# Office AMS: JavaSCRIPT Object Model Provisioning Samples

|  |  |
| --- | --- |
| Summary: | Applies to: |
| This application sample demonstrates how to use the JavaScript Object Model to provision a variety of common assets to a SharePoint site collection. | * Office 365 Multi-Tenant (MT) * Office 365 Dedicated (D) * SharePoint 2013 on-premises |
| Solution: | Provisioning.Jsom, version 1.0 |
| Author: | Jim Blanchard (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.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | |

## Version log

Change log between different releases.

|  |  |
| --- | --- |
| Version | Notes |
| 1.0 | Initial |

# General comments

This solution contains two different projects which do the same thing using two different SharePoint App Model approaches. The goal is to demonstrate a couple of things:

* Deploying sites, site columns, content types, lists and list items using JavaScript alone.
* Controlling the sequencing of the provisioning, catching errors when they happen and having the opportunity to log and/or react to them.

The samples currently use values hardcoded into the page; it pulls them out of the DOM using jQuery. This could be replaced with the appropriate mechanism: user entered dropdowns, pulled from a web service, et cetera. The JavaScript library enables the rest.

Provisioning.Jsom.Ncss – This solution uses a no-code sandboxed solution to deploy a page and a JavaScript file to the Site Pages library of a default site. This simulates a trusted scenario where the provisioning logic is deployed directly to a SharePoint host web.

Provisioning.Jsom.App – This is a straight immersive SharePoint hosted app. In this case, the page and JavaScript is loaded into an app web and utilizes the cross domain scripting library to submit JSOM calls to the host web.

In both cases, the JavaScript demonstrates using jQuery Promises to sequence asynchronous calls and handle successes and failures.

# SCENARIo: JSOM Provisioning

The JavaScript Object Model offers powerful capabilities for interacting with SharePoint and can provide a useful alternative to CSOM when a provider-hosted app isn’t a suitable option. This solution contains two projects that complete the same tasks via slightly different mechanisms. Before discussing the differences, first we’ll look at the similarities.

**Promises via the jQuery Deferred Functionality**

Promises are a feature of JavaScript that allow asynchronous operations to be chained together. A full discussion of Promises is beyond the scope of this document, the jQuery website has thorough documentation. Briefly, the jQuery deferred object permits a function to perform an asynchronous operation and signal to the caller the success or failure of that operation.

function createSite() {

var dfd = $.Deferred();

// Do work

// Call an asynchronous function

ctx.executeQueryAsync(

// Everything worked

function () { dfd.resolve(); },

// The function returned an error

function (sender, args) {

console.log("Site creation failure: " + args.get\_message());

dfd.reject();

});

return dfd.promise();

}

The caller of this function can make decisions based on the status of the Deferred object.

createSite().then(

// The Deferred object was successfully resolved

function () {

console.log("Site provisioned");

// Do something else

},

// The Deferred object was rejected

function () { console.log("Site could not be created."); });

**Provisioning Using JSOM**

The pattern for each provisioning operation is basically the same.

* A client context is created to refer to the site being modified
* Changes are made to the client context via JSOM
* The executeQueryAsync method is called on the client context and success/failure handlers are registered to handle what happens next based on the outcome.
* The failure handlers usually register to receive information about the error that occurred to help with problem diagnosis.

**Cross Domain Provisioning**

The NCSS approach is relatively straightforward. The code is loaded from the host web directly, so there are no cross domain issues to worry about. The JavaScript can just directly call into the host web and provision.

The immersive app approach adds the wrinkle that the app is running in an app web underneath the host web in a separate domain. The browser’s single origin policy prevents JavaScript loaded from the app web to target the host web. In this case the cross domain library is used to transfer the call over so that it actually occurs in the host web.

var dfd = $.Deferred();

var ctx = new SP.ClientContext(appweburl);

var fct = new SP.ProxyWebRequestExecutorFactory(appweburl);

ctx.set\_webRequestExecutorFactory(fct);

var appctx = new SP.AppContextSite(ctx, hostweburl);

var thisWeb = appctx.get\_web();

ctx.load(thisWeb);

var wci = constructWebCreationInformation(sitename, siteurl, sitetemplate)

thisWeb.get\_webs().add(wci);

thisWeb.update();

ctx.executeQueryAsync(

function () {

console.log(thisWeb.get\_title());

dfd.resolve();

},

function (sender, args) {

console.log("Site creation failure: " + args.get\_message());

dfd.reject();

});

return dfd.promise();

**Usage**

Provisioning.Jsom.Ncss – This solution builds to a WSP that can be added to the Solutions Gallery of a site collection. Once the solution is activated and the features that the solution adds to the site is also activated, the provisioning.aspx page is provisioned into the Site Pages library of the site.

Provisioning.Jsom.App – This can be built to a .App file and deployed like any other SharePoint hosted app.

# Dependencies

* sp.js
* sp.runtime.js
* sp.requestexecutor.js