# Office AMS: Customization via JavaScript

|  |  |
| --- | --- |
| Summary: | Applies to: |
| This sample shows how one can use JavaScript to update SharePoint sites. This technique is a valuable model to do small UI customizations like removing UI elements, translating content or updating complex UI elements. | * Office 365 Multi Tenant (MT) * Office 365 Dedicated (D) * SharePoint 2013 on-premises |
| Solution: | Core.JavaScriptCustomization, version 1.0 |
| Author: | Bert Jansen, Microsoft |
| //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // THIS CODE IS PROVIDED \*AS IS\* WITHOUT WARRANTY OF  // ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING ANY  // IMPLIED WARRANTIES OF FITNESS FOR A PARTICULAR  // PURPOSE, MERCHANTABILITY, OR NON-INFRINGEMENT.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | |

# Scenario 1: Basic JavaScript customization

This sample shows how one can use JavaScript to perform basic updates to a SharePoint site. The later scenarios built further on the basis created in this sample.

## Loading your scripts

SharePoint team sites by default make use of the Minimal Download Strategy (MDS) technique to improve performance. If we want to load custom JavaScript files we have to take this in account by loading the scripts via the below pattern:

// Register script for MDS if possible

RegisterModuleInit("scenario1.js", RemoteManager\_Inject); //MDS registration

RemoteManager\_Inject(); //non MDS run

if (typeof (Sys) != "undefined" && Boolean(Sys) && Boolean(Sys.Application)) {

Sys.Application.notifyScriptLoaded();

}

if (typeof (NotifyScriptLoadedAndExecuteWaitingJobs) == "function") {

NotifyScriptLoadedAndExecuteWaitingJobs("scenario1.js");

}

When the page that contains your script is loaded either the MDS engine (when MDS is enabled) launches your main function (RemoteManager\_Inject) or your function is launched directly for non MDS invocations. The function that’s called is your entry point to load other scripts and to perform the required customizations. Loading other scripts often is needed: the sample shows how you can load the popular jQuery library. When loading other scripts it’s important that the script parts that depend on the loaded script are only executed after the other script was loaded and this is guaranteed via the below construct:

var jQuery = "https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.2.min.js";

// load jQuery and if complete load the js resource file

loadScript(jQuery, function () {

// your customizations go here

});

## Inserting your JavaScript hook into SharePoint Sites

The above descript JavaScript file needs to be loaded by SharePoint. A typical approach would be customizing the master page, but we’ve opted to show a less intrusive approach based on SharePoint custom actions. In the samples you’ll find the below in the code behind of the scenario web pages:

cc.Web.AddJsLink(Utilities.Scenario1Key, Utilities.BuildScenarioJavaScriptUrl(Utilities.Scenario1Key, this.Request));

cc.Web.DeleteJsLink(Utilities.Scenario1Key);

If we look at the above in more detail (in Office AMS Core) you’ll see that in method public bool AddJsLink(string key, string scriptLinks) a block of JavaScript is constructed that will load out JavaScript customization file. Finally this block of JavaScript is used as ScriptBlock in a custom SharePoint action that’s defined via public bool AddCustomAction(CustomActionEntity customAction).

# Scenario 2: Translating via JavaScript

Scenario 2 introduces the concept of JavaScript based resource files. Today’s there’s no possibility to use custom server resource files in the Cloud Application Model, so JavaScript customization is a possible way to mitigate this. The shown technique is applicable for basic translations and we would not advise to use this concept to translate huge amounts of UI elements because the translation will happen at runtime which means the users might see the translation “happen” if there’s a lot of content to translate. To perform some limited translations in navigation elements, page titles or column titles this is however a good approach.

## Loading the JavaScript based resource files

The same load script approach as described in scenario 1 is also used to load the resource files and wait with using them until they’re loaded. Additionally there some code that makes it possible to dynamically load the correct resource file: the whole system depends on having only the correct resource file loaded as the resource files are just a collection of global variables holding translated strings. Below code shows how to extract the used URL of the customization JavaScript file and the potential URL arguments (e.g. rev=8989) to use them for loading the resource file. This assumes the resources files are stored in the same location as your JavaScript customization file. The language to be used is fetched from the \_spPageContextInfo object.

var scriptUrl = "";

var scriptRevision = "";

// iterate the loaded scripts to find the scenario2 script.

// We use the script URL to dynamically build the url for the resource file

// to be loaded.

$('script').each(function (i, el) {

if (el.src.toLowerCase().indexOf('scenario2.js') > -1) {

scriptUrl = el.src;

scriptRevision = scriptUrl.substring(scriptUrl.indexOf('.js') + 3);

scriptUrl = scriptUrl.substring(0, scriptUrl.indexOf('.js'));

}

})

var resourcesFile = scriptUrl + "." + \_spPageContextInfo.currentUICultureName.toLowerCase() + ".js" + scriptRevision;

// load the JS resource file based on the user's language

loadScript(resourcesFile, function () {

// your customizations go here

});

## Applying the translations

To apply a translation is actually nothing more than replacing the text (and potentially also the title attribute) of the element you want to translate by the appropriate variable coming from the JavaScript resource file. Below sample shows this:

// Note that you can use the jQuery each function to iterate all elements

// that match your jQuery selector.

$("span.ms-navedit-flyoutArrow").each(function () {

if (this.innerText.toLowerCase().indexOf('my quicklaunch entry') > -1) {

//update the label

$(this).find('.menu-item-text').text(quickLauch\_Scenario2);

//update the tooltip

$(this).parent().attr("title", quickLauch\_Scenario2);

}

});

// Change the title of the "Hello SharePoint" page

if (IsOnPage("Hello%20SharePoint.aspx")) {

$("#DeltaPlaceHolderPageTitleInTitleArea").find("A").each(function () {

if ($(this).text().toLowerCase().indexOf("hello sharepoint") > -1) {

//update the label

$(this).text(pageTitle\_HelloSharePoint);

//update the tooltip

$(this).attr("title", pageTitle\_HelloSharePoint);

}

});

}

# Scenario 3: Advanced customizations

This scenario shows some more advanced techniques like how to deal with asynchronously loaded page content or page content which is loaded dynamically inside the page during a user event like an onClick event.

## Dealing with asynchronoulsy loaded page content

The solution to both patterns described above depends on timed execution of your JavaScript customization code. We do this by:

1. using a time that fires our code
2. when our code runs it might be that the content we want to update has not yet (fully) loaded and thus our code might not work…alternatively the content was loaded and our code worked fine
3. When the code worked fine we’re all good, when the code did not work we actually use a timer to have the same code run again within x milliseconds
4. Eventually our code will run and the content which it should update has loaded, thus the code will work fine

Below code snippets show the base elements of this pattern.

### Define the variables

//Variables used to control the asynchronous requests

var asyncReqExecutedTime = 350; //Milliseconds

var columnReqExecuted = false;

var columnReqCount = 1

### Call our code via a timer

setTimeout(ColumnReq, asyncReqExecutedTime);

### The code that makes the change

function ColumnReq() {

// If we managed to execute this code then we're done

if (columnReqExecuted) {

return false;

}

//Place your actual "customization logic" here and don't forget the

//flag the request to be true when you're code did succeed

//when the selector returns data this means that the XSLT based listview

//has finished loading and thus the request can be flagged as done

$(".ms-vh-div").each(function () {

if (this.innerText.toLowerCase().indexOf("column1") > -1) {

$(this).text(Column1\_Title);

} if (this.innerText.toLowerCase().indexOf("column2") > -1) {

$(this).text(Column2\_Title);

}

columnReqExecuted = true;

});

// we apparently did not manage to do the change since the element where

// not yet loaded. Schedule another retry

if (!columnReqExecuted) {

// We've tried too many times...something must be wrong here

if (columnReqCount > 15) {

columnReqExecuted = true;

}

else {

columnReqCount = columnReqCount + 1;

// setup the next attempt

setTimeout(ColumnReq, asyncReqExecutedTime);

}

}

}

## Dealing with dynmically created page content

There are places in SharePoint where page content is dynamically loaded via JavaScript triggered by a user action. A typical example here is the site access request system: the popup showing the roles you can assign to a user only is generated after you’ve opened it by a click. This means that our JavaScript code needs to “inject” itself so that it’s executed after the users click event and this is what below sample shows:

//Place your actual "customization logic" here and don't forget the flag

// the request to be true when you're code did succeed

$("a.ms-ellipsis-a").each(function (i, v) {

if ($(this).attr('onclick')) {

pendingReqExecuted = true;

// bind our click event after the original click event, meaning SharePoint

// will first show the content on click and then our code

// will run and update the content loaded by SharePoint

$(this).bind('click', function (e) {

$(".ms-accRqCllOt-PrmCmbBx").find('option').each(function () {

if ($(this).html().toLowerCase().indexOf("[edit]") > -1 ||

$(this).html().toLowerCase().indexOf("[read]") > -1 ) {

// do nothing as we want to leave this entries in the list

} else {

// all entries, except the "placeholder" do have an # in their

// value and thus need to be deleted

if ($(this).attr("value").indexOf("#") > -1) {

$(this).remove();

}

}

});

});

}

});