JavaScript API to vCloud Director

# Table of Contents

Table of Contents 1

Introduction 3

Bootstrap 3

Event model 3

Event methods 4

Starting the SDK 4

Events 4

Authentication 5

The User object 5

Events 5

The SDK internal data model 6

Reading the data model 6

Refreshing the data model 7

Events 7

Saving to and retrieving from HTML5 local storage 7

Common methods 8

vApp methods 9

VM methods 10

Template Methods 10

Other stored data 11

Custom vCD API calls 11

Metadata 12

Templates and Catalog 12

Tasks 14

Search 14

Localization 14

# Introduction

SilverLining is a VMware Fling that implements a simplified user experience to the cloud. The simplified workflows and interactions allow end users easily find virtualized applications and add them into their personal workspace.

The Fling includes both a JavaScript SDK (vcloud-js-sdk) and a reference implementation (silverlining) to exercise the SDK. The implementation is created in HTML5, CSS, and JavaScript and can be used as an example of how to create a custom UI that talks to VMware vCloud Director (vCD).

The JavaScript SDK communicates with vCD 5.1 through the vCD API. A limited number of API calls are supported in this release. For a reference to the complete vCD API see <http://www.vmware.com/go/vcloudapi> for more information.

*This document is provided only as a guide and is a work in progress.*

# Bootstrap

The Javascript SDK (vcloud-js-sdk) depends on jquery (http://jquery.com/) and jquery-base64 (<https://github.com/carlo/jquery-base64>).

You’re application HTML should include these dependencies, e.g.

<script src="js/lib/jquery-1.7.1.min.js"></script

<script src="js/lib/jquery.base64.min.js"></script

<script src="js/lib/vcloud-js-sdk.min.js"></script>

You instantiate the SDK using your vCloud Director 5.1 API URL, e.g.

var cloud = new vmware.cloud(“https://10.0.0.1/api/",

vmware.cloudVersion.V5\_1);

This cloud object will be used throughout all the code examples in this document.

# Event model

The SDK contains an event manager that emits events and provides methods to register callbacks on these events.

For example, you might want to register a function as a callback that handles the response to the SDK login method, like this

cloud.register(vmware.events.cloud.LOGIN, onLoginResult);

The event manager logs registrations and events in an array called vmware.events.log

## Event methods

The SDK provides some utility methods to interact with the event manager:

cloud.register(event, function\_name, data)

Register a function name as a callback to handle the emitted event. An optional data object can be provided. Note that you can register multiple callbacks to a single event using this method.

cloud.once(event, function\_name, data)

This operates the same way as cloud.register except it will only call the function\_name once and not every time the event is emitted.

cloud.unregister(event, function\_name)

Unregister a function name as a callback to handle the emitted event.

cloud.trigger(event, data)

Trigger the callbacks registered to handle the emitted event. An optional data object can be provided.

The callback is given an object that contains the name of the emitted event, *key*, the optional data object given when registered, handerData, and the optional data object provides in this trigger method, eventData. An example of a callback function handling an emitted event would be:

function onLoginResult (loginEvent) {

console.log(’event emitted: ‘+ loginEvent.key)

console.log(‘handler data: ‘+ loginEvent.handlerData)

console.log(’event data: ‘+ loginEvent.eventData)

// logic to handle login event

}

# Starting the SDK

As described in the section above you instantiate the SDK like this

var cloud = new vmware.cloud(“https://10.0.0.1/api/",

vmware.cloudVersion.V5\_1);

You can use the event model methods to register a callback to handle the following events emitted after the SDK has been initialized.

## Events

vmware.events.cloud.INITIALIZATION\_COMPLETE

Emitted after validation of the vCloud Director API end-point as defined by the given URL.

For example, a callback to handle the *INITIALIZATION\_COMPLETE* event could be registered to run once to test if a vCD session already exists:

cloud.once(vmware.events.cloud.INITIALIZATION\_COMPLETE, function() {

console.log('SDK ready’);

cloud.confirmLoggedIn();

});

vmware.events.cloud.ERROR

This is emitted any time there is an error resulting from an SDK method. A string containing a localized error message is passed to the callback function, e.g.

cloud.register(vmware.events.cloud.ERROR, function(e) {

console.error('SDK error: '+ e.eventData);

});

# Authentication

You authenticate to the vCloud Director server using the login SDK method, e.g.

cloud.login(username, password, organization);

You can check that the session cookie has not expired using another SDK method called confirmLoggedIn, e.g.

cloud.confirmLoggedIn();

## The User object

The SDK stores basic information about the authenticated user in an object that has some useful methods to retrieve specific information:

cloud.getUserName();

cloud.getUserOrgName();

cloud.getUserOrgUrl();

cloud.getAdminUrl();

The first two methods simply return the string of the user and organization name.

The getUserOrgUrl method returns the REST URL, or href, of the organization. This can be useful if you want to create custom vCD API calls using the SDK to get more information about the organization and those vCD objects associated with it.

The getAdminUrl method returns the REST URL end-point, or href, used as a prefix for vCD API administrative calls. This is useful when making custom vCD API administrative calls but can also be used to test if the authenticated user has administrative rights, e.g.

if (cloud.getAdminUrl() !== undefined) {

console.info(‘User, ‘+ cloud.getUserName +’, has admin rights.’);

}

The SDK has methods to help make custom vCD API calls, explained later, if you need to get more details about users.

## Events

vmware.events.cloud.LOGIN

Emitted on completion of a login method call. An object passed to the callback function contains the success Boolean. The confirmLoggedIn method adds a confirm Boolean to the object passed to the callback function.

The following example shows how one callback could handle the LOGIN event emitted from the login and confirmLoggedIn methods:

function onLogin (e) {

if (e.eventData.success) {

if (!e.eventData.confirm) {

console.log('Logged into '+ vcd.getUserOrg() +' as '+

vcd.getUserName());

}

else {

console.log('Session still exists');

}

// continue with authenticated session...

}

else {

if (e.eventData.confirm) {

console.log('Session expired');

// continue as if user is not authenticated...

}

else {

console.log('Invalid credentials');

}

}

}

# The SDK internal data model

Once authenticated, the SDK will automatically fetch information about the vApps, Templates and VMs available to the user and store this information in an internal model. The SDK provides methods to work with the data in this model.

Using this model, the SDK can update the data independently of any UI display logic that needs to retrieve data from it. This is especially useful if there is any network lag or temporary disconnect since this data model acts as a cache. Any operations in the UI logic are not interrupted, unless there is a specific reason to communicate with the Cloud Director server, such as a power operation on a vApp.

## Reading the data model

The SDK provides methods to work with vApps, VMs and Templates stored in the data model. This is explained in more detail in a later section.

The SDK also provides methods to return arrays of these objects:

cloud.getVApps(cloud.SORTBY.DATE);

cloud.getVApps(cloud.SORTBY.NAME);

The getVApps method returns an array of vApp objects from the data model using a date or name sort order.

All VM objects are embedded in their associated vApp objects and can be read using logic similar to the following example.

var vapps = cloud.getVApps(cloud.SORTBY.DATE),

vapp = {},

vms = [],

vm = {};

for (var i=0; i<vapps.length; i++) {

vapp = vapps[i];

console.info(‘vApp ‘+ vapp.getName() +’ object...’);

console.dir(vapp);

vms = vapp.getChildren();

for (var j=0; j<vms.length; j++) {

vm = vms[j];

console.info(‘VM ‘+ vm.getName() +’ object...’);

console.dir(vm);

}

}

cloud.getCatalog();

The getCatalog method returns an array of Template objects from the data model.

## Refreshing the data model

The SDK will not automatically refresh the data model unless it needs to, e.g. updating the state of a vApp/VM because of a power operation or adding a new vApp because a Template was instantiated.

Therefore, it is up to the UI logic to use the methods the SDK provides:

cloud.updateModels();

The *updateModels* method will refresh the data for vApps and VMs in the data model.

cloud.getAllTemplates();

The *getAllTemplates* method will refresh the data for Templates in the data model.

Typically, the data for Templates does not change as frequently as that for vApps and VMs.

## Events

vmware.events.cloud.REFRESH\_COMPLETE

Emitted after the SDK has refreshed the vApp and VM information in the data model

vmware.events.cloud.TEMPLATE\_REFRESH

Emitted after the SDK has refreshed the Template information in the data model.

## Saving to and retrieving from HTML5 local storage

The SDK method, saveCache, creates a string version of this data model or cache. This can be saved somewhere, like the local storage provided HTML5 capable browsers, on log-out or after a session has expired, e.g.

localStorage.vcd\_cache = cloud.saveCache();

The cache can be restored using the SDK method, loadCache, e.g.

cloud.loadCache(localStorage.vcd\_cache);

Restoration is typically done after authentication when the SDK is still making asynchronous calls to refresh the data model so that the UI rendering logic is not waiting for data to appear in the model.

## Common methods

The SDK provides some common methods for the vApp and VM objects stored in the data model. These methods include:

isVM();

The *isVM* method returns *true* if the object is a VM.

getName();

The *getName* method returns the text of the vApp/VM name.

getDescription();

The *getDescription* method returns the text of the vApp/VM description.

getStatus();

getStatusMessage();

The *getSatus* method returns an integer representation of the status of the vApp/VM. This status is defined by the following properties in the SDK:

STATUS\_ON = 0;

STATUS\_OFF = 1;

STATUS\_WORKING = 2;

STATUS\_SUSPENDED = 3;

STATUS\_ERROR = 4;

STATUS\_PARTIAL = 5;

More usefully, the *getStatusMessage* method returns a localized string of the vApp/VM status.

canPowerOn();

canPowerOff();

canSuspend();

The *canPowerOn*, *canPowerOff* and *canSuspend* methods return *true* if the associated power state action can be performed on the vApp/VM object.

powerOn();

powerOff();

suspend();

The *powerOn*, *powerOff* and *suspend* methods perform the associated power operation on the vApp/VM object.

getID();

The *getID* method returns the ID of the vApp/VM object.

getHref();

The *getHref* method returns the REST URL of the vApp/VM object.

setAttr(key, value);

The *setAttr* method changes the value of the given attribute key of the vApp/VM object. It can also add new key/value pairs to the vApp/VM object.

getAttr(key);

The *getAttr* method returns the value of the given attribute key for the vApp/VM object.

favorite(val);

The *favourite* method, without the *val* parameter, will return the value stored in the *favorite* vApp/VM object property. The same method, with the *val* parameter, will set the *favorite* property to that value. A method to toggle this object property could look like the following:

toggleFavorite = function (vappObj) {

var val = 1;

if (vappObj.favorite() == 1) val = 0;

cloud.metadata.register(cloud.metadata.set(vappObj, 'favorite', val),

function() {

vappObj.favorite(val);

});

};

vCD metadata support in the SDK is discussed in a later section.

## vApp methods

In addition to the common methods described above, the SDK provides the following methods for a vApp object:

getNumberOfVMs();

The *getNumberOfVMs* method returns the number of VMs in the vApp.

getOwnerName();

The *getOwnerName* method returns a string of the user name that owns the vApp.

getCreationDate();

The *getCreationDate* method returns a string of the creation date of the vApp.

getVDCName();

The *getVDCName* method returns a string of the VDC name that provides resources for the vApp.

getChildren();

The *getChildren* method returns an array of VM objects contained in the vApp.

edit(name, description);

The *edit* method changes the name and description of the vApp to those given.

canDelete();

The *canDelete* method return *true* if a delete operation is allowed on the vApp. The SDK provides a *deleteVApp* method to delete a vApp, e.g.

cloud.deleteVApp(vappObj.getID());

## VM methods

In addition to the common methods described above, the SDK provides the following methods for a VM object:

getIP();

The *getIP* method returns the IP address for the primary network interface of the VM.

getNetwork();

The *getNetwork* method returns the string of the network name that the primary network interface of the VM is connected to.

getGuestOS();

The *getGuestOS* method returns the string of guest operating system of the VM.

getThumbnailImage();

The *getThumbnailImage* method returns a data URI string representing a thumbnail image of the VM console screen.

## Template Methods

The SDK provides the following methods for a Template object:

getName();

The *getName* method returns the text string of the Template name.

getDescription();

The *getDescription* method returns the text string of Template description.

getHref();

The *getHref* method returns the REST URL of the Template object.

getChildren();

The *getChildren* method returns an array of VM objects contained in the Template.

getOwnerName();

The *getOwnerName* method returns a string of the user name that owns the Template.

getCatalogName();

The *getCatalogName* method returns a string of the Catalog name the Template is indexed by.

getNetwork();

The *getNetwork* method returns the string of the network name used to connect to the primary network interface of the VM before the Template was created.

getCPUMhz();

getMemoryMB();

getStorageKB();

The *getCPUMhz*, *getMemoryMB* and *getStorageKB* methods return resource metrics as identified by the Template for it to be successfully instantiated.

setAttr(key, value);

The *setAttr* method changes the value of the given attribute key of the Template object. It can also add new key/value pairs to the Template object.

getAttr(key);

The *getAttr* method returns the value of the given attribute key for the Template object.

# Other stored data

The SDK stores other data that that explained above. This data can be considered independent of that in the data model and, as such, separate methods are provided by the SDK for the UI logic to use at will.

cloud.taskHistory();

cloud.metrics();

cloud.getNetworks();

cloud.getVdcList();

The taskHistory method returns an array of tasks performed within the organization.

The metrics method returns an array of statistics about the organization.

The *getVdcList* method returns an array of VDCs available to the authenticated user in this organization.

The *getNetworks* method returns an array of VDC Networks available to the authenticated user in this organization.

When the SDK understands that you have an authenticated session with vCD, as described earlier, it will make all the nessesary vCD API calls to populate the data model, but it will also make the calls to populate the task history, metrics, etc. talked about here. The SDK makes this operation available using the *begin* method:

cloud.begin();

This can be a very time-consuming operation, should not be done on a frequent basis and performed only when a full refresh of the data is needed.

# Custom vCD API calls

The SDK provides a generic method to make your own calls to the vCD API. You should use the complete vCD API reference (<http://www.vmware.com/go/vcloudapi>) to understand what REST URLs to use and what responses to expect.

The method to make a vCD API call is:

cloud.fetchURL(url, method, acceptType, callback\_function);

This *fetchURL* method wraps the jQuery ajax method which handles the request header definition and parses out the xml from the response, passing this to a callback you define. An example using this method might be:

var url = cloud.base

+ 'query?type=user&format=records&filter=name=='

+ vcd.getUserName(),

userDetail = {};

cloud.fetchURL(url, 'GET', null, function (xml) {

userDetail = $(xml).find('UserRecord');

console.info(‘User ‘+ userDetail.attr('name') +’ object...’);

console.dir(userDetail);

});

In this example, the vCD API query method is used. A REST URL is constructed to fetch the *UserRecord* for our authenticated user. Note that the *base* property provided by the SDK returns the vCD API end-point URL. This URL is then used in the SDKs *fetchURL* method to make the asynchronous vCD API call. A callback, the anonymous function, is also passed to the *fetchURL* method to process the response from the vCD API call.

# Metadata

Metadata is a feature of vCloud Director 5.1 used to add custom data to vCloud objects. You can defined typed key-value pairs and associate them with vApps, VM, etc.

GOT UP TO THIS POINT

It is used on vApps and Templates to provide information on number of downloads, favorites and featured library items. The code is in the metadata.js file. Note that this implementation does not work with the date/time typed metadata.

To get the metadata associated with an object

get(object);

To set the object’s key to value:

set(object, key, value);

Register callbacks for metadata actions, which will be triggered on completion:

cloud.metadata.register(cloud.metadata.set(vapp, 'favorite', val), function() {

vapp.favorite(val);

self.refresh();

});

# Templates and Catalog

To pull templates from a catalog, there are several options as for a query. The most useful and overarching request is as follows:

/api/query?type=vAppTemplate&format=records&page=1&pageSize=128

The default pageSize is 25, and the maximum is 128. It doesn't tell you how many pages there are, but there is a link to the next page, so you can move through the entire catalog. We currently only work with one catalog.

Once you have templates, you have to pass the link to the template as well as some XML to instantiate it:

/api/vdc/vdc-id-here/action/instantiateVAppTemplate

This XML determines the name, description and other attributes of the vApp you wish to instantiate. An example follows.

<?xml version="1.0" encoding="UTF-8"?>

<InstantiateVAppTemplateParams

xmlns="http://www.vmware.com/vcloud/v1.5"

name="Linux FTP server"

deploy="true"

powerOn="true"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:ovf="http://schemas.dmtf.org/ovf/envelope/1">

<Description>Example FTP Server</Description>

<InstantiationParams>

<NetworkConfigSection>

<ovf:Info>Configuration parameters for logical networks

</ovf:Info>

<NetworkConfig

networkName="vAppNetwork">

<Configuration>

<ParentNetwork

href="https://vcloud.example.com/api/network/54" />

<FenceMode>bridged</FenceMode>

</Configuration>

</NetworkConfig>

</NetworkConfigSection>

</InstantiationParams>

<Source

href="https://vcloud.example.com/api/vAppTemplate/vappTemplate-111" />

Actions on templates defined in model.js

getAttr(key)

save()

load()

getHref()

getName()

getDescription()

getDownloads()

getNetwork()

getChildren()

getOwnerName()

getCatalogName()

getCPUMhz()

getMemoryMB()

getStorageKB()

The arguments to instantiate a vApp from a template include the name of the vApp, description, networkName, parentNetwork and the address of the template that we want to instantiate. In addition, a flag to power it on after instantiation is included.

NOTE: the networkName and/or parentNetwork must correspond with the networks specified in the template.

instantTempParams(name, description, networkName, parentNetwork, href, powerOn)

# Tasks

[to do]

# Search

[to do]

# Localization

A simple technique is used for presenting text in a localized fashion. The file localizer.js defines the phrases that can be translated from a key to a string value. Note that not all text is localized. Here are the items localized in the distribution:

localizer.setlang(localizer.languages.EN)

.add("SYSTEM\_NOT\_SUPPORTED", "You are attempting to connect to a system no longer supported.")

.add("LOST\_CONNECTIVITY", "You seem to have lost connectivity - your changes will not be saved until connectivity is reestablished.")

.add("STATUS\_ON", "Powered On")

.add("STATUS\_OFF", "Powered Off")

.add("STATUS\_ERROR", "Error!")

.add("STATUS\_WORKING", "Working...")

.add("STATUS\_PARTIAL", "Partially On")

.add("STATUS\_SUSPENDED", "Suspended")

.add("TASK\_POWERING\_ON", "Powering On")

.add("TASK\_POWERING\_OFF", "Powering Off")

.add("TASK\_CONSOLE", "Connecting To Console")

.add("TASK\_SUSPEND", "Suspending")

.add("TASK\_CREATE\_VAPP", "Creating vApp")

.add("TASK\_DELETE", "Deleting")

.add("TASK\_UNDEPLOY\_VM", "Undeploying VM")

.add("TASK\_DEPLOY\_VM", "Deploying VM")

.add("TASK\_UNDEPLOY\_VAPP", "Undeploying vApp")

.add("TASK\_DEPLOY\_VAPP", "Deploying vApp")

.add("TASK\_EDIT\_VAPP", "Editing vApp")

.add("TASK\_UPLOAD\_OVF", "Uploading OVF");

To use the localized text call the function:

localizer.get("STATUS\_ON");