JavaScript API to vCloud Director

Table of Contents

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.

You can register a function as a callback to handle an emitted event by using the register method:

```
cloud.register(event, function_name, data);
```

The optional *data* object can be provided which will be passed through to the callback as part of the event object.

Note that you can register multiple callbacks to a single event using this method.

Registering a function as a callback that handles the response to the SDK login method would look like this:

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

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

You might only want an event to trigger a callback once and not every time the event is emitted. This is handled using the *once* method shown here:

```
cloud.once(event, function_name, data);
```

To unregister a function as a callback to handle the emitted event you use the following *unregister* method:

```
cloud.unregister(event, function_name);
```

To emit an event, you use the *trigger* method which sill fire the callbacks registered to handle this event:

```
cloud.trigger(event, data);
```

The optional data object can be provided and will be passed through to the callback as part of the event object.

A callback is given an object that contains the event name (*key*), data objects provided in the registration methods (*handerData*) and data objects provided in the trigger methods (*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`

The ERROR event 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);
        }
    }
```

The SDK doesn't fetch a list of vCD catalogs; instead the *getCatalog* method returns an array of template objects that exist within all the catalogs available to the authenticated user:

```
cloud.getCatalog();
```

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. Each object contains attributes built from the XML response of the query made by the SDK. While the SDK provides a *get* method to retrieve the values of these attributes, there are some purpose made methods which 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 *favorite* method offers a way to flag a vApp/VM. Without the *val* parameter, it will return the value stored in the objects favorite attribute. The same method, with the *val* parameter, will set the favorite property to that value. A function to toggle this favorite attribute and use its value to set a vCD metadata property on the vApp object 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. The thumbnail size supplied from vCD is 64px by 48px.

Template Methods

The SDK provides the following methods for a Template object. As with the vApp/VM objects, each Template object contains attributes built from the XML response of the query made by the SDK. While the SDK provides a *get* method to retrieve the values of these attributes, there are some purpose made methods which include:

Alister Lewis-Bowen 1/11/13 4:51 PM
Deleted: :

```
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

Other data is retrieved by the SDK. 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.

```
cloud.taskHistory();
cloud.metrics();
cloud.getVdcList();
cloud.getNetworks();
```

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 necessary vCD API calls to populate the data model, but it will also make the calls to populate the task history, metrics, VDCs and VDC Networks described here. The SDK initiates this operation by 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 define typed key-value pairs and associate them with vApps, VM, etc.

The SDK provides a simple way to get the metadata from an vApp or VM object using the *get* metadata method, e.g.

```
cloud.metadata.get(object);
```

Likewise, a *set* metadata method is also provided:

```
cloud.metadata.set(object, key, value);
```

The SDK metadata service includes its own event management system that you use in conjunction with any *get* or *set* method calls, e.g.

```
cloud.metadata.register(method, callback);
```

An example to retrieve a 'favorite' metadata value from all vApps available to you and using the vApp object *favorite* method to set it in the SDK internal data model might look like the following:

```
    var vapps = cloud.getVApps(vcd.SORTBY.DATE);

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

        cloud.metadata.register(
            cloud.metadata.get(vapp), function (data) {
                if (data.favorite !== undefined) vapp.favorite(data.favorite);
                // good place to refresh the UI
        });
    }
```

Assuming the UI logic can set the favorite attribute for a vApp, the following could be used to set this metadata on the specific vApp:

```
    cloud.metadata.register(
        cloud.metadata.set(vapp, 'favorite', value), function (data) {
            vapp.favorite(value);
            // good place to refresh the UI
    });
```

Creating a vApp from a Template

VMware vCloud Director allows Users to create new vApps from templates. The SDK provides a method to perform this instantiation, e.g.

```
cloud.instantiateVApp(name, description, vdcName, networkName, templateUrl,
    powerOn);
```

This *instantiateVApp* method will respond with true if the initiate validation is successful. Then vCD will respond with a task that can be examined with the SDK's task manager as explained below.

The name and description are text strings but remember that the name should be unique. The VDC name should be taken from the array of available VDCs provided by the SDK. Similarly, the Network name should be taken from the array of available Networks provided by the SDK. The template URL is found in the template object and the power on parameter is a Boolean flagging whether the vApp should be powered on after instantiation.

A simple example of how a vApp might be created given an existing template object might be:

```
    var tmpl = cloud.getCatalog()[0],
        name = tmpl.getName() +'-'+ Math.floor(Math.random()*1000),
        desc = tmpl.getDescription() || '',
        vdc = cloud.getVdcList()[0],
        network = cloud.getNetworks[0],
        template = tmpl.getHref(),
        powerOn = true;
    if (cloud.instantiateVApp(name, desc, vdc, network, template, powerOn)) {
        console.debug('SDK is instantiating vApp: '+ name);
    }
    else {
        console.debug('SDK could not instantiate template: '+ tmpl.getName());
    }
```

Tasks

VMware Cloud Director uses tasks to track the progress of long-running operations. For example, when a power operation is performed on a VM or when a vApp is instantiated from a Template, a task is created. The SDK contains a very simple task manager that keeps track of these tasks and offers methods to access the details of each task.

An array of recent tasks summaries can be retrieved using the *taskLog* task manager method, e.g.

```
cloud.taskManager.taskLog();
```

Each task summary contains the array of elements; owner, action description, timestamp, status, task URL. Given the task URL, a task object can be retrieved using the *details* task manager method, e.g.

```
cloud.taskManager.details(taskUrl);
```

Creating an array of running task objects might look the following:

```
    var running = [];
    var tasks = cloud.taskManager.taskLog();

    for (var i=0; i<tasks.length; i++) {
        if (tasks[i][3] === 'running') {
            running.push(cloud.taskManager.details(tasks[i][4]));
        }
    }

    console.dir(running);
```

The task object includes information about the object being worked on, the organization within which the task was performed, the name of the user that started the task, etc.

Another useful method lists the vApps, VMs and Templates IDs that have associated running tasks:

```
cloud.taskManager.inProgress();
```

This could be used to assign a busy state to a list of vApps or VMs.

While the task manager will periodically update its task store, a method is supplied to force an update:

```
cloud.taskManager.update()
```

This is useful to call from any manual refresh control you may implement in the UI logic.

Events

```
vmware.events.cloud.TASK_START
```

Emitted after the SDK notices a task has been started.

```
vmware.events.cloud.TASK_COMPLETE
```

Emitted after the SDK notices a task has been completed.

Search

The SDK provides methods to search for templates within the SDK internal data model. A search can be performed using a general term, specific facets (name, cpu range, etc.) or a combination of both.

To perform a general search using a search term string, you would use the *searchCatalog* method shown here:

```
cloud.searchCatalog(seachTerm);
```

The *searchCatalogFaceted* method allows you to define an object of attributes or facets to be searched for. The facets are *name*, *owner*, *catalog*, *cpu*, *memory* and *storage*. You can also include *searchTerm* as another attribute. The *cpu*, *memory* and *storage* facet values can be a range, e.g. 256-1024. The following shows how you can define facets for the *searchCatalogFaceted* method to look for templates that have up to 2Mhz of CPU:

```
cloud.searchCatalogFaceted({
    'name': '',       // template name
    'owner': '',      // template owner username
    'catalog': '',    // catalog name
    'cpu': '0-2',     // cpu Mhz range, '-'=any, e.g. '0-2', '3-4', '4-'
    'memory': '-',    // memory MB range, '-'=any, e.g. '0-127', '128-256',
'256-1024', '1024-'
    'storage': '-',   // storage KB range, '-'=any, e.g. '0-199999', '200000-
399999', '4000000-'
    'searchTerm': '' // general search term which is not a facet
});
```

Events

```
vmware.events.cloud.SEARCH_COMPLETE
```

Emitted after the SDK has completed a search for templates.

Localization

The SDK provides a very simple mechanism to provide localized text. However, currently any updates to localization requires that you change the SDK source code, re-compile the SDK, and replace the updated SDK library file in the directory your web application expects it to be.

In the *src/* directory of the SDK source tree, the *text.js* file defines the *localizer* object, language registration and translation mappings.

To define a new language, add a key/value to the *localizer.languages* object. You will see the EN key already there. To add French for example, you use the standard language abbreviation, FR and increment the value to 1, like this:

```
    languages: {
        EN: 0,
        FR: 1
    }
```

You introduce a new language to localize using the *localizer.setlang* method, e.g.

```
    localizer.setlang(localizer.languages.FR);
```

To add a mapping, you provide a token and its associated language string using the *localizer.add* method, like this:

```
    localizer.setlang(localizer.languages.FR)
        .add("STATUS_ON", "Mis sous tension");
```

You continue to add these mappings to provide a full translation of the tokens. Currently the tokens used in the SDK are indicated in the following default mappings:

```
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");
```

The *localizer.get* method is used to retrieve the localized text in the rest of the SDK source code, e.g.

```
    localizer.get('STATUS_ON');
```