**HTL BLOCK STATEMENTS**

HTML Template Language (HTL) block statements are custom **data** attributes added directly to existing HTML. This allows easy and unobtrusive annotation of a prototype static HTML page, converting it to a functioning dynamic template without breaking the validity of the HTML code.

**sly element**

The **<sly> element** does not get displayed in the resulting HTML and can be used instead of the data-sly-unwrap. The goal of the <sly> element is to make it more obvious that the element is not outputted. If you want you can still use data-sly-unwrap.

|  |  |
| --- | --- |
|  | **<sly data-sly-test.varone="${properties.yourProp}" />** |

As with data-sly-unwrap, try to minimize the use of this.

**use**

**data-sly-use**: Initializes a helper object (defined in JavaScript or Java) and exposes it through a variable.

Initialize a JavaScript object, where the source file is located in the same directory as the template. Note that the filename must be used:

|  |  |
| --- | --- |
|  | **<div data-sly-use.nav="navigation.js">${nav.foo}</div>** |

Initialize a Java class, where the source file is located in the same directory as the template. Note that the classname must be used, not the file name:

|  |  |
| --- | --- |
|  | **<div data-sly-use.nav="Navigation">${nav.foo}</div>** |

Initialize a Java class, where that class is installed as part of an OSGi bundle. Note that its fully-qualified class name must be used:

|  |  |
| --- | --- |
|  | **<div data-sly-use.nav="org.example.Navigation">${nav.foo}</div>** |

Parameters can be passed to the initialization using *options*. Generally this feature should only be used by HTL code that is itself within a **data-sly-template** block:

|  |  |
| --- | --- |
|  | **<div data-sly-use.nav="${'navigation.js' @parentPage=currentPage}">${nav.foo}</div>** |

Initialize another HTL template that can then be called using **data-sly-call**:

|  |  |
| --- | --- |
|  | **<div data-sly-use.nav="navTemplate.html" data-sly-call="${nav.foo}"></div>** |

The following table gives an overview of the pros and cons of each API.

|  |  |  |
| --- | --- | --- |
|  | **Java Use-API** | **JavaScript Use-API** |
| **Pros** | * faster * can be inspected with a debugger * easy to unit-test | * can be modified by front-end developers * is located within the component, keeping the view logic of a component close to it's corresponding template |
| **Cons** | * cannot be modified by front-end developers | * slower * no debugger (yet) * harder to unit-test |

For page components, it is recommended to use a mixed model, where all model logic is located in Java, providing clear APIs that are agnostic to anything that happens in the view (i.e. within the components). AEM comes with great default models like the Page or the Resource API that should be able to cover most cases.

All view logic that is specific to a component should be placed within that component as JavaScript, because it belongs to that component.

**unwrap**

**data-sly-unwrap**: Removes the host element from the generated markup while retaining its content. This allows the exclusion of elements that are required as part of HTL presentation logic but are not desired in the actual output.

However, this statement should be used sparingly. In general it is better to keep the HTL markup as close as possible to the intended output markup. In other words, when adding HTL block statements, try as much as possible to simply annotate the existing HTML, without introducing new elements.

For example, this

|  |  |
| --- | --- |
|  | **<p data-sly-use.nav="navigation.js">Hello World</p>** |

will produce

|  |  |
| --- | --- |
|  | **<p>Hello World</p>** |

Whereas this,

|  |  |
| --- | --- |
|  | **<p data-sly-use.nav="navigation.js" data-sly-unwrap>Hello World</p>** |

will produce

|  |  |
| --- | --- |
|  | **Hello World** |

It is also possible to conditionally unwrap an element:

|  |  |
| --- | --- |
|  | **<div class="popup" data-sly-unwrap="${isPopup}">content</div>** |

**text**

**data-sly-text**: Replaces the content of its host element with the specified text.

For example, this

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

is equivalent to

|  |  |
| --- | --- |
|  | **<p data-sly-text="${properties.jcr:description}">Lorem ipsum</p>** |

Both will display the value of **jcr:description** as paragraph text. The advantage of the second method is that is allows the unobtrusive annotation of HTML while keeping the static placeholder content from the original designer.

**attribute**

**data-sly-attribute**: Adds attributes to the host element.

For example, this

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

is equivalent to

|  |  |
| --- | --- |
|  | **<div title="Lorem Ipsum" data-sly-attribute.title="${properties.jcr:title}"></div>** |

Both will set the **title** attribute to the value of **jcr:title**. The advantage of the second method is that is allows the unobtrusive annotation of HTML while keeping the static placeholder content from the original designer.

Attributes are resolved left to right, with the rightmost instance of an attribute (either literal or defined via **data-sly-attribute**) taking precedence over any instances of the same attribute (defined either literally or via **data-sly-attribute**) defined to its left.

Note that an attribute (either **literal** or set via **data-sly-attribute**) whose value *evaluates* to the empty string will be removed in the final markup. The one exception to this rule is that a *literal* attribute set to a *literal* empty string will be preserved. For example,

|  |  |
| --- | --- |
|  | **<div class="${''}" data-sly-attribute.id="${''}"></div>** |

produces,

|  |  |
| --- | --- |
|  | **<div></div>** |

but,

|  |  |
| --- | --- |
|  | **<div class="" data-sly-attribute.id=""></div>** |

produces,

|  |  |
| --- | --- |
|  | **<div class=""></div>** |

To set multiple attributes, pass a map object hold key-value pairs corresponding to the attributes and their values. For example, assuming,

|  |  |
| --- | --- |
|  | **attrMap = {**  **title: "myTitle",**  **class: "myClass",**  **id: "myId"**  **}** |

Then,

|  |  |
| --- | --- |
|  | **<div data-sly-attribute="${attrMap}"></div>** |

produces,

|  |  |
| --- | --- |
|  | **<div title="myTitle" class="myClass" id="myId"></div>** |

**element**

**data-sly-element**: Replaces the element name of the host element.

For example,

|  |  |
| --- | --- |
|  | **<h1 data-sly-element="${titleLevel}">text</h1>** |

Replaces the **h1** with the value of **titleLevel**.

For security reasons, **data-sly-element** accepts only the following element names:

|  |  |
| --- | --- |
|  | **a abbr address article aside b bdi bdo blockquote br caption cite code col colgroup**  **data dd del dfn div dl dt em figcaption figure footer h1 h2 h3 h4 h5 h6 header i ins**  **kbd li main mark nav ol p pre q rp rt ruby s samp section small span strong sub**  **sup table tbody td tfoot th thead time tr u var wbr** |

To set other elements, XSS security must be turned off (**@context='unsafe'**).

**test**

**data-sly-test:** Conditionally removes the host element and it's content. A value of **false** removes the element; a value of **true** retains the element.

For example, the **p** element and its content will only be rendered if **isShown** is **true**:

|  |  |
| --- | --- |
|  | **<p data-sly-test="${isShown}">text</p>** |

The result of a test can be assigned to a variable that can be used later. This is usually used to construct "if else" logic, since there is no explicit else statement:

|  |  |
| --- | --- |
|  | **<p data-sly-test.abc="${a || b || c}">is true</p>**  **<p data-sly-test="${!abc}">or not</p>** |

The variable, once set, has global scope within the HTL file.

Following are some examples on comparing values:

|  |  |
| --- | --- |
|  | **<div data-sly-test="${properties.jcr:title == 'test'}">TEST</div>**  **<div data-sly-test="${properties.jcr:title != 'test'}">NOT TEST</div>**    **<div data-sly-test="${properties['jcr:title'].length > 3}">Title is longer than 3</div>**  **<div data-sly-test="${properties['jcr:title'].length >= 0}">Title is longer or equal to zero </div>**    **<div data-sly-test="${properties['jcr:title'].length > aemComponent.MAX\_LENGTH}">**  **Title is longer than the limit of ${aemComponent.MAX\_LENGTH}**  **</div>** |

**repeat**

With data-sly-repeat you can *repeat* an element multiple times based on the list that is specified.

|  |  |
| --- | --- |
|  | **<div data-sly-repeat="${currentPage.listChildren}">${item.name}</div>** |

This works the same way as data-sly-list, except that you do not need a container element.

The following example shows that you can also refer to the *item* for attributes:

|  |  |
| --- | --- |
|  | **<div data-sly-repeat="${currentPage.listChildren}" data-sly-attribute.class="${item.name}">${item.name}</div>** |

**list**

**data-sly-list**: Repeats the content of the host element for each enumerable property in the provided object.

Here is a simple loop:

|  |  |
| --- | --- |
|  | **<dl data-sly-list="${currentPage.listChildren}">**  **<dt>index: ${itemList.index}</dt>**  **<dd>value: ${item.title}</dd>**  **</dl>** |

The following default variables are available within the scope of the list:

**item**: The current item in the iteration.

**itemList**: Object holding the following properties:

**index**: zero-based counter (**0..length-1**).

**count**: one-based counter (**1..length**).

**first**: **true** if the current item is the first item.

**middle**: **true** if the current item is neither the first nor the last item.

**last**: **true** if the current item is the last item.

**odd**: **true** if **index** is odd.

**even**: **true** if **index** is even.

Defining an identifier on the **data-sly-list** statement allows you to rename the **itemList** and **item** variables. **item** will become ***<variable>*** and **itemList** will become ***<variable>*List**.

|  |  |
| --- | --- |
|  | **<dl data-sly-list.child="${currentPage.listChildren}">**  **<dt>index: ${childList.index}</dt>**  **<dd>value: ${child.title}</dd>**  **</dl>** |

You can also access properties dynamically:

|  |  |
| --- | --- |
|  | **<dl data-sly-list.child="${myObj}">**  **<dt>key: ${child}</dt>**  **<dd>value: ${myObj[child]}</dd>**  **</dl>** |

**resource**

**data-sly-resource**: Includes the result of rendering the indicated resource through the sling **resolution and rendering process.**

**Better Example:**

**<div data-sly-resource="${'par1' @ *resourceType='wcm/foundation/components/parsys'}"*></div>**

A simple resource include:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="path/to/resource"></article>** |

Options allow a number of additional variants:

Manipulating the path of the resource:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${ @ path='path/to/resource'}"></article>**  **<article data-sly-resource="${'resource' @ prependPath='my/path'}"></article>**  **<article data-sly-resource="${'my/path' @ appendPath='resource'}"></article>** |

Add (or replace) a selector:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ selectors='selector'}"></article>** |

Add, replace or remove multiple selectors:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ selectors=['s1', 's2']}"></article>** |

Add a selector to the existing ones:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ addSelectors='selector'}"></article>** |

Remove some selectors from the existing ones:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ removeSelectors='selector1'}"></article>** |

Remove all selectors:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ removeSelectors}"></article>** |

Overrides the resource type of the resource:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ resourceType='my/resource/type'}"></article>** |

Changes the WCM mode:

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ wcmmode='disabled'}"></article>** |

By default, the AEM decoration tags are disabled, the decorationTagName option allows to bring them back, and the cssClassName to add classes to that element.

|  |  |
| --- | --- |
|  | **<article data-sly-resource="${'path/to/resource' @ decorationTagName='span',**  **cssClassName='className'}"></article>** |

**include**

**data-sly-include**: Replaces the content of the host element with the markup generated by the indicated HTML template file (HTL, JSP, ESP etc.) when it is processed by its corresponding template engine. The rendering context of the *included file* will not include the current HTL context (that of the *including file*); Consequently, for inclusion of HTL files, the current **data-sly-use** would have to be repeated in the included file (In such a case it is usually better to use **data-sly-template** and **data-sly-call**)

A simple include:

|  |  |
| --- | --- |
|  | **<section data-sly-include="path/to/template.html"></section>** |

JSPs can be included the same way:

|  |  |
| --- | --- |
|  | **<section data-sly-include="path/to/template.jsp"></section>** |

Options let you manipulate the path of the file:

|  |  |
| --- | --- |
|  | **<section data-sly-include="${ @ path='path/to/template.html'}"></section>**  **<section data-sly-include="${'template.html' @ prependPath='my/path'}"></section>**  **<section data-sly-include="${'my/path' @ appendPath='template.html'}"></section>** |

You can also change the WCM mode:

|  |  |
| --- | --- |
|  | **<section data-sly-include="${'template.html' @ wcmmode='disabled'}"></section>** |

**template & call**

**data-sly-template**: Defines a template. The host element and its content are not output by HTL

**data-sly-call**: Calls a template defined with data-sly-template. The content of the called template (optionally parameterized) replaces the content of the host element of the call.

Define a static template and then call it:

|  |  |
| --- | --- |
|  | **<template data-sly-template.one>blah</template>**  **<div data-sly-call="${one}"></div>** |

Define a dynamic template and then call it with parameters:

|  |  |
| --- | --- |
|  | **<template data-sly-template.two="${ @ title}"><h1>${title}</h1></template>**  **<div data-sly-call="${two @ title=properties.jcr:title}"></div>** |

Templates located in a different file, can be initialised with **data-sly-use**. Note that in this case **data-sly-use** and **data-sly-call** could also be placed on the same element:

|  |  |
| --- | --- |
|  | **<div data-sly-use.lib="templateLib.html">**  **<div data-sly-call="${lib.one}"></div>**  **<div data-sly-call="${lib.two @ title=properties.jcr:title}"></div>**  **</div>** |

Template recursion is supported:

|  |  |
| --- | --- |
|  | **<template data-sly-template.nav="${ @ page}">**  **<ul data-sly-list="${page.listChildren}">**  **<li>**  **<div class="title">${item.title}</div>**  **<div data-sly-call="${nav @ page=item}" data-sly-unwrap></div>**  **</li>**  **</ul>**  **</template>**  **<div data-sly-call="${nav @ page=currentPage}" data-sly-unwrap></div>** |

**i18n and locale objects**

When you are using i18n and HTL, you can now also pass in custom locale objects.

|  |  |
| --- | --- |
|  | **${'Hello World' @ i18n, locale=request.locale}** |

**URL manipulation**

A new set of url manipulations is available.

See the following examples on their usage:

Adds the html extension to a path.

|  |  |
| --- | --- |
|  | **<a href="${item.path @ extension = 'html'}">${item.name}</a>** |

Adds the html extension and a selector to a path.

|  |  |
| --- | --- |
|  | **<a href="${item.path @ extension = 'html', selectors='products'}">${item.name}</a>** |

Adds the html extension and a fragment (#value) to a path.

|  |  |
| --- | --- |
|  | **<a href="${item.path @ extension = 'html', fragment=item.name}">${item.name}</a>** |

**HTL Features Supported in AEM 6.3**

The following new HTL features are supported in Adobe Experience Manager (AEM) 6.3:

**Number/Date-formatting**

AEM 6.3 supports native formatting of numbers and dates, without writing custom code. This also supports timezone and locale.

The following examples show that the format is specified first, then the value that needs formatting:

|  |  |
| --- | --- |
|  | **<h2>${ 'dd-MMMM-yyyy hh:mm:ss' @**  **format=currentPage.lastModified,**  **timezone='PST',**  **locale='fr'}</h2>**    **<h2>${ '#.00' @ format=300}</h2>** |

**data-sly-use with resources**

This allows to get resources directly in HTL with data-sly-use and does not require to write code to get the resource.

For example:

|  |  |
| --- | --- |
|  | **<div data-sly-use.product=“/etc/commerce/product/12345”>**  **${ product.title }**  **</div>** |

**Request-attributes**

In the *data-sly-include* and *data-sly-resource* you can now pass *requestAttributes* in order to use them in the receiving HTL-script.

This allows you to properly pass-in parameters into scripts or components.

|  |  |
| --- | --- |
|  | **<sly data-sly-use.settings="com.adobe.examples.htl.core.hashmap.Settings"**  **data-sly-include="${ 'productdetails.html' @ requestAttributes=settings.settings}" />** |

Java-code of the Settings class, the Map is used to pass in the requestAttributes:

|  |  |
| --- | --- |
|  | **public class Settings extends WCMUsePojo {**    **// used to pass is requestAttributes to data-sly-resource**  **public Map<String, Object> settings = new HashMap<String, Object>();**    **@Override**  **public void activate() throws Exception {**  **settings.put("layout", "flex");**  **}**  **}** |

For example, via a Sling-Model, you can consume the value of the specified requestAttributes.

In this example, *layout* is injected via the Map from the Use-class:

|  |  |
| --- | --- |
|  | **@Model(adaptables=SlingHttpServletRequest.class)**  **public class ProductSettings {**  **@Inject @Optional @Default(values="empty")**  **public String layout;**    **}** |

**Fix for @extension**

The @extension works in all scenarios in AEM 6.3, before you could have a result like *www.adobe.com.html*and alsochecks whether to add or not add the extension.

|  |  |
| --- | --- |
|  | **${ link @ extension = 'html' }** |

**The following are code samples for HTL:**

* [We.Retail Reference Implementation](https://helpx.adobe.com/experience-manager/6-3/sites/developing/using/we-retail.html)  
  An AEM site, fully implemented with HTL.
* [HTL TodoMVC Example](https://github.com/Adobe-Marketing-Cloud/aem-sightly-sample-todomvc)  
  An AEM + HTL implementation of the famous exercise.

**The following developer tools are available for HTL:**

* [HTL Read Eval Print Loop](https://github.com/Adobe-Marketing-Cloud/aem-htl-repl)  
  A live execution environment for playing with the language, great for learning.
* [AEM Brackets extension](https://helpx.adobe.com/experience-manager/6-3/sites/developing/using/aem-brackets.html)  
  An easy tool for front-end developers to work on AEM components.
* [AEM Eclipse extension](https://helpx.adobe.com/experience-manager/6-3/sites/developing/using/aem-eclipse.html)  
  A tool for back-end developers to create AEM projects and easily edit them.