**HTL – Expression Language**

The HTML Template Language uses an expression language to access the data structures that provide the dynamic elements of the HTML output. These expressions are delimited by characters **${** and **}**. To avoid malformed HTML, expressions can only be used in attribute values, in element content, or in comments.

|  |  |
| --- | --- |
|  | **<!-- ${component.path} -->**  **<h1 class="${component.name}">**  **${properties.jcr:title}**  **</h1>** |

Expressions can be escaped by prepended by a **\** character, for instance **\${test}** will render **${test}**.

**Variables**

Variables are containers that store data values or objects. The names of variables are called identifiers.

Without having to specify anything, HTL provides access to all objects that were commonly available in JSP after including **global.jsp**. The [Global Objects](https://helpx.adobe.com/experience-manager/htl/using/global-objects.html) page provides the list of all objects provided access to by HTL.

**Property Access**

There are two ways to access properties of variables, with a dot notation, or with a bracket notation:

**${currentPage.title}  
${currentPage['title']} or ${currentPage["title"]}**

The simpler dot notation should be preferred for most cases, and the brackets notation should be used to access properties that contain invalid identifier characters, or to access properties dynamically. The following two chapters will provide details about these two cases.

The accessed properties can be functions, however passing arguments is not supported, so only functions that don't expect arguments can accessed, like getters. This is a desired limitation, which is intended to reduce the amount of logic embedded into expressions. If needed, the [**data-sly-use**](https://helpx.adobe.com/experience-manager/htl/using/block-statements.html#use) statement can be used to pass parameters to the logic.

Also shown in the example above is that Java getter functions, like **getTitle()**, can be accessed without prepending the **get**, and by lowering the case of the character that follows.

**Valid Indentifier Characters**

The names of variables, called identifiers, conform to certain rules. They must start with a letter (**A**-**Z** and **a**-**z**), or an underscore (**\_**), and subsequent characters can also be digits (**0**-**9**) or colon (**:**). Unicode letters such as **å** and **ü** cannot be used in identifiers.

Given that the colon (**:**) character is common in AEM property names, it is convenient that it is a valid identifier character:

**${properties.jcr:title}**

The bracket notation can be used to access properties that contain invalid identifier characters, like the space character in the example below:

**${properties['my property']}**

**Accessing Members Dynamically**

|  |  |
| --- | --- |
|  | **${properties[myVar]}** |

**Permissive Handling of Null Values**

|  |  |
| --- | --- |
|  | **${currentPage.lastModified.time.toString}** |

**Literals**

A literal is a notation for representing a fixed value.

**Boolean**

Boolean represents a logical entity and can have two values: **true**, and **false**.

**${true} ${false}**

**Numbers**

There is only one number type: positive integers. While other number formats, like floating point, are supported in variables, but cannot be expressed as literals.

**${42}**

**Strings**

They represent textual data, and can be single or double quoted:

**${'foo'} ${"bar"}**

In addition to ordinary characters, following special characters can be used:

* **\\** Backslash character
* **\'** Single quote (or apostrophe)
* **\"** Double quote
* **\t** Tabulation
* **\n** New line
* **\r** Carriage return
* **\f** Form feed
* **\b** Backspace
* **\uXXXX** The Unicode character specified by the four hexadecimal digits XXXX.  
  Some useful unicode escape sequences are:
  + **\u0022** for **"**
  + **\u0027** for **'**

For characters not listed above, preceding a backslash caracter will display an error.

Here are some examples of how to use string escaping:

|  |  |
| --- | --- |
|  | **<p>${'it\'s great, she said "yes!"'}</p>**  **<p title="${'it\'s great, she said \u0022yes!\u0022'}">...</p>** |

which will result in following output, because HTL will apply context-specific escaping:

|  |  |
| --- | --- |
|  | **<p>it&#39;s great, she said &#34;yes!&#34;</p>**  **<p title="it&#39;s great, she said &#34;yes!&#34;">...</p>** |

**Arrays**

An array is an ordered set of values that can be referred to with a name and an index. The types of its elements can be mixed.

|  |  |
| --- | --- |
|  | **${[1,2,3,4]}**  **${myArray[2]}** |

Arrays are useful to provide a list of values from the template.

|  |  |
| --- | --- |
|  | **<ul data-sly-list="${[1,2,3,4]}">**  **<li>${item}</li>**  **</ul>** |

**Operators**

**Logical Operators**

These operators are typically used with Boolean values, however, like in JavaScript, they actually return the value of one of the specified operands, so when used with non-Boolean values, they may return a non-Boolean value.

If a value can be converted to **true**, the value is so-called truthy. If a value can be converted to **false**, the value is so-called falsy. Values that can be converted to **false** are: undefined variables, null values, the number zero, and empty strings.

**Logical NOT**

**${!myVar}** returns **false** if its single operand can be converted to **true**; otherwise, returns **true**.

This can for instance be used to invert a test condition, like displaying an element only if there are no child pages:

|  |  |
| --- | --- |
|  | **<p data-sly-test="${!currentPage.hasChild}">current page has no children</p>** |

**Logical AND**

**${varOne && varTwo}** returns **varOne** if it is falsy; otherwise, returns **varTwo**.

This operator can be used to test two conditions at once, like verifying the existence of two properties:

|  |  |
| --- | --- |
|  | **<div data-sly-test="${properties.jcr:title && properties.jcr:description}">**  **<h1>${properties.jcr:title}</h1>**  **<p>${properties.jcr:description}</p>**  **</div>** |

The logical AND operator can also be used to conditionally display HTML attributes, because HTL removes attributes with values set dynamically that evaluate to false, or to an empty string. So in the example below, the **class** attribute is only shown if **logic.showClass** is truthy and if **logic.className** exists and is not empty:

|  |  |
| --- | --- |
| **1** | **<div class="${logic.showClass && logic.className}">...</div>** |

**Logical OR**

**${varOne || varTwo}** returns **varOne** if it is truthy; otherwise, returns **varTwo**.

This operator can be used to test if one of two conditions apply, like verifying the existence of at least one property:

|  |  |
| --- | --- |
|  | **<div data-sly-test="${properties.jcr:title || properties.jcr:description}">...</div>** |

As the logical OR operator returns the first variable that is truthy, it can also very conveniently be used to provide fallback values.

conditionally display HTML attributes, because HTL removes attributes with values set by expressions that evaluate to false or to an empty string. So the example below will display **properties.jcr:**title if it exists and is not empty, else it falls back to dislaying **properties.jcr:description** if it exists and is not empty, else it will display the message "no title or description provided":

|  |  |
| --- | --- |
|  | **<p>${properties.jcr:title || properties.jcr:description || "no title or description provided"}</p>** |

**Conditional (ternary) Operator**

**${varCondition ? varOne : varTwo}** returns **varOne** if **varCondition** is truthy; otherwise it returns **varTwo**.

This operator can typically be used to define conditions within expressions, like displaying a different message based on the status of the page:

|  |  |
| --- | --- |
|  | **<p>${currentPage.isLocked ? "page is locked" : "page can be edited"}</p>** |

An important note, since colon characters are also permitted in identifiers, it is best to separate the ternary operators with a white space to provide clarity to the parser:

|  |  |
| --- | --- |
|  | **<p>${properties.showDescription ? properties.jcr:description : properties.jcr:title}</p>** |

**Comparison Operators**

The equality and inequality operators only support operands that are of identical types. When the types don't match, an error is displayed.

* Strings are equal when they have the same sequence of characters.
* Numbers are equal when they have the same value
* Booleans are equal if both are **true** or both are **false**.
* Null or undefined variables are equal to themselves and to each other.

**${varOne == varTwo}** returns **true** if **varOne** and **varTwo** are equal.

**${varOne != varTwo}** returns **true** if **varOne** and **varTwo** are not equal.

The relational operators only support operands that are numbers. For all other types, an error is displayed.

**${varOne > varTwo}** returns **true** if **varOne** is greater than **varTwo**.

**${varOne < varTwo}** returns **true** if **varOne** is smaller than **varTwo**.

**${varOne >= varTwo}** returns **true** if **varOne** is greater or equal to **varTwo**.

**${varOne <= varTwo}** returns **true** if **varOne** is smaller or equal to **varTwo**.

**Grouping parentheses**

The grouping operator **(** **)** controls the precedence of evaluation in expressions.

**${varOne && (varTwo || varThree)}**

**Options**

Expression options can act on the expression and modify it, or serve as parameters when used in conjunction with block statements.

Everything after the **@** is an option:

|  |  |
| --- | --- |
|  | **${myVar @ optOne}** |

Options can have a value, which may be a variable or a literal:

|  |  |
| --- | --- |
|  | **${myVar @ optOne=someVar}**  **${myVar @ optOne='bar'}**  **${myVar @ optOne=10}**  **${myVar @ optOne=true}** |

Multiple options are separated by commas:

|  |  |
| --- | --- |
|  | **${myVar @ optOne, optTwo=bar}** |

Parametric expressions containing only options are also possible:

|  |  |
| --- | --- |
|  | **${@ optOne, optTwo=bar}** |

**String Formatting**

Option that replaces the enumerated placeholders, {*n*}, with the corresponding variable:

|  |  |
| --- | --- |
|  | **${'Page {0} of {1}' @ format=[current, total]}** |

**Internationalization**

Translates the string to the language of the current *source* (see below), using the current [dictionary](https://helpx.adobe.com/experience-manager/6-3/sites/developing/using/i18n-translator.html). If no translation is found, the original string is used.

|  |  |
| --- | --- |
|  | **${'Page' @ i18n}** |

The hint option can be used to provide a comment for translators, specifying the context in which the text is used:

|  |  |
| --- | --- |
|  | **${'Page' @ i18n, hint='Translation Hint'}** |

The default source for the language is 'resource', meaning that the text gets translated to the same language as the content. This can be changed to 'user', meaning that the language is taken from the browser locale or from the locale of the logged-in user:

|  |  |
| --- | --- |
|  | **${'Page' @ i18n, source='user'}** |

Providing an explicit locale overrides the source settings:

|  |  |
| --- | --- |
|  | **${'Page' @ i18n, locale='en-US'}** |

To embed variables into a translated string, the format option can be used:

|  |  |
| --- | --- |
|  | **${'Page {0} of {1}' @ i18n, format=[current, total]}** |

**Array Join**

By default, when displaying an array as text, HTL will display comma separated values (without spacing).

Use the join option to specify a different separator:

|  |  |
| --- | --- |
|  | **${['one', 'two'] @ join='; '}** |

**Display Context**

The display context of a HTL expression refers to its location within the structure of the HTML page. For example, if the expression appears in place that would produce a text node once rendered, then it is said to be in a **text** context. If it is found within the value of an attribute, then it is said to be in an **attribute** context, and so forth.

With the exception of script (JS) and style (CSS) contexts, HTL will automatically detect the context of expressions and escape them appropriately, to prevent XSS security problems. In the case of scripts and CSS, the desired context behavior must be explicitly set. Additionally, the context behavior can also be explicitly set in any other case where an override of the automatic behavior is desired.

Here we have three variables in three different contexts: **properties.link** (**uri** context), **properties.title**(**attribute** context) and **properties.text**(**text** context). HTL will escape each of these differently in accordance with the security requirements of their respective contexts. No explicit context setting is required in normal cases such as this one:

|  |  |
| --- | --- |
|  | **<a href="${properties.link}" title="${properties.title}">${properties.text}</a>** |

To safely output markup (that is, where the expression itself evaluates to HTML), the **html** context is used:

|  |  |
| --- | --- |
|  | **<div>${properties.richText @ context='html'}</div>** |

Explicit context must be set for style contexts:

|  |  |
| --- | --- |
|  | **<span style="color: ${properties.color @ context='styleToken'};">...</span>** |

Explicit context must be set for script contexts:

|  |  |
| --- | --- |
|  | **<span onclick="${properties.function @ context='scriptToken'}();">...</span>** |

Escaping and XSS protection can also be turned off:

|  |  |
| --- | --- |
|  | **<div>${myScript @ context='unsafe'}</div>** |

**Context Settings**

|  |  |  |
| --- | --- | --- |
| **Context** | **When to use** | **What it does** |
| **text** | *Default for content inside elements* | Encodes all HTML special characters. |
| **html** | To safely output markup | Filters HTML to meet the AntiSamy policy rules, removing what doesn't match the rules. |
| **attribute** | *Default for attribute values* | Encodes all HTML special characters. |
| **uri** | To display links and paths *Default for****href****and****src****attribute values* | Validates URI for writing as an **href** or **src** attribute value, outputs nothing if validation fails. |
| **number** | To display numbers | Validates URI for containing an integer, outputs zero if validation fails. |
| **attributeName** | *Default for****data-sly-attribute****when setting attribute names* | Validates the attribute name, outputs nothing if validation fails. |
| **elementName** | *Default for****data-sly-element*** | Validates the element name, outputs nothing if validation fails. |
| **scriptToken** | For JS identifiers, literal numbers, or literal strings | Validates the JavaScript token, outputs nothing if validation fails. |
| **scriptString** | Within JS strings | Encodes characters that would break out of the string. |
| **scriptComment** | Within JS comments | Validates the JavaScript comment, outputs nothing if validation fails. |
| **styleToken** | For CSS identifiers, numbers, dimensions, strings, hex colours or functions. | Validates the CSS token, outputs nothing if validation fails. |
| **styleString** | Within CSS strings | Encodes characters that would break out of the string. |
| **styleComment** | Within CSS comments | Validates the CSS comment, outputs nothing if validation fails. |
| **unsafe** | Only if none of the above does the job | Disables escaping and XSS protection completely. |