to add a macro condition field into a custom form, use the following form control: **Condition builder**

Building conditions using macro rules

Macro rules allow you to create macro conditions without any knowledge of the Kentico macro language (K#). Instead of writing code, you build conditions out of text-based clauses. Each macro rule represents a requirement, which can either be true or false at the time when the system evaluates the condition.

To create a condition using macro rules, click **Edit** next to a condition field in the user interface. The **Rule designer** dialog opens.



Adding rules to conditions

1. Select the rule in the list of **Available rules**.

   > **Tip**: Use the filter above the list to find the rules that you need.

2. Click **Add rule** ( ).

The rule appears in the main designer area, displayed as a text description of the corresponding requirements.

Setting parameters for rules

Parameters allow you to configure the exact requirements of macro rules. For example, in a rule that requires the current user to be a member of certain roles, you select the roles through a parameter.

Macro rules display parameters as underlined sections within the rule text. To set the value of a parameter, click the underlined text and fill in the value inside a separate dialog.

Combining multiple rules together

If you add multiple rules to a condition, the rule designer automatically places operators between the rules.

- The **and** operator means that the combination of two rules is only true if *both* rules are also true.
- If two rules are connected using the **or** operator, the result is true if *at least one* of the rules is fulfilled.

Click on operators to switch between the two options. The final result of the condition depends on the results of individual rules, and the operators placed between them.

The rule designer uses *indentation* to group together macro rules and set the precedence of the operators. The system first evaluates the rules on the level with the largest indentation. The result is passed to the level above, and this process continues until the final result of the condition is known.

To increase or decrease the indentation, select a rule and click **Indent** or **Unindent** on the toolbar above the designer area. Clicking **Auto indent** modifies the indentation of all rules in the condition — ensures precedence for all groups of rules connected through the **and** operator .

You can change the order of the rules by dragging them to different positions in the designer area.

Removing rules from the condition

You can delete rules by clicking buttons on the toolbar above the designer area:

- **Delete** - removes the currently selected rule

- **Clear all rules** - removes all rules from the condition

Editing the macro code of conditions

You can view the K# code of the overall condition defined by the added macro rules. Click **View condition in K#** ( ) on the toolbar.

If you wish to manually edit the code, switch to the dialog's **Code** tab.

> **Note**: The system cannot convert general macro code into macro rules. If you edit a condition on the **Code** tab, the content of the rule designer is deleted.

Creating macro rules

Macro rules allow non-technical users to create macro conditions without any knowledge of K# macro syntax. The rules are internally implemented as predefined macro expressions, but appear as purely text-based clauses describing certain requirements. Users build conditions out of macro rules inside a dedicated *Rule designer* interface.

The system contains many different types of rules by default, and allows you to create custom rules to fulfill any requirements of your users. There are several categories of macro rules:

Rule category	Description	Managed in
Global	Global rules are available when creating conditions in any part of the system. **Note**: Certain types of conditions that are closely related to individual applications do not offer global rules.	**Macro rules -> Global**
Form validation	Allow the system to validate input values of form fields. Users can select the validation rules when defining fields in the field editor (general fields) and the form builder.	**Macro rules -> Form validation**
Workflow	Users can select workflow rules when adding conditions to workflow scopes or advanced workflow steps.	**Workflows -> Macro rules**
On-line marketing	Users can select on-line marketing rules when building conditions for: • Content personalization variants • Dynamic contact groups • Campaigns • Triggers and step transitions for marketing automation processes • Personas	**Contact management -> Configuration -> Macro rules**
Reporting	Users can select reporting rules when specifying conditions for Report subscriptions.	Open the **Reporting** application, select any report category in the tree and open the **Macro rules** tab (in the collapsible panel).
E-commerce	The system provides rules for creating conditions that limit discounts: • Order discounts • Catalog discounts • Free shipping offers	**Store configuration** or **Multistore configuration -> Discount rules**

Defining macro rules

To prepare new macro rules for your users:

1. Navigate to the management interface for the appropriate type of rule (see the table above).
2. Click **New macro rule**.
3. Fill in the following properties:

General

Display name	Sets the name displayed to users in the list of macro rules in the rule designer. Also serves as a basic description of the rule's purpose.
Name	Serves as a unique identifier of the rule (for example in the API).
Description	A text description of the rule's purpose and parameters. The text appears as a tooltip when users hover over the rule in the rule designer.
Enabled	If disabled, users cannot select the rule when building conditions. **Note**: Disabling a rule does not affect the functionality of existing conditions where users have added the given rule.
Rule data	
User text	Defines the text displayed in the rule designer when users insert the rule into a condition. To add parameters into the text, enter the *Column name* of a specific parameter enclosed in curly brackets, for example **{days}**. In the rule designer, the parameter appears as an underlined section in the text. Users can set the parameter's value by clicking the underlined text. See the Adding rule parameters section for more information.
Condition	The actual condition represented by the rule. Define the condition through standard macro code (K#). The field provides autocomplete support. Add any parameters offered by the rule to the appropriate position in the code, using the same syntax as for the **User text** field, for example **{days}**. When the system resolves the condition, the parameter expression is replaced by the value set for the parameter by users in the rule designer.
Required data	Limits for which conditions the macro rule is available. The rule only appears when building conditions that have the specified data items available in the resolving context. You can add multiple required data items separated by semicolons (;). Leave this property empty unless you are creating rules specifically for conditions with a customized resolving context.
Requires context	Enable this property if the rule's condition needs to access context data (information about the current user, the currently viewed page, etc.) in order to work correctly. This ensures that the rule only appears for conditions that have the context available when they are resolved. For example, the context is accessible when evaluating the conditions of Content personalization variants on the website's page, but not when the system is building dynamic Contact groups.

4. Click **Save**.

Adding rule parameters

Parameters are variables inside macro rules that modify the resulting condition. By adding parameters, you can make flexible rules that are usable for a wide range of scenarios. For example, in a macro rule that requires the current user to be a member of certain roles, the names of the roles would be a parameter.

To create a parameter for a macro rule:

1. Edit the macro rule.
2. Open the **Parameters** tab.
3. Prepare the parameter as a form field. Every parameter has a certain data type and various other settings. The interface that users see when editing the parameter's values depends on the selected Form control.

Refer to Reference - Field editor for details about working with the field editor.

4. Click **Save**.
5. Switch to the rule's **General** tab.
6. Place the parameter into the rule's **User text** and **Condition** code.
    - The syntax for inserting parameters is the parameter's *Column name* enclosed in curly brackets, for example ***{role}***.
    - When the system resolves conditions containing the macro rule, the parameter expressions in the condition code are replaced by the values entered by users. You need to position the parameter expression accordingly.
7. Click **Save**.

Users can now set the values of the rule's parameters when building conditions.

Automatic rule parameters

There are several predefined parameters you can use to add common functionality to macro rules:

- **{_is}**, **{_has}**, **{was}**, **{_will}**, **{_perfectum}** - provide an easy way to negate a rule's condition. Each variant offers different wording, so you can use the one that matches the text of your rule. The parameters allow users to choose between a positive and negative option, for example *is* or *is not*. If the negative option is selected, the parameter resolves into the K# negation operator "!" in the condition code. With the positive option, the parameter returns an empty string.

- **{_any}** - useful for rules where a list of items needs to be specified through another parameter. The parameter allows users to switch between two options that determine how the rule processes the item list — *any* (at least one of the items must meet the given condition) or *all* (the condition must be fulfilled for all items in the list). When resolved, the parameter returns either a *false* (any) or *tr ue* (all) value. In the condition code, you can insert the parameter as an additional argument of macro methods that work with object lists, which automatically ensures the required functionality. For example: *CurrentUser.IsInRole("{roles}", {_any})*

When you enter one of the automatic parameters into the **User text** field of a rule, the system automatically creates a field with the corresponding configuration and default values on the **Parameters** tab.

> **Escaping special characters in text parameters**
>
> Certain characters inside the values of text parameters could cause the macro condition of rules to become invalid if unhandled. For example, if a rule's condition is *CurrentUser.UserNickName == "{name}"*, a quote character inside the value of the *name* param eter would prematurely close the string, leading to invalid macro syntax and an incorrect parameter value.
>
> To avoid the problem, the system automatically escapes quote and backslash characters inside parameters entered as string literals (directly enclosed inside quotes).
>
> For more complex parameters, you can explicitly enable escaping of string special characters by adding **|(escapestring)** to the parameter, for example:
> *"Prefix {parameter|(escapestring)}"*

Modifying existing macro rules

When you make changes in the definition of a macro rule or its parameters, the system does NOT update the functionality of existing conditions. This may lead to incorrect behavior of conditions containing the modified rule.

To make sure your conditions are up-to-date, you need to manually perform the following steps after modifying an existing macro rule:

1. Find the condition fields that contain the rule. You can use the macro report tool to search.
2. Click **Edit** next to the field to open the rule designer.
3. If required, update the parameters for all occurrences of the modified rule.
4. Click **Save & Close**.

When you save the condition, the system uses the current definitions of the contained macro rules.

Caching the results of macros

You can store the results of macro expressions in the application's cache. Caching allows the system to save resources when processing macros — macros only need to be resolved once, and subsequent occurrences load the result from the application's memory (until the cache expires).

To cache the result of a macro, enclose the expression into the **Cache** method (as the first parameter). Additionally, you can specify the following *optional* parameters for the method:

- **int cacheMinutes** - the number of minutes for which the cache stores the macro's result. The default value is 10 minutes.
- **bool condition** - a boolean condition that must be fulfilled (true) in order to cache the macro's result.
- **string cacheItemName** - the name of the cache key that stores the macro's result. If you set the same cache key name for multiple macros, the macros share the same cached value. The default name includes variables, such as the macro's text (including the sign ature), and the name and culture of the user viewing the page where the macro was resolved.
- **string cacheItemNameParts** - any number of parameters whose values are combined with the *cacheItemName* into the name of the cache key.
- **CMSCacheDependency cacheDependency** - sets dependencies for the cache key (use the *GetCacheDependency* method to get the dependency object).

Example - Basic

The following macro caches the value of the **"string".ToUpper()** expression:

```
{% Cache("string".ToUpper()) %}
```

When the system resolves the macro, the result (STRING) is saved into the cache for 10 minutes. If the macro needs to be resolved again while the cache is still valid, the system does not process the macro at all, and loads the result directly from the cache.

Tip: You can confirm the caching functionality of macros by viewing the system's cache debugs (cache items and cache access).

Example - Advanced

The following example demonstrates how to specify the optional parameters when caching macros:

```
{% Cache(CurrentUser.GetFormattedUserName(), 5, true, "MacroCacheKey|UserName|" +
CurrentUser.UserName, GetCacheDependency("cms.user|all")) %}
```

- The macro caches the formatted name of the current user for 5 minutes.
- The name of the macro's cache key contains the username, so each user has a separate record in the cache.
- The **cms.user.all** cache dependency ensures that the system automatically clears the cache whenever a user account in the system is modified.

When the system resolves the macro for different users, you can find the corresponding cache keys and their values in the cache items debug.

Macro troubleshooting

If you encounter problems when working with macro expressions, the system provides tools that can help identify and fix the issues:

Debugging macros

The macro debug allows you to analyze how the system resolves macros. If you encounter problems with macros not working correctly, the debug can help you:

- Confirm when and where your macros are being processed
- Identify the exact source of problems
- Detect syntax errors

Enabling the macro debug

To use the macro debug, you need to adjust the settings in **Settings -> System -> Debug**:

Setting	Description
Enable macro debug	Enables macro debugging and the **Macros (K#)** tab in the **Debug** application.
Enable detailed macro debug	If enabled, the macro debug displays the results of all subelements used within macro expressions. This allows you to check the exact data content of a macro's components during each step of the resolving process. Detailed macro debugging is highly recommended. If disabled, the debug only shows the final result of each macro. You can enable the detailed debug only for specific expressions by adding the **|(debug)** parameter to the given macro.
Display macro debug on live site	If enabled, macro debug information is also displayed at the bottom of each live site page. Macro debugging must also be enabled.

Debug macros resolved on UI pages	If enabled, macros resolved on the pages of the administration interface are also included in the macro debug. Macro debugging must also be enabled.
Log macros to file	If enabled, the system saves the macro debug log into the *logmacros.log* file in the *~\App_Data* folder. Macro debugging must also be enabled.
Macro debug log length	Sets the maximum length of the macro debug log, i.e. the number of requests for which the macro debug stores information. If empty, the value of the **Default log length** setting is used.
Display stack information	If enabled, the system tracks the code stack when debugging macros and displays the information in the **Context** column. This information is only available in the debugging UI and on the live site, not in the debug log written into the *logmacros.log* file.

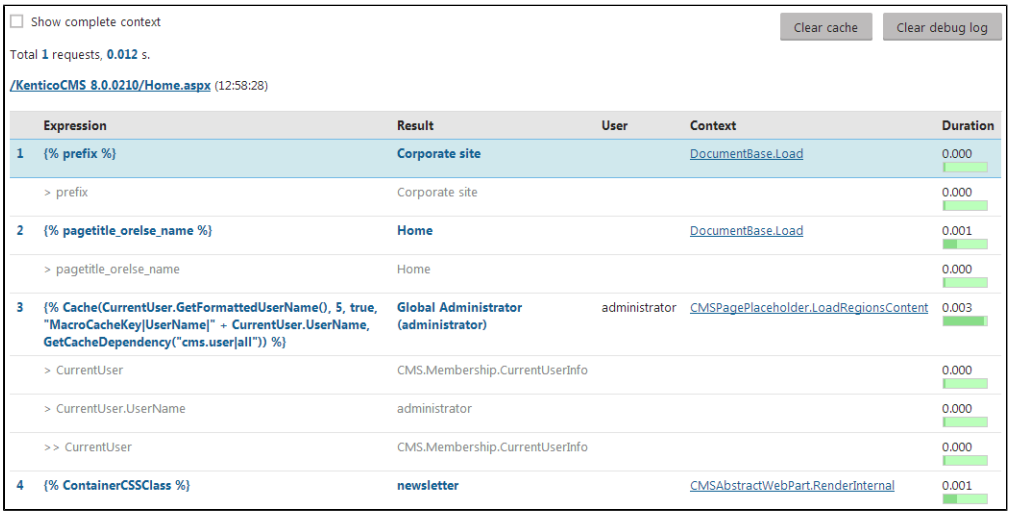
Tip: You can also enable macro debugging through the "debug everything" settings in the **All** section of the **Debug** settings category.

Debugging macros

To access the macro debug, open the **Debug** application and select the **Macros (K#)** tab.

The log displays a list of recent page requests, along with the macros that the system resolved while processing the requests. For each macro, you can see:

- The exact **Expression**
- The **Result** (the value into which the system resolved the macro)
  - If detailed macro debugging is enabled, you can see the results of all sub-expressions that make up the macro.
- The name of the macro's author (the user name in the macro signature)
- The **Context** in which the macro was resolved
  - Click on the method to see the full stack trace.
  - If you enable the **Show complete context** option at the top of the interface, the stack trace is displayed for all macros.
- The **Duration** of the resolving process



When debugging macros, we recommend using two separate tabs or browsers to:

- Adjust the code of your macro
- Execute the macro by loading/refreshing the page that contains the expression or triggers the related functionality

To remove all previously logged macro debug data, click **Clear debug log**.

Working with macro signatures

Whenever a user saves a macro expression, the system automatically adds a security signature. The signature contains the user name of the macro's author and a hash of the given expression. To increase the level of security, the hash function used when creating macro signatures appends a salt to the input (a sequence of additional characters). The salt value depends on the configuration of the application's environment, so the signatures are in most cases not valid when deploying macros to other instances of Kentico.

You can recognize signed macro expressions by the # character, which the system automatically inserts before the closing **%}** parentheses when saving the text containing the macro.

```
{% CurrentUser.Children["cms_category"][0].CategoryName #%}
```

If the execution of the macro requires any permissions, the system resolves the macro only if the **user specified by the signature** has the appropriate permissions.

Permissions are only checked when resolving macro expressions that:

- access objects that are members of another object, for example: *{% CurrentDocument.DocumentPageTemplate %}*
- access collections of objects, for example: *{% CurrentUser.Children["cms_category"][0].CategoryName %}*

The system carries out a security check for each access to any collection, not just for the final macro result.

Unsuccessful security checks prevent the system from resolving macros. If you encounter such problems, you can view the event log or the macro debug log, which provide information about performed security checks and their results.

Tip: You can confirm whether your macros have valid signatures by searching for expressions in **System -> Macros -> Report**.

See: Searching for macros

Disabling the security signature for specific macros

You can stop the system from adding macro signatures by manually adding @ character before the closing %} sequence of a macro:

```
{% CurrentPageInfo.DocumentName @%}
```

These macros do not store the user name of the author or the security hash, so the internal length of the expression does not exceed the visible number of characters. This allows you to safely use macros in fields with a limited character count.

Unsigned macros are always evaluated with the permissions of a public user.

Configuring the hash salt for macro signatures

The system appends a salt to the input of the hash function that creates macro signatures.

By default, new instances of Kentico use a randomly generated GUID as the hash salt. The installer sets the custom salt value through the **CMSHashStringSalt** key in the *appSettings* section of the web.config file. Instances without this web.config key use the application's main database connection string as the salt (the exact **CMSConnectionString** value in the web.config file). We strongly recommend using a custom salt (CMSHashStringSalt).

A custom salt provides the following benefits:

- Stability (you do not need to re-sign macros if your connection string changes)
- You can use the same salt for other instances of Kentico, which makes deployment easier and allows content staging of macros

The best option is to set the hash salt value before you start creating content for your website. Changing the salt causes all current hash values to become invalid (including macro signatures).

To change the salt value for your application, edit the value of the **CMSHashStringSalt** key in your web.config file. You can use any string as the value, but the salt should be random and at least 16 characters long. For example:

```
<add key="CMSHashStringSalt" value="e68b9ad6-a461-4707-8e3e-ece73f03dd02" />
```

Warning: In addition to macro signatures, the system uses the **CMSHashStringSalt** value for other hash functions. Changing the hash salt on a website that already has defined content may break dialog links and images on your website. If you encounter such problems, you need to re-save the given content (the system then creates the hashes using the new salt).

Re-signing macros

If the hash salt value used by your application changes, the security signatures of existing macro expressions become invalid. This may lead to problems with unresolved macros, for example when:

- You have set a new custom salt via the **CMSHashStringSalt** web.config key.

- You are using Content staging to transfer data containing macros to an instance of Kentico with a different hash salt.
- Your application does not have a custom hash salt, and the connection string has changed (for example when moving to a different server or after setting a new database password).

To re-sign individual macros, find the expression in the user interface and save the value (you can use the macro report tool to find the location). The system creates a new signature for the macro based on the application's current environment.

You can also repair invalid macro signatures by re-signing all macros in the system:

1. Go to **System -> Macros -> Signatures**.
2. Fill in the **Old salt** field.

   > If you leave the **Sign all macros** option *disabled*, the system checks the original signatures before re-signing macros. Only macros that have a valid signature under the old salt are re-signed and the user names of the macro authors remain unchanged. You need to enter the old salt that was used to generate the security hash of the existing macro expressions in the system.
   >
   > - If your application uses a custom hash salt, enter the original value of the **CMSHashStringSalt** web.config key.
   > - If the original hash salt was a connection string, enter the value in format: *Persist Security Info=False;database=DBName;server=ServerName;user id=DBUser;password=pwd;Current Language=English;Connection Timeout=240;*

   > If you enable **Sign all macros**, the macro re-signing process skips the signature integrity check and creates new signatures for all macros. This includes macros that are unsigned or have invalid signatures. The new signatures use the name of the user who started the re-signing procedure. You do not need to enter the old salt value in this case.

3. Type in the **New salt** that will be used to re-sign the macros.
   - By default, the field automatically loads the current application's hash salt value. To enter a different value, disable the **Use current salt** option.

     > **Important**
     >
     > In order for the system to correctly validate macro signatures, the new hash salt value must match the application's current salt (the **CMSHashStringSalt** key value or connection string). Only change the **New salt** val ue if you are planning to change your application salt.

4. Click **Update macro signatures**.

The system replaces the security signature in all occurrences of macros based on the new salt.

Searching for macros

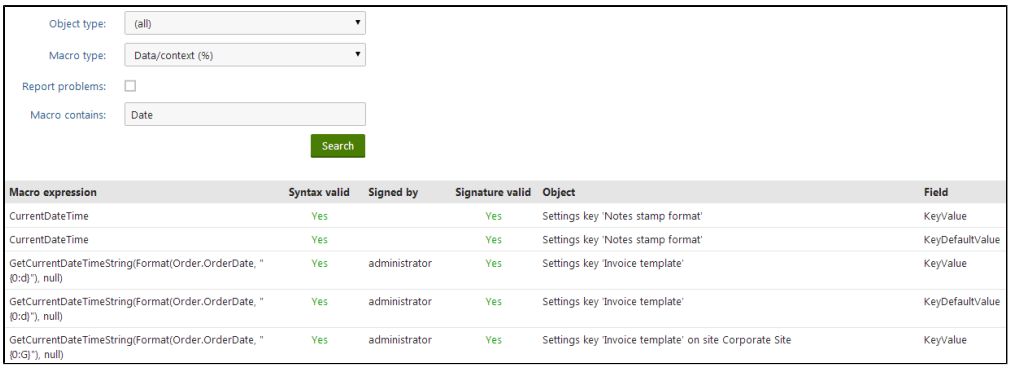
The system provides a report where you can find all occurrences of macro expressions:

1. Open the **System** application.
2. Select the **Macros -> Report** tab.
3. Specify the filtering options to narrow down the list of macros:
   - **Object type** - shows only macros stored in the data of objects of the selected type
   - **Macro type** - switches between *Context* {% ... %}, *Query string* {? ... ?} and *Localization* {$ ... $} macros
   - **Report problems** - if checked, the report only displays macros that contain syntax errors or have invalid security signatures.
   - **Macro contains** - searches for macros that contain the specified text in their expression
4. Click **Search**.

> **Tip**: Use the **Report problems** option to search the system for instances of macros that do not resolve correctly. Such macros can lead to problems on your websites.

The list displays the following information for each macro:

- The text of the macro expression (if the text is a macro rule, hover over the value to view the actual macro code)
- An indicator showing whether the macro is syntactically valid
- The name of the macro's author (the user name in the macro's signature)
- An indicator showing whether the macro's security signature is valid
- The object type and exact field containing the macro



Extending the macro engine

The system allows you to extend the macro engine in several different ways. You can prepare custom macros that provide any functionality required by your users.

Registering custom macro methods

In addition to the default macro methods, you can also create your own methods. Users can then run custom functionality by calling the methods inside macro expressions.

Use the following process to add macro methods into the system:

1. Define the methods inside a container class
2. Register your macro method container for a certain object type or macro namespace

Defining macro methods

1. Open your project in Visual Studio.
2. Create a new class. In *web site* projects, you can either add the class into the **App_Code** folder or as part of a custom assembly.
3. Edit the class and add a reference to the **CMS.MacroEngine** namespace:

```
using CMS.MacroEngine;
```

4. Make the class inherit from **MacroMethodContainer**:

```
/// <summary>
/// Sample MacroMethodContainer class.
/// </summary>
public class CustomMacroMethods : MacroMethodContainer
{

}
```

5. Define your methods inside the container class. Macro methods must always have the following signature:

```
public static object MyMethod(EvaluationContext context, params object[]
parameters)
```

The **EvaluationContext** parameter allows you to get information about the context in which the macro containing the method was resolved. For example, you can check the username in the macro signature (for security purposes) or the values of macro parameters, such as the culture or case sensitivity of string comparisons.

The **parameters** array stores the method's parameters. When the system resolves the method, the values of the arguments pass into the array. You need to define individual parameters via attributes (see below).

Note

We strongly recommend that you ensure the following in the code of your methods:

- **Caching** - if you load data in the method's code, use the Kentico caching API to optimize the performance
- **Security** - you need to handle permissions and other security checks manually in the method's code to avoid security vulnerabilities. You can use the **UserName** property of the method's **EvaluationContext** parameter to get the name of the user who entered the macro.

6. Add the **MacroMethod** attribute above the method declaration. Specify the following parameters for the attribute:

   - **Type** - the return type of the method.
   - **Comment** - comment displayed for the method in the macro autocomplete help.
   - **Minimum parameters** - the minimum number of parameters that must be specified when calling the method (minimum overload).

7. Add a **MacroMethodParam** attribute for each of the method's parameters. For every parameter, you must specify:

   - **Index number** (sets the index of the parameter in the *params* array)
   - **Name**
   - **Data type**
   - **Comment** (appears in the macro autocomplete)

Example

```
using CMS.MacroEngine;
using CMS.Helpers;

public class CustomMacroMethods : MacroMethodContainer
{
 [MacroMethod(typeof(string), "Combines two strings, or appends a culture
suffix when called with one parameter.", 1)]
 [MacroMethodParam(0, "param1", typeof(string), "First part of the string.")]
 [MacroMethodParam(1, "param2", typeof(string), "Second part of the string
(optional).")]
 public static object ConnectStrings(EvaluationContext context, params
object[] parameters)
 {
  // Branches according to the number of the method's parameters
  switch (parameters.Length)
     {
   case 1:
    // Overload with one parameter
    return ValidationHelper.GetString(parameters[0], "") + " - Resolved in
culture: " + context.Culture;

   case 2:
    // Overload with two parameters
    return ValidationHelper.GetString(parameters[0], "") + " - " +
ValidationHelper.GetString(parameters[1], "");

   default:
    // No other overloads are supported
    throw new NotSupportedException();
  }
 }
}
```

Registering macro method containers

After you prepare your macro methods in a container class, register the container by extending an object type or macro namespace.

1. Edit the class containing your macro method container.
2. Add a **RegisterExtension** assembly attribute above the class declaration for each type that you wish to extend (requires a reference to the **CMS** namespace).

Specify the type parameters for the **RegisterExtension** attribute in the following format:

[assembly: RegisterExtension(typeof(<*macro method container class*>), typeof(<*extended type*>))]

You can extend the following types:

- General system types (string, int, ...)
- Kentico API object types (UserInfo, TreeNode, ...)
- Macro namespaces (SystemNamespace, StringNamespace, MathNamespace, ...)
- Custom types

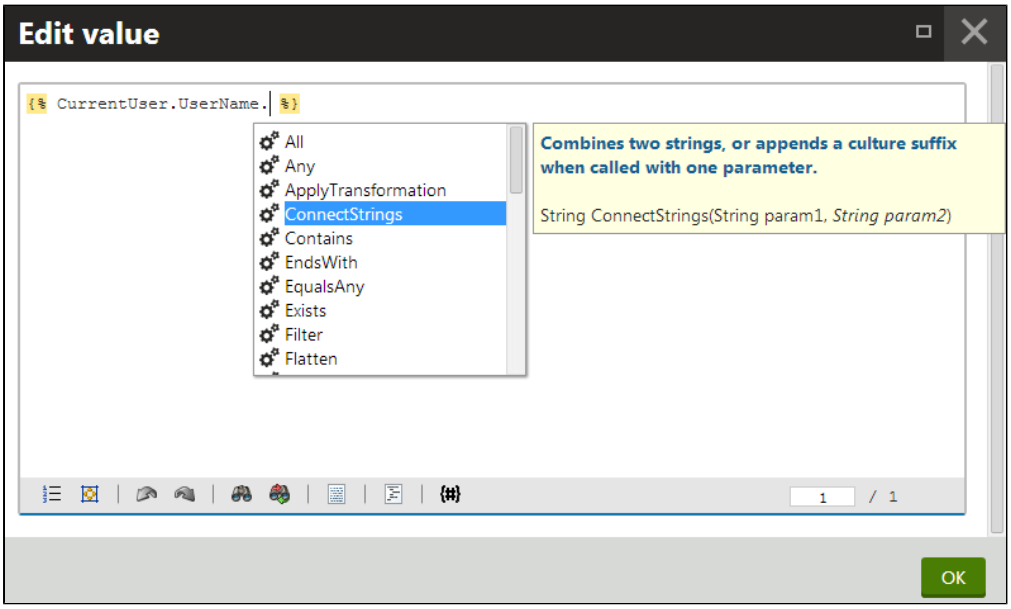
Example

```
using CMS;

using CMS.MacroEngine;
using CMS.Helpers;

// Makes all methods in the 'CustomMacroMethods' container class available for
string objects
[assembly: RegisterExtension(typeof(CustomMacroMethods), typeof(string))]
// Registers methods from the 'CustomMacroMethods' container into the "String"
macro namespace
[assembly: RegisterExtension(typeof(CustomMacroMethods), typeof(StringNamespace))]

public class CustomMacroMethods : MacroMethodContainer
{
...
```

Result - Calling custom methods in macros

Once you have your custom macro methods implemented and registered, you can call them in macro expressions.

Open any macro code editor in the administration interface. The autocomplete help offers the new methods for the extended types or namespaces.



For example:

- Append the method to a macro object of the data type that you extended (*string* in the sample code):

```
{% CurrentUser.UserName.ConnectStrings() %}
```

> The recommended K# syntax is to use infix notation for the first parameter of methods. You can call the method in the following forms:

- **{% "String1".ConnectStrings("String2") %}** - recommended and supported by the autocomplete help
- **{% ConnectStrings("String1", "String2") %}**

OR

- Enter the extended macro namespace, and then call the method:

```
{% String.ConnectStrings("First part", "Second part") %}
```

Macro method code samples

The Kentico installation includes code examples of custom macro method registration, which you can add directly to your web project. To access the samples:

1. Open your Kentico installation directory (by default *C:\Program Files\Kentico\<version>*).
2. Expand the *CodeSamples\App_Code Samples* sub-directory.
3. Copy the **Samples** folder into the **CMS\App_Code** folder of your web project.

> **Web application installations**
>
> If your project was installed in the web application format, copy the samples into the **Old_App_Code** folder instead.
>
> You must also manually include the sample class files into the project:
>
> a. Open your application in Visual Studio.
> b. Expand the **CMSApp_AppCode** project in the Solution Explorer.
> c. Click **Show all files** at the top of the Solution Explorer.
> d. Expand the *Old_App_Code* folder, right-click the new **Samples** sub-folder and select **Include in Project**.

You can find the macro method examples in the following classes:

- **Samples/Macros/CustomMacroMethods.cs** - class containing the sample macro methods.
- **Samples/Modules/SampleMacroModule.cs** - initializes the custom method registration.

Adding custom macro fields

Fields (properties) allow users to load values in macro expressions. Field values can be of any supported type, from simple scalar types (strings, numbers) to objects and collections. The macro engine supports fields both as members of objects or namespaces, and as independent values.

The system provides two different ways to add your own macro fields:

- Register fields into macro resolvers
- Extend existing objects or macro namespaces

Registering fields into macro resolvers

Macro resolvers are system components that ensure the processing of macros. The resolvers are organized in a hierarchy that allows child resolvers to inherit all macro options from the parent. The ***Global resolver*** is the parent of all other resolvers in the system. By adding fields into the global resolver, you create new macros that are available in all parts of the system.

To add custom macro fields, you need to register data sources into resolvers:

- Named sources
- Named callback sources
- Anonymous sources

Named sources

Use named macro sources to directly register objects or scalar values into resolvers. Named sources appear in the macro autocomplete help and macro tree.

Call the **SetNamedSourceData** method for a resolver object, with the following parameters:

- A string that sets the name of the macro field (used in macro syntax).
- The object that the system returns when resolving the field in macro expressions.
- (Optional) By default, the registered fields appear in the high priority section of the autocomplete help and macro tree. To add namespaces with normal priority, add **false** as the third parameter.

Example

```
using CMS.MacroEngine;
using CMS.Membership;

...

// Registers macro fields into the global resolver
MacroContext.GlobalResolver.SetNamedSourceData("MacroField", "RESOLVED Value 1");
MacroContext.GlobalResolver.SetNamedSourceData("MyUser",
UserInfoProvider.GetUserInfo("administrator"));
```

Users can work with the new macros in any part of the system that uses the given macro resolver. If the field's value is an object containing additional fields, you always need to access the fields as members of the object itself. For example:

```
The name of my user in upper case is: {% MyUser.UserName.ToUpper() %}
First eight characters of the MacroField value: {% MacroField.Substring(0, 8) %}
```

Named callback sources

You can define macro fields that use a callback method to get the value. This approach allows you to calculate the field's value in a separate method.

Call the **SetNamedSourceDataCallback** method for a resolver object with the following parameters:

- A string that sets the name of the macro field (used in macro syntax).
- The name of the callback method.
- (Optional) By default, the registered fields appear in the high priority section of the autocomplete help and macro tree. To add namespaces with normal priority, add **false** as the third parameter.

Example

```
using CMS.MacroEngine;

...

// Registers a single macro field, with the value processed using a callback method
MacroContext.GlobalResolver.SetNamedSourceDataCallback("CallbackProperty",
MacroPropertyEvaluator);

...

// Callback method that provides the value of the 'CallbackProperty' macro field
private object MacroPropertyEvaluator(EvaluationContext context)
{
 // Returns the result of the macro.
 // The example is only a demonstration of the basic principles. The method can be
as complex as required.
 return "Macro return value";
}
```

Users can work with the new macros in any part of the system that uses the given macro resolver. For example: *{% CallbackProperty %}*

> **Note**: The system executes the callback for every occurrence of the custom macro. If the evaluator method contains any computationally intensive logic, we recommend implementing a caching mechanism for the result to optimize performance.

Anonymous sources

When you register an object as an anonymous macro source, users can access the object's data fields, but not the object itself.

Call the **SetAnonymousSourceData** method for a resolver object, with the appropriate data object as the parameter (for example a DataRow).

```
using CMS.MacroEngine;

...

// Prepares sample data in a DataRow
DataTable table = new DataTable();
table.Columns.Add("Field1", typeof(string));
table.Columns.Add("Field2", typeof(string));
table.Columns.Add("Field3", typeof(int));

table.Rows.Add("Value1", "Value2", 42);

// Registers the data into the global resolver as an anonymous source
MacroContext.GlobalResolver.AddAnonymousSourceData(table.Rows[0]);
```

In macro expressions, users can access the data fields (columns) inside the anonymous source, without referring to the actual registered object.

```
{% Field1 %}
{% Field2 %}
{% Field3 %}
```

Example - Adding custom macros to the global resolver

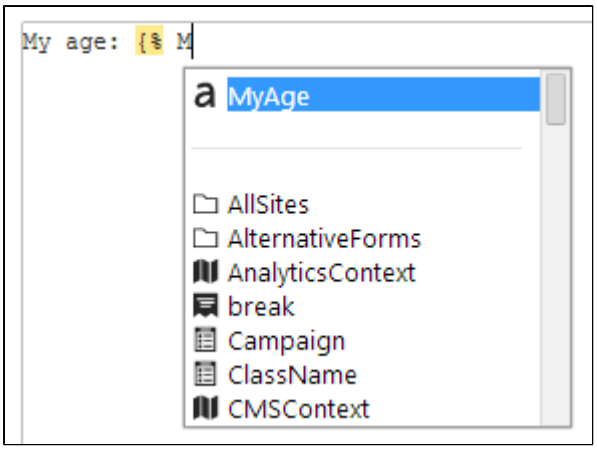
The following example demonstrates how to add a custom field to the global macro resolver. The global resolver is the parent of all other resolvers. Macro fields that you add into the global resolver become available in all parts of the system. The sample macro uses a callback method to calculate the age of the current user.

1. Open your web project in Visual Studio.
2. Create a new class in the **App_Code** folder (or **CMSApp_AppCode -> Old_App_Code** on web application projects).
3. Extend the **CMSModuleLoader** partial class.
4. Create a new class inside *CMSModuleLoader* that inherits from **CMSLoaderAttribute**.
5. Add the attribute defined by the internal class before the definition of the *CMSModuleLoader* partial class.
6. Override the **Init** method inside the attribute class, and call the **SetNamedSourceDataCallback** method for the **GlobalResolver**.
7. Define the callback method.

```
using CMS.Base;
using CMS.MacroEngine;
using CMS.Membership;

[MacroLoader]
public partial class CMSModuleLoader
{
 /// <summary>
 /// Attribute class for registering custom macro extensions.
 /// </summary>
 private class MacroLoaderAttribute : CMSLoaderAttribute
 {
  /// <summary>
  /// Called automatically when the application starts.
  /// </summary>
  public override void Init()
  {
   // Adds the 'MyAge' property to the global macro resolver
   MacroContext.GlobalResolver.SetNamedSourceDataCallback("MyAge",
MyAgeEvaluator);
  }

  // Callback method that defines the result of the 'MyAge' macro property
  private object MyAgeEvaluator(EvaluationContext context)
  {
   // Gets the birth date of the current user
   DateTime dateOfBirth =
MembershipContext.AuthenticatedUser.UserSettings.UserDateOfBirth;

   if (dateOfBirth == DateTime.MinValue)
            {
                return "Date of birth is unknown.";
            }

   DateTime now = DateTime.Now;

   // Calculates the basic difference in years
   int age = now.Year - dateOfBirth.Year;

   // Subtracts one year if the current user's birthday has not occurred yet
this year
   if (dateOfBirth.AddYears(age) > now)
   {
    age--;
   }

   return age;
  }
 }
}
```

8. Save the class. On *web application* installations, build the **CMSApp** project.

You can now try entering the ***{% MyAge %}*** expression in any part of the system where macros are supported. The macro autocomplete help automatically offers the custom property in the high priority section.



Adding custom macro fields to existing object types

The macro engine allows you to add ***static*** fields to existing object types or macro namespaces.

Start by defining your fields inside a container class:

1. Open your project in Visual Studio.
2. Create a new class. In *web site* projects, you can either add the class into the **App_Code** folder or as part of a custom assembly.
3. Make the class inherit from **MacroFieldContainer**.
4. Override the **RegisterFields** method.
5. Call **RegisterField** to define your custom fields.

```
using CMS.MacroEngine;
using CMS.Membership;

public class CustomMacroFields : MacroFieldContainer
{
    protected override void RegisterFields()
    {
        base.RegisterFields();

  // Defines a custom macro field in the container
  RegisterField(new MacroField("MyUser", () =>
UserInfoProvider.GetUserInfo("administrator")));
 }
}
```

The *RegisterFields* method accepts a **MacroField** parameter. Specify the following information for each macro field:

- The field's name (used in macro syntax).
- The value of the field. Define the value using a lambda expression.

Register your macro field container class by extending an object type or macro namespace.

Add a **RegisterExtension** assembly attribute above the class declaration for each type that you wish to extend (requires a reference to the **CMS** namespace).

Specify the type parameters for the **RegisterExtension** attribute in the following format:

[assembly: RegisterExtension(typeof(<*macro field container class*>), typeof(<*extended type*>))]

You can extend the following types:

- General system types (string, int, ...)
- Kentico API object types (UserInfo, TreeNode, ...)
- Macro namespaces (SystemNamespace, StringNamespace, MathNamespace, ...)
- Custom types

Example

```
using CMS;

using CMS.MacroEngine;
using CMS.Membership;

// Registers fields from the 'CustomMacroFields' container into the "System" macro
namespace
[assembly: RegisterExtension(typeof(CustomMacroFields), typeof(SystemNamespace))]

public class CustomMacroFields : MacroFieldContainer
{
...
```

The attribute registers your custom field container class. You can access the field's value in macro expressions as a member of the extended object type or namespace.

Creating macro namespaces

Macro namespaces serve as containers for static macro methods and fields. Users can access the members of namespaces when writing macro expressions, for example *{% Math.Pi %}* or *{% Math.Log(x) %}*. Namespaces also appear in the macro autocomplete help. The system uses several default namespaces such as *Math*, *String* or *Util*, and you can create your own namespaces for custom macros.

To add a custom macro namespace:

1. Create a class inheriting from **MacroNamespace<Namespace type>**. In *web site* projects, you can either add the class into the **App_Code** folder or as part of a custom assembly.
2. Register macro fields or methods into the namespace — add **Extension** attributes to the class, with the types of the appropriate container classes as parameters.

```
using CMS.Base;
using CMS.MacroEngine;

[Extension(typeof(CustomMacroFields))]
[Extension(typeof(CustomMacroMethods))]
public class CustomMacroNamespace : MacroNamespace<CustomMacroNamespace>
{
}
```

See Registering custom macro methods and Adding custom macro fields to learn about creating container classes for macro fields and methods.

Registering macro namespaces

Once you have defined the macro namespace class, you need to register the namespace as a source into a macro resolver (typically the global resolver).

We recommend registering your macro namespaces at the beginning of the application's life cycle. Choose one of the following options:

- During the initialization process of the application itself — use the **CMSModuleLoader** partial class in the **App_Code** folder.
- When initializing custom modules — override the **OnInit** method of the module class.

The following steps describe how to register a macro namespace into the global resolver using the App_Code folder:

1. Create a class file in the **App_Code** folder (or **CMSApp_AppCode -> Old_App_Code** on web application projects).
2. Extend the **CMSModuleLoader** partial class.
3. Create a new class inside *CMSModuleLoader* that inherits from **CMSLoaderAttribute**.
4. Add the attribute defined by the internal class before the definition of the *CMSModuleLoader* partial class.
5. Override the **Init** method inside the attribute class.
6. Call the **SetNamedSourceData** method for the global resolver with the following parameters:

    - A string that sets the visible name of the namespace (used in macro syntax).
    - An instance of your macro namespace class.
    - (Optional) By default, the registered namespace appears in the high priority section of the autocomplete help and macro

tree. To add namespaces with normal priority, add **false** as the third parameter.

```
using CMS.Base;
using CMS.MacroEngine;

[MacroNamespaceLoader]
public partial class CMSModuleLoader
{
    /// <summary>
    /// Attribute class that ensures the registration of custom macro namespaces.
    /// </summary>
    private class MacroNamespaceLoaderAttribute : CMSLoaderAttribute
    {
        /// <summary>
        /// Called automatically when the application starts.
        /// </summary>
        public override void Init()
        {
   // Registers "CustomNamespace" into the macro engine
   MacroContext.GlobalResolver.SetNamedSourceData("CustomNamespace",
CustomMacroNamespace.Instance);
  }
    }
}
```

The system registers your custom macro namespace when the application starts. Users can access the namespace's members when writing macro expressions.

Registering namespaces as anonymous sources

By registering a macro namespace as an anonymous source, you can allow users to access the namespace's members directly without writing the namespace as a prefix. For example, *{% Field %}* instead of *{% Namespace.Field %}*.

```
// Registers "CustomNamespace" as an anonymous macro source
MacroContext.GlobalResolver.AddAnonymousSourceData(CustomMacroNamespace.Instance);
```

You can register the same namespace as both a named and anonymous source. If you only register a namespace as an anonymous source, users cannot access the members using the prefix notation, and the namespace does not appear in the macro autocomplete help.

Using custom macro resolvers

Viewing is restricted - remove the restriction when the page is done!

**

1. Create a resolver instance.
2. Register data sources (fields) into the resolver.

```
using CMS.MacroEngine;



public class MyResolvers : ResolverDefinition
{
 private static MacroResolver mCustomResolver = null;

 public static MacroResolver CustomResolver
 {
  get
  {
   if (mCustomResolver == null)
   {
      // Creates a new macro resolver as a child of the global resolver
      MacroResolver resolver = MacroResolver.GetInstance();

      // Registers macro fields into the resolver
      resolver.SetNamedSourceData("Field1", "Value1"));
      resolver.SetNamedSourceData("Field2", "Value2"));

      mCustomResolver = resolver;
   }

   return mNewsletterResolver;
  }
 }
}



ExtendList<MacroResolverStorage,
MacroResolver>.With("NewsletterResolver").WithLazyInitialization(() =>
NewsletterResolvers.NewsletterResolver);
```

3. *on app init (App_Code / module) -

```
using CMS.Base;
using CMS.MacroEngine;

[MacroLoader]
public partial class CMSModuleLoader
{
    /// <summary>
    /// Attribute class ensuring the registration of custom macro resolvers.
    /// </summary>
    private class MacroLoaderAttribute : CMSLoaderAttribute
    {
        /// <summary>
        /// Called automatically when the application starts.
        /// </summary>
        public override void Init()
        {
   ExtendList<MacroResolverStorage,
MacroResolver>.With("CustomResolver").WithLazyInitialization(() =>
MyResolvers.CustomResolver);
        }
    }
}
```

4. Assign the resolver to macro UI components (editing fields / controls).**

*

> **Note**: You can explicitly set the resolver that the system uses by adding the **|(resolver)** parameter to individual macro expressions.

**

> If you define a static resolver shared across the entire system, always create a new instance when calling the resolver's **ResolveM acros** method. Ensures thread safety.**

Resolving macros using the API

If you need to process macro expressions inside text values in your custom code, use the **MacroResolver.Resolve** method. Specify the string where you want to resolve macros as the method's input parameter.

For example:

```
using CMS.MacroEngine;

...

// Resolves macros in the specified string using a new instance of the global
resolver
string resolvedTextGlobal = MacroResolver.Resolve("The current user is: {%
CurrentUser.UserName %}");
```

> The method evaluates the macros using a new instance of the global resolver and automatically ensures thread-safe processing.
>
> Macro resolvers are system components that provide the processing of macros. The resolvers are organized in a hierarchy that allows child resolvers to inherit all macro options from the parent. The global resolver is the parent of all other resolvers in the system.

Resolving localization macros

If you only need to resolve localization macros in text, call the **ResHelper.LocalizeString** static method.

```
using CMS.Helpers;

...

// Resolves localization macros in text
string localizedResult = ResHelper.LocalizeString("{$general.actiondenied$}");
```

Reference - Macro methods

Macro expressions allow you to use a large variety of methods. The following categories list the methods according to the type of provided functionality:

- Data manipulation
- Data conversion
- Text manipulation
- Advanced text processing
- Date and time modification
- Mathematical operations
- User membership and permissions
- Transformations

Keep in mind that the recommended macro syntax is to use infix notation for the first parameter of methods.

For example: ***{% "word".ToUpper() %}*** instead of ***{% ToUpper("word") %}***

Data manipulation

Method	Return type	Parameters	Description
GetValue	object	• ISimpleDataContainer container • string column • *object defaultValue*	Gets the value of the specified data column of an object that implements the ISimpleDataContainer interface. You can specify a default value used if the requested data is null.
GetProperty	object	• IHierarchicalObject object • string property • *object defaultValue*	Gets the value of the specified property of an IHierarchicalObject object. You can specify a default value used if the requested property is null.
GetItem	object	• IEnumerable collection • int index • *object defaultValue*	Gets the object at the specified index of a collection. You can specify a default value used if the requested object is null.
OrderBy	AbstractObjectCollection	• AbstractObjectCollection collection • string orderBy	Returns the collection of objects, with the order defined by the specified SQL ORDER BY clause.
Where	AbstractObjectCollection	• AbstractObjectCollection collection • string where	Filters a collection of objects according to the specified SQL Where condition.
TopN	AbstractObjectCollection	• AbstractObjectCollection collection • int topn	Returns only the specified number of objects in the collection.

Columns	AbstractObjectCollection	• AbstractObjectCollection collection • string columns	Returns the collection of objects containing only the specified data columns. Separate the columns using commas.
Filter	InfoObjectCollection	• IEnumerable collection • string condition	Filters a collection of objects according to the specified macro condition.
ClassNames	TreeNodeCollection	• TreeNodeCollection collection • string classNames	Filters a TreeNodeCollection of pages according to the specified page types. The second parameter must contain a list of allowed page type code names, separated by semicolons.
InList	bool	• object object • IEnumerable collection	Returns a true value if the object exists within the specified collection.
All	bool	• IEnumerable collection • string condition	Returns a true value if all of the objects in the collection match the given condition. Specify the condition as a macro expression.
Any	bool	• IEnumerable collection • *string condition*	Returns a true value if at least one object in the collection matches the given condition. Specify the condition as a macro expression. If you leave out the condition parameter, the method returns true if the collection contains at least one object. Throws an exception if the collection is null.
Exists	bool	• IEnumerable collection • *string condition*	Returns a true value if at least one object in the collection matches the given condition. Specify the condition as a macro expression. If you leave out the condition parameter, the method returns true if the collection contains at least one object. Returns false if the collection is null.
RandomSelection	IList	• IEnumerable items • *int numberOfItems*	Returns randomly selected objects from the collection. You can optionally specify the number of items.
SelectInterval	IList	• IEnumerable items • int lowerBound • int upperBound	Returns the specified interval of objects from the collection.
Cache	object	• object expression • *int cacheMinutes* • *bool condition* • *string cacheItemName* • *string cacheItemNameParts* • *CMSCacheDependency cacheDependency*	Evaluates the specified macro expressions and stores the result in the server-side application cache. The system only evaluates the expression if the result is not found in the cache. See also: Caching the results of macros

GetCacheDependency	CMSCacheDependency	• string dependencies	Returns a CMSCacheDependency object based on the specified dummy cache keys. See also: • Setting cache dependencies • Caching the results of macros

> Back to the list of macro method categories

Data conversion

Method	Return type	Parameters	Description
ToString	string	• object value • *string defaultValue* • *string culture* • *string format*	Converts an object to a string. Returns the optional default value if the conversion is not possible. You can also specify the culture context and a formatting string used for the conversion.
ToInt	int	• object value • *int defaultValue*	Converts an object to an integer number. Returns the optional default value if the conversion is not possible.
ToBool	bool	• object value • *bool defaultValue*	Converts an object to a boolean value. Returns the optional default value if the conversion is not possible.
ToDouble	double	• object value • *double defaultValue* • *string culture*	Converts an object to a double number. Returns the optional default value if the conversion is not possible. You can also specify a culture code to determine the formatting of the decimal number.
ToGuid	guid	• object value • *Guid defaultValue*	Converts an object to a GUID value. Returns the optional default value if the conversion is not possible.
ToDateTime	DateTime	• object value • *DateTime defaultValue* • *string culture*	Converts an object to a DateTime value. Returns the optional default value if the conversion is not possible. You can also specify a culture code to determine the date format.
FromOADate	DateTime	• double value	Converts a double representation of a date and time value (OLE Automation Date) to a DateTime object.
ToTimeSpan	TimeSpan	• object value	Converts an object to a TimeSpan value. Returns null if the conversion is not possible.
ToBaseInfo	BaseInfo	• object value • *BaseInfo defaultValue*	Converts an object to a BaseInfo — the general type for Kentico system objects and pages. Returns the optional default value if the conversion is not possible.

List	ArrayList	• object items	Converts a list of objects to an ArrayList.

> Back to the list of macro method categories

Text manipulation

Method	Return type	Parameters	Description
Contains	bool	• string text • string search	Returns a true value if the string specified by the second parameter occurs within the first string.
EndsWith	bool	• string text • string findText	Indicates whether the string specified by the second parameter occurs at the end of the first string.
Format	string	• string value • string format	Replaces all formatting expressions in a string using the text equivalents specified in the second parameter. Based on composite formatting.
FormatNotEmpty	string	• string value • string format • *string emptyResult*	If the first parameter is not empty or null, replaces all formatting expressions in the string using the text equivalents specified in the second parameter. Returns the optional third parameter if the value is null or empty. Based on composite formatting.
GetMatch	string	• string text • string regex	Matches the string value to the specified regular expression and returns the match.
IndexOf	int	• string text • string searchFor	Returns the index of the first occurrence of the second string within the first string.
LastIndexOf	int	• string text • string searchFor	Returns the index of the last occurrence of the second string within the first string.
LimitLength	string	• string text • int length • *string padString* • *bool wholeWords*	Limits the length of the string to the specified number of characters. The *padString* parameter allows you to set a string that the method appends to the end of the limited result. The length of the parameter is included in the maximum length. If you set the *wholeWords* para meter to true, the method preserves the last word in the string.
LoremIpsum	string	• *int length*	Generates lorem Ipsum text. You can specify the number of generated characters (1000 by default).
Matches	bool	• string text • string regex	Indicates whether the string matches the specified regular expression.

NotContains	bool	• string text • string search	Returns a true value if the string specified by the second parameter does NOT occur within the first string.
PadLeft	string	• string text • int length • *string paddingString*	Adds leading characters to the string until the total length matches the second parameter. By default, the method uses spaces as the padding character, but you can optionally specify a different character.
PadRight	string	• string text • int length • *string paddingString*	Adds trailing characters to the string until the total length matches the second parameter. By default, the method uses spaces as the padding character, but you can optionally specify a different character.
RegexReplace	string	• string text • string regex • string replacement	Replaces strings that match the regular expression pattern within the first string using the replacement string.
Remove	string	• string text • int position • *int length*	Deletes characters from the string, starting at the specified index. By default, the method removes all remaining characters from the specified position. You can optionally specify the number of characters to be removed.
Replace	string	• string text • string replace • string replacement	Replaces all occurrences of the second string within the first string using the replacement string.
Split	string[]	• string text • string delimiters • *bool removeEmpty*	Separates the string into an array of substrings according to the specified delimiting characters. If the optional third parameter is true, the method removes empty substrings from the result.
StartsWith	bool	• string text • string findText	Indicates whether the string specified by the second parameter occurs at the start of the first string.
Substring	string	• string text • int index • *int length*	Returns a substring starting at the specified index. By default, the method returns all remaining characters from the specified index. You can optionally specify the number of characters to be returned.
ToLower	string	• string text	Converts the entire string to lower case.
ToUpper	string	• string text	Converts the entire string to upper case.
Trim	string	• string text • *string charsToTrim*	Removes all leading and trailing occurrences of characters from the string. By default, the method trims white space characters, but you can optionally specify a different set of characters (as a string).

TrimEnd	string	• string text • *string charsToTrim*	Removes all trailing occurrences of characters from the string. By default, the method trims white space characters, but you can optionally specify a different set of characters (as a string).
TrimStart	string	• string text • *string charsToTrim*	Removes all leading occurrences of characters from the string. By default, the method trims white space characters, but you can optionally specify a different set of characters (as a string).

Tips: To perform lexicographical comparison of strings, use the basic comparison operators (==, <, >, <=, >=). Add macro parameters to specify case sensitivity and the culture context of text operations.

> Back to the list of macro method categories

Advanced text processing

Method	Return type	Parameters	Description
GetResourceString	string	• string resourceStringKey • *string culture*	Translates the specified resource (localization) string. By default, the target language of the localization depends on the current culture. You can specify the target language using the optional parameter.
JSEscape	string	• string text	Escapes the string for safe usage in JavaScript (to avoid XSS vulnerabilities).
Localize	string	• string inputText • *string culture*	Resolves localization expressions within the specified text. By default, the target language of the localization depends on the current culture. You can specify the target language using the optional parameter.
MapPath	string	• string path	Returns the physical file path that matches the specified virtual path.
ResolveBBCode	string	• string text	Resolves BBCode tags in the specified text, for example: *{% ResolveBBCode("[quote]Sample text[/quote]") %}*.
ResolveMacroExpressions	string	• string expression	Resolves the specified macro expression (without the *{% %}* parentheses).
ResolveMacros	string	• string inputText	Resolves all macros within the specified text.
ResolveUrl	string	• string url	Resolves relative URLs into absolute.
StripTags	string	• string text	Removes all HTML tags from the specified text.
SQLEscape	string	• string text	Escapes the string for safe usage in SQL commans (to avoid SQL injection).

UnresolveUrl	string	• string url	Converts absolute URLs to relative URLs (starting with ~).
UrlEncode	string	• string url	Applies URL encoding to the specified string.

> Back to the list of macro method categories

Date and time modification

Method	Return type	Parameters	Description
AddMilliseconds	DateTime	• DateTime datetime • int milliseconds	Adds the specified number of milliseconds to a DateTime value.
AddSeconds	DateTime	• DateTime datetime • int seconds	Adds the specified number of seconds to a DateTime value.
AddMinutes	DateTime	• DateTime datetime • int minutes	Adds the specified number of minutes to a DateTime value.
AddHours	DateTime	• DateTime datetime • int hours	Adds the specified number of hours to a DateTime value.
AddDays	DateTime	• DateTime datetime • int days	Adds the specified number of days to a DateTime value.
AddWeeks	DateTime	• DateTime datetime • int weeks	Adds the specified number of weeks to a DateTime value.
AddMonths	DateTime	• DateTime datetime • int months	Adds the specified number of months to a DateTime value.
AddYears	DateTime	• DateTime datetime • int years	Adds the specified number of years to a DateTime value.
ToShortDateString	string	• DateTime datetime	Converts the value of the DateTime parameter to an equivalent short date string representation.
ToShortTimeString	string	• DateTime datetime	Converts the value of the DateTime parameter to an equivalent short time string representation.

> Back to the list of macro method categories

Mathematical operations

The macro autocomplete help only shows the mathematical methods as members of the **Math** namespace, for example *{% Math.Abs(-2) %}*. However, the system resolves the methods even without the namespace.

Method	Return type	Parameters	Description
Abs	double	• double number	The absolute value of the specified number.
Acos	double	• double number	The angle whose cosine is the specified number.
Asin	double	• double number	The angle whose sine is the specified number.
Atan	double	• double number	The angle whose tangent is the specified number.
Average	double	• InfoObjectCollection collection • string columnName	The average of all numbers in the specified data column of the objects in the collection.

Ceiling	double	• double number	The smallest whole number greater than or equal to the specified number.
Cos	double	• double number	The cosine of the specified angle.
Cosh	double	• double number	The hyperbolic cosine of the specified angle.
Exp	double	• double number	**e** raised to the specified power.
Floor	double	• double number	The largest whole number lesser than or equal to the specified number.
GetRandomInt	int	• *int minValue* • *int maxValue* • *int seed*	Generates a random positive integer. You can use the optional parameters to specify the range of possible numbers, and the seed number for the generator.
GetRandomDouble	double	• *int minValue* • *int maxValue* • *int seed*	Generates a random positive decimal number. You can use the optional parameters to specify the range of possible numbers, and the seed number for the generator.
IsOdd	bool	• int number	Returns a true value if the specified number is odd.
IsEven	bool	• int number	Returns a true value if the specified number is even.
Log	double	• double number	The base **e** logarithm of a specified number.
Log10	double	• double number	The base 10 logarithm of a specified number.
Max	double	• double parameters	The maximum from the given list of numbers.
Maximum	double	• InfoObjectCollection collection • string columnName	The maximum of all numbers in the specified data column of the objects in the collection.
Min	double	• double parameters	The minimum from the given list of numbers.
Minimum		• InfoObjectCollection collection • string columnName	The minimum of all numbers in the specified data column of the objects in the collection.
Modulo	int	• int left • int right	The modulo of two integer numbers.
Pow	double	• double base • double exp	The number raised to the specified power.
Percent	double	• double percent	Multiples the specified number by 0.01.

Round	double	• double number • *int digits* • *string mode*	The number nearest to the specified value. The optional second parameter sets the number of fractional digits in the rounded value. You can use the third parameter to set the rounding mode for numbers half-way between two other numbers. Supported variants are: • AwayFromZero - rounds to the nearest number away from zero • ToEven - rounds toward the nearest even number
Sign	double	• double number	Returns a value indicating the sign of a number: • -1 (the number is less than zero) • 0 (the number is zero) • 1 (the number is greater than zero)
Sin	double	• double number	The sine of the specified angle.
Sinh	double	• double number	The hyperbolic sine of the specified angle.
Sqrt	double	• double number	The square root of a specified number.
Sum	double	• InfoObjectCollection collection • string columnName	The sum of all numbers in the specified data column of the objects in the collection.
Tan	double	• double number	The tangent of the specified angle.
Tanh	double	• double number	The hyperbolic tangent of the specified angle.

> Back to the list of macro method categories

User membership and permissions

Method	Return type	Parameters	Description
CheckPrivilegeLevel	bool	• object user (UserInfo) • UserPrivilegeLevelEnum privilegeLevel	Checks the user's privilege level. Returns a true value if the user's privilege level matches the required *UserPrivilegeLevelEnum* value or is higher. You can check the following levels: • *{% CurrentUser.CheckPrivilegeLevel(UserPrivilegeLevelEnum.Editor) %}* • *{% CurrentUser.CheckPrivilegeLevel(UserPrivilegeLevelEnum.Admin) %}* • *{% CurrentUser.CheckPrivilegeLevel(UserPrivilegeLevelEnum.GlobalAdmin) %}*

GetFormattedUserName	string	• object user (UserInfo) • *bool isLiveSite*	Returns the formatted username of the specified user object. The default format is *<full name> (<nickname>)* if the user has a nickname, otherwise *<full name> (<username>)*. The **isLiveSite** parameter determines whether the macro displays the username on the live website.
HasAnyMembership	bool	• object user (UserInfo)	Checks whether the user belongs to any membership in the system.
HasMembership	bool	• object user (UserInfo) • string userMemberships • *bool allMemberships*	Checks if a user belongs to the specified memberships. Enter the code names of memberships through the **user Memberships** parameter. When checking multiple memberships, use semicolons (;) to separate the code names. If the **allMemberships** parameter is true, the user must belong to all specified memberships for the check to succeed. To check whether the user belongs to a membership on a global level, add the period character (.) prefix before the membership code name.
IsAuthorizedPerResource	bool	• object user (UserInfo) • string resource • string permission	Evaluates whether a user has a specific permission for a resource (module). Use code names to identify the resource and permission.
IsAuthorizedPerUIElement	bool	• object user (UserInfo) • string resource • string elementName	Evaluates whether a user is allowed to access a specific UI element of a resource (module). Use code names to identify the resource and element.
IsInGroup	bool	• object user (UserInfo) • string userGroup	Checks if a user belongs to a group (community or workgroup).
IsInRole	bool	• object user (UserInfo) • string userRole • *bool allRoles*	Checks if a user belongs to the specified roles. Enter the code names of roles through the **userRole** parameter. When checking multiple roles, use semicolons (;) to separate the code names. If the **allRoles** parameter is true, the user must belong to all specified roles for the check to succeed. To check whether the user belongs to a role on a global level, add the period character (.) prefix before the role code name.

> Back to the list of macro method categories

Transformations

Method	Return type	Parameters	Description

ApplyTransformation	string	• IEnumerable collection • string transformationName • *string contentBeforeTransformationName* • *string contentAfterTransformationName*	Applies **Text / XML** transformations to a collection of items. See also: Using transformations in macro expressions
Transform	string	• IEnumerable collection • string transformationText	Applies ad-hoc transformation code (Text / XML type) to a collection of items.

When writing **Text / XML** transformations, you can use all ASCX transformation methods inside macro expressions.

Outside of transformations, the macro autocomplete only offers the methods under the **Transformation** namespace, for example *Transformation.GetDocumentUrl()*. However, you can manually enter the methods even without the namespace.

> Back to the list of macro method categories