**HTL – JAVASCRIPT USE API**

The HTML Template Langugae (HTL) JavaScript Use-API enables a HTL file to access helper code written in JavaScript. This allows all complex business logic to be encapsulated in the JavaScript code, while the HTL code deals only with direct markup production.

**A Simple Example**

We define a component, **info**, located at

**/apps/my-example/components/info**

It contains two files:

* **info.js**: a JavaScript file that defines the use-class.
* **info.html**: an HTL file that defines the component **info**. This code will use the functionality of **info.js** through the use-API.

**/apps/my-example/component/info/info.js**

|  |  |
| --- | --- |
|  | **"use strict";**  **use(function () {**  **var info = {};**  **info.title = granite.resource.properties["title"];**  **info.description = granite.resource.properties["description"];**  **return info;**  **});** |

**/apps/my-example/component/info/info.html**

|  |  |
| --- | --- |
|  | **<div data-sly-use.info="Info">**  **<h1>${info.title}</h1>**  **<p>${info.description}</p>**  **</div>** |

We also create a content node that uses the **info** component at

**/content/my-example**, with properties:

* **sling:resourceType = "my-example/component/info"**
* **title = "My Example"**
* **description = "This is some example content."**

Here is the resulting repository structure:

**Repository Structure**

|  |  |
| --- | --- |
|  | **{**  **"apps": {**  **"my-example": {**  **"components": {**  **"info": {**  **"info.html": {**  **...**  **},**  **"info.js": {**  **...**  **}**  **}**  **}**  **}**  **},**  **"content": {**  **"my-example": {**  **"sling:resourceType": "my-example/component/info",**  **"title": "My Example",**  **"description": "This is some example content."**  **}**  **}**  **}** |

Consider following component template:

|  |  |
| --- | --- |
|  | **<section class="component-name" data-sly-use.component="component.js">**  **<h1>${component.title}</h1>**  **<p>${component.description}</p>**  **</section>** |

The corresponding logic can be written using following ***server-side*** JavaScript, located in a **component.js** file right next to the template:

|  |  |
| --- | --- |
|  | **use(function () {**  **var Constants = {**  **DESCRIPTION\_PROP: "jcr:description",**  **DESCRIPTION\_LENGTH: 50**  **};**    **var title = currentPage.getNavigationTitle() || currentPage.getTitle() || currentPage.getName();**  **var description = properties.get(Constants.DESCRIPTION\_PROP, "").substr(0, Constants.DESCRIPTION\_LENGTH);**    **return {**  **title: title,**  **description: description**  **};**  **});** |

This tries to take the **title** from different sources and crops the description to 50 characters.

**Dependencies**

Let's imagine that we have a utility class that is already equipped with smart features, like the default logic for the navigation title or nicely cutting a string to a certain length:

|  |  |
| --- | --- |
|  | **use(['../utils/MyUtils.js'], function (utils) {**  **var Constants = {**  **DESCRIPTION\_PROP: "jcr:description",**  **DESCRIPTION\_LENGTH: 50**  **};**    **var title = utils.getNavigationTitle(currentPage);**  **var description = properties.get(Constants.DESCRIPTION\_PROP, "").substr(0, Constants.DESCRIPTION\_LENGTH);**    **return {**  **title: title,**  **description: description**  **};**  **});** |

**Extending**

The dependency pattern can also be used to extend the logic of another component (which typically is the **sling:resourceSuperType** of the current component).

Imagine that the parent component already provides the **title**, and we want to add a **description** as well:

|  |  |
| --- | --- |
|  | **use(['../parent-component/parent-component.js'], function (component) {**  **var Constants = {**  **DESCRIPTION\_PROP: "jcr:description",**  **DESCRIPTION\_LENGTH: 50**  **};**    **component.description = utils.shortenString(properties.get(Constants.DESCRIPTION\_PROP, ""), Constants.DESCRIPTION\_LENGTH);**    **return component;**  **});** |

**Passing Parameters to a Template**

In the case of **data-sly-template** statements that can be independent from components, it can be useful to pass parameters to the associated Use-API.

So in our component let's call a template that is located in a different file:

|  |  |
| --- | --- |
|  | **<section class="component-name" data-sly-use.tmpl="template.html" data-sly-call="${tmpl.templateName @ page=currentPage}"></section>** |

Then this is the template located in **template.html**:

|  |  |
| --- | --- |
|  | **<template data-sly-template.templateName="${@ page}" data-sly-use.tmpl="${'template.js' @ page=page, descriptionLength=50}">**  **<h1>${tmpl.title}</h1>**  **<p>${tmpl.description}</p>**  **</template>** |

The corresponding logic can be written using following ***server-side*** JavaScript, located in a **template.js** file right next to the template file:

|  |  |
| --- | --- |
|  | **use(function () {**  **var Constants = {**  **DESCRIPTION\_PROP: "jcr:description"**  **};**    **var title = this.page.getNavigationTitle() || this.page.getTitle() || this.page.getName();**  **var description = this.page.getProperties().get(Constants.DESCRIPTION\_PROP, "").substr(0, this.descriptionLength);**    **return {**  **title: title,**  **description: description**  **};**  **});** |

The passed parameters are set on the **this** keyword.