Hewlett Packard Enterprise

Cloud Service Automation
Edition (optional)

Dynamic Options and Cascading Lists

Software version: 4.70

Document release date: July 2016

Software release date: July 2016

Contents

Introduction

Subscriber options allow designers to create option sets during modeling a service design. These option sets are exposed to subscribers or consumers, which allow the subscribers or consumers to select an option value in the Marketplace Portal. The option can be a String, Boolean, Integer or a List type. This white paper outlines various choices available to populate the list properties.

Consider an infrastructure cloud offering which provisions a simple server. A subscriber has to choose an operating system (OS) and OS Edition from a selection list. The designer can model the option list and load the list statically at design time or dynamically load it from an external source like a file or a database.

This white paper covers the following cases with examples:

- Populate a list at design time – static loading
- Populate a list from an external source – dynamic loading

Note: This white paper assumes you are familiar with the Cloud Service Management Console and Marketplace Portal.

Dynamic loading

An option list property can be populated by capturing data from an external source like a file or a database. The list property is associated with a JavaScript (JS) file which embeds the logic to capture data from an external source and wraps the data as XML. CSA provides a framework to execute JavaScript in the context of JBOSS and returns XML as an HTTP response.

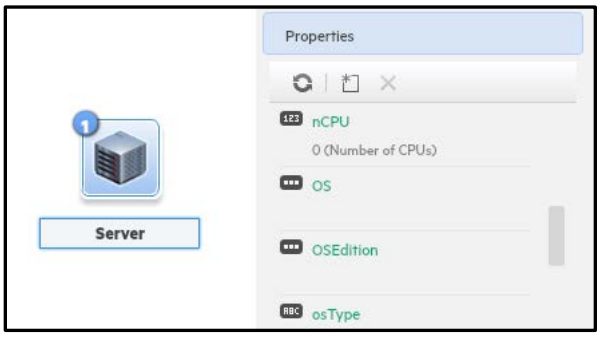
When an offering is requested in the Marketplace Portal, the subscriber options in the Marketplace Portal are loaded, triggering an HTTP request call to the JavaScript. CSA executes the script and returns XML data to populate the list in the Marketplace Portal.

Component properties

Consider that you have created a service design that has a Server component. Create two list properties, **OS** (Operating System) and **OSEdition** on the server component.

1. In the Server (Component), in the **Properties** tab, click the **Create New Property** icon.
2. Enter the following information in the Create Property wizard and click **Create**:
   - **Type**: List
   - **Name**: OS
   - **Display Name**: OS
3. Repeat these steps to create the OSEdition list property.

The OS and OSEdition properties appear as List properties in the Server Properties list.



Subscriber options

Create a subscriber option set called **Server Deployment** and add an option called **Server**.

1. Select the **Subscriber Options** tab and click **Add Option Set**.

2. Enter **Server Deployment** in the **Display Name** field.
3. In the Server Deployment option set, click the **Add Option** button and enter **Server** in the **Display Name** field and click **Save**.

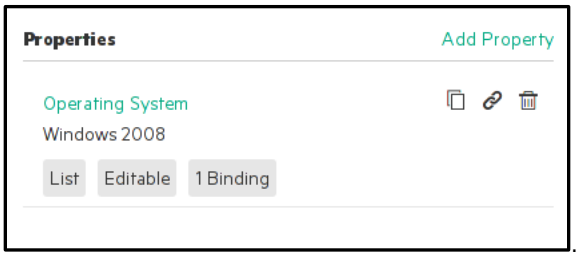
Static loading

In the **Add Property** area, create a list property called **Operating System** for the subscriber option, **Server**.

1. Click **Add Property** to add a list property.
2. Enter the following information, then click **Done**. Static loading is the default:
   - **Property** Type: List
   - **Name**: OperatingSystem
   - **Display** Name: Operating System
   - Click **Editable**
   - Click **Single Select**
   - Under **List Items**, enter the following operating system names and values:

Display Name	Value
Windows 2008	Windows 2008
Windows 2012	Windows 2012

3. Bind the Operating System property to the OS property on the Server component.
4. Click **Save**



Limitations on the static list property

The following table identifies the character length of attribute values:

Display Name	Value
Display Name	255
Description	255
Value	4000

Dynamic loading

CSA includes out-of-the-box scripts from which you can select for dynamic loading. You can also add new scripts using the Manage Scripts dialog. All the JavaScript files available in this dialog are stored in the CSA database. Scripts are visible across all organizations in the Marketplace Portal.

The script is invoked at subscription ordering or modification time by the out-of-the-box CSA user `csaReportingUser`, who has read-only access to CSA. For more information about this user, see the *Cloud Service Automation Configuration Guide*.

Contents of the script

To load a list of key-value pairs like (key1, name1) or (key2, name2) into the option property, the script needs to wrap the key-value pairs in XML as shown below. The script must write this XML to its standard output (body of HTTP response).

```
<Property>
        <availableValues>
                <value>key1</value>
                <displayName>name1</displayName>
                <description>Key description</description>
                <initialPrice>11</initialPrice>
                <recurringPrice>1</recurringPrice>
        </availableValues>
        <availableValues>
                <value>key2</value>
                <displayName>name2</displayName>
                <description>Key description</description>
                <initialPrice>12</initialPrice>
                <recurringPrice>2</recurringPrice>
        </availableValues>
</Property>
```

Each **availableValues** tag describes one item in the list of the dynamic list property. The tags **value**, **displayName** and **description** describe how the value is processed and displayed.

The tags **initialPrice** and **recurringPrice** are optional. These tags describe how each item affects the price when it is selected as an offering in the Marketplace Portal. Currency of these values and recurrence period are part of the service offering.

XML may be produced in the script either by using the print/println functions (see Appendix C - JavaScript sample code) or preferably by a more JavaScript-friendly way as follows:

- The script creates a variable **availableValues** with a value containing an array of objects where each object contains some of these keys: **value**, **displayName**, **description**, **initialPrice**, **recurringPrice** with strings as values.

Server edition script

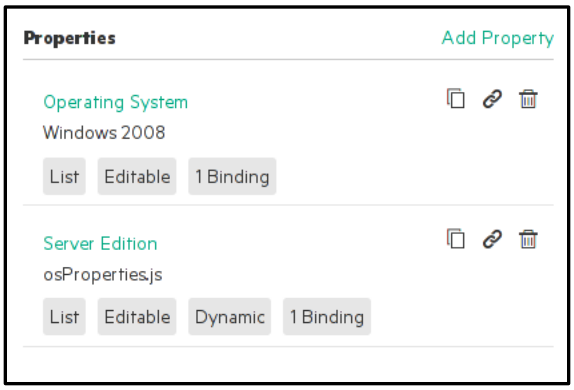
In this scenario, you will return values **Standard** and **Enterprise** as choices for the operating system edition. The script below will return these values. Save the content to the file `osProperties.js` and upload this file via the Manage Scripts dialog.

```
availableValues = [ {
    'value' : 'Standard’,
    'displayName' : 'Standard',
    'description' : 'Standard'
},  {
    'value' : 'Enterprise',
    'displayName' : 'Enterprise',
    'description' : 'Enterprise'
} ];
```

Dynamic property

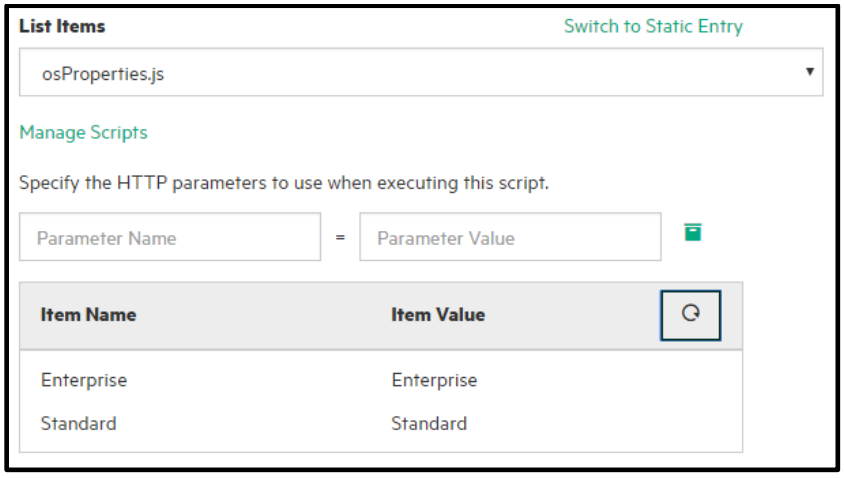
Create a list property called **Server Edition** and configure the property using the **Switch to Dynamic Entry** option.

1. Enter the following information and then click **Done**:
   - **Property Type:** List
   - **Name:** ServerEdition
   - **Display Name**: Server Edition
   - Click **Single Select**
   - Click **Switch to Dynamic Entry**
   - Under **List Items**: select a JavaScript script to use for the dynamic values. In this case, select the `osProperties.js` script that you created in the Server edition script section above.
2. Bind the Server Edition property to the OSEdition property on the Server component.
3. Click **Save**.



Test the Dynamic Query

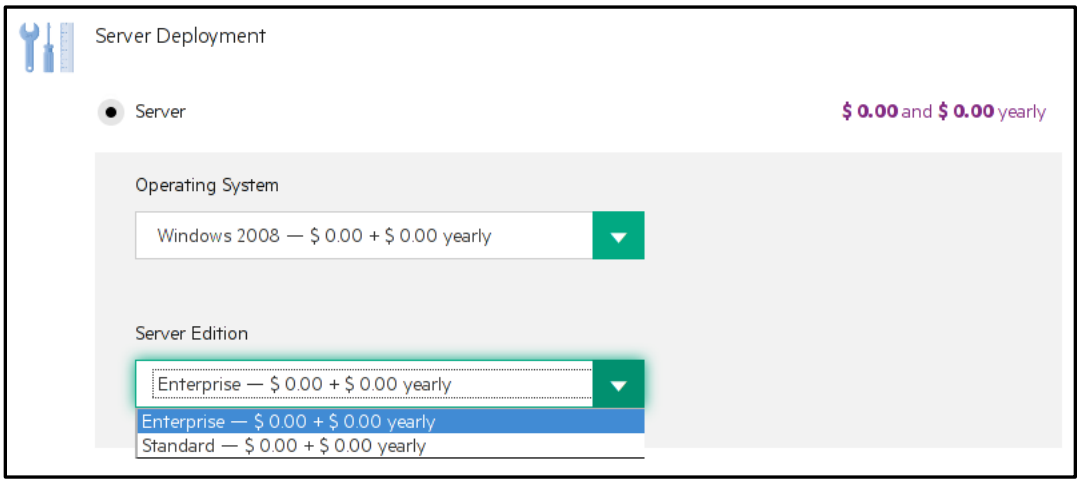
CSA provides the capabilities to test the JavaScript script on the Designer. Click the **Refresh Data** button to validate the script.



See **Appendix A** for information on developing and debugging the JavaScript scripts. See **Appendix C** for additional sample JavaScript code snippets.

Dynamic list in the Marketplace Portal

In the Marketplace Portal, the dynamic list will display as shown below:



Limitations

Scope of dynamic list

The dynamic list feature of populating property values can only be used in the Option model. This feature is not available for properties defined on components, providers, or internal actions.

Also, there is no limitation to the number of option-sets or options in Option model. But there is a limitation on the number of nested options that be defined with an option set; for example, an option within an option. CSA allows only three levels of nested options in the model.

Time to load data from a script

CSA has defined a property to set a default time limit to load a script. CSA will terminate the execution of the script after the default time and return a blank response to the Marketplace Portal. The default time limit can be updated by changing the property in the file as shown below:

Property	**`DynamicPropertyFetch.READ_TIMEOUT=300000`**
File	`csa.properties`
Location	`CSA_HOME/jboss-as/standalone/deployments/csa.war/WEB-INF/classes`

Size of XML data

CSA defines a property to limit the XML response size from a script. An XML response greater than the defined limit will throw an exception in the Marketplace Portal. The limit value can be updated by changing the property in the file as shown below:

Property (character length)	**`DynamicPropertyFetch.RESPONSE_SIZE=50000`**
File	`csa.properties`
Location	`CSA_HOME/jboss-as/standalone/deployments/csa.war/WEB-INF/classes`

Note: name-value pairs are wrapped as XML `<Property>…</Property>` elements. XML tags add to the overall payload of the response size. `<Property>…</Property>` XML alone contributes around 100 characters in length. For example: “**name”** is a four character length and **“value”** is a five character length that when wrapped into `<Property>…</Property>` elements transforms into approximately 110 characters. Therefore, set the **DynamicPropertyFetch.RESPONSE_SIZE** appropriately.

Cascading options

Consider an offering for an infrastructure service where the choices for Server Edition are different for each of the choices for the Operating System (OS). For example, for this scenario, the OS choices are Windows and RHEL. You will offer the choices of Standard or Enterprise for Windows, and Server or Desktop for RHEL.

The Server Edition list should be filtered based on the OS selected.

The following table summarizes the subscriber options to be displayed:

Operating System	**Server Edition**
Windows	Standard, Enterprise
RHEL	Server, Desktop

Here you need to change the script and the subscriber option properties.

To change the subscriber option to serve these different choices, you must modify the `osProperties.js` script and the properties. You can create a new version of the design for the modifications, or you can edit the existing design. However, for this scenario, these values will be updated as described below.

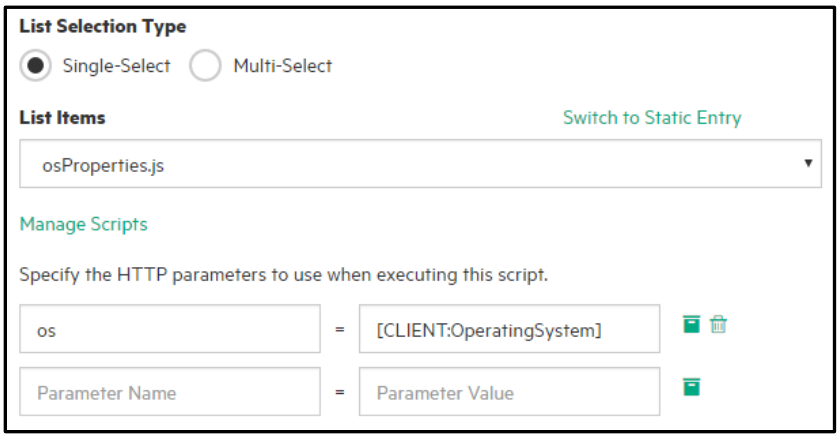
Note: See the Component properties section to create list properties on a component. Also see the Static loading section to populate a list of OS values on the option property. See the Dynamic loading section to populate a list of Server Edition values on the option property.

In the **Operating System** property, modify the **List Items** to contain the values in the table:

Display Name	Value
Windows	Windows
RHEL	RHEL

The **Server Edition** property is dependent on the **Operating System** to list its values. Therefore, the **Server Edition** needs the value of the **Operating System** property to be passed to its script. To pass the selected value from one property to another, use a HTTP request parameter. The value of the parameter is in the form of `[CLIENT:<name_of_property>]` where `<name_of_property>` is the property whose value controls the selection of the value in the dynamic property. It is the **Name** field of the property, not **Display Name**.

In this example the property **Server Edition** uses the parameter **os** and the passed value is **`[CLIENT:OperatingSystem]`** as shown below. Modify the **Server Edition** property and click **Done**. Click **Save** to save the subscriber options.



The JS script then finds this parameter in the **request** variable, and gets the value by calling **request.os**. This parameter variable will have one of two values, **Windows** or **RHEL**. Below is the new version of the `osProperties.js` script:

```
function value(osName, edition) {
    return {
        'value': edition,
        'displayName': edition,
        'description': osName+' '+edition
    };
}

var availableValues = [];
var osName = request.os;
switch(osName) {
    case 'Windows':
        availableValues.push(value(osName, 'Standard'));
        availableValues.push(value(osName, 'Enterprise'));
        break;
    case 'RHEL':
        availableValues.push(value(osName, 'Server'));
        availableValues.push(value(osName, 'Desktop'));
        break;
}
```

Test the Dynamic Query

To test the dynamic query and validate the script, complete the following steps:

1. Click the **Refresh Data** button to validate the script. While testing the script in the Designer, the actual value sent to the script in the **os** parameter will be the selected value in the **Operating System** property that is saved on the server.
2. To change the selection, do the following:

a. Close the Edit Property dialog for the **Server Edition** property.

b. Open the **Operating System** property and select another value in **List Items**.

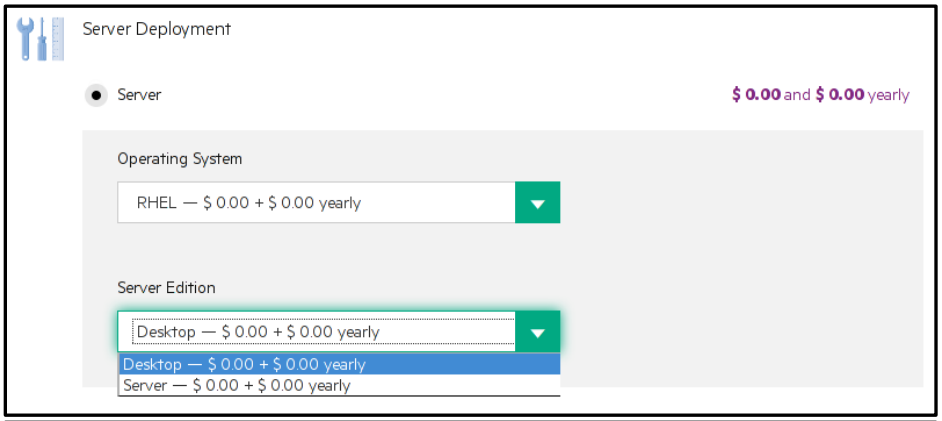
c. Click **Done**. Remember to click **Save** to save the subscriber options.

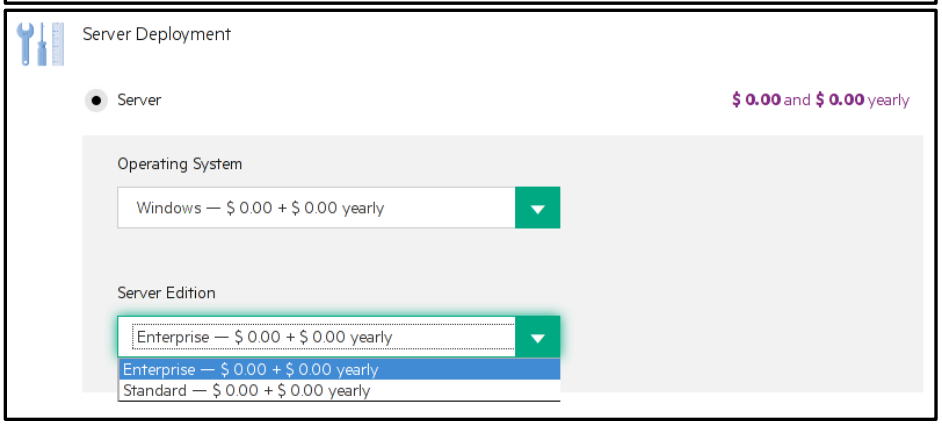
3. Open the **Server Edition** property and click the **Refresh Data** button.

Another way to test the script temporarily is to put the value directly to the Parameter Value box instead of `[CLIENT:OperatingSystem]`. Use the parameter **os** and parameter value **RHEL** (or **Windows**) and click the **Refresh Data** button. This way you do not need to switch to the other property to test different values.

Dynamic list in the Marketplace Portal

In the Marketplace Portal, the dynamic list will display the choices for the **RHEL Operating System** property and the **Windows Operating System** property, as shown below:





Limitations

Cascading option within an option set

The option values can be passed from the parent to the child list within an option set. You cannot pass option values across option sets.

Resolving multiple option values

Cascading option values in a child can resolve only one parent option. Passing multiple parent option values into a child option will result in an error.

For example: Passing multiple OS values into a child option will result in an error as shown below.

```
os1=[CLIENT:OperatingSystem1]& os2=[CLIENT:OperatingSystem2]
```

CSA Helper API

Server tokens

CSA supports passing artifact IDs/token IDs as HTTP request parameters into the JavaScript. The value of each parameter takes the form of `[TOKEN:<token_name>]`, or `[PORTAL:<token_name>]`. To choose a supported token, click the Select Token button to the right of the parameter to open the list of supported tokens, and select the token you want.

Secure access to various resources (credentials for remote HTTP/S access, limited file access, limited DB access)

Object **ResourceAlias**: used to refer to entries in the `resource-alias.xml` file, which holds various configurations for resources such as database connection, filesystem access, credentials. Resource entry in `resource-alias.xml` may look as follows:

```
<entry name='myuser for servername'>
        <username>myuser</username>
        <requiredProtocol>https</requiredProtocol>
        <requiredURLPrefix>https://servername/somepath/</requiredURLPrefix>
        <password>ENC(3oKr9eADA7bE53Zk2t9wIA==)</password>
</entry>
```

Note: To encrypt your own password, run the password utility available among the CSA tools, for example:
`java -jar passwordUtil.jar encrypt <my_password>`

Object is then instantiated like this:

```
new ResourceAlias('myuser for servername')
```

Resource alias is consumed by the HTTP/S client, JDBC database access, Filesystem access functions.

All the tags inside the entry are optional. In this case an entry is specified (using the `requiredProtocol` tag) to be usable with only https (to specify both http and https, the `requiredProtocol` tag may be duplicated with other values). Tag `requiredURLPrefix` specifies any URL with which it is used has to start with this prefix. If multiple tag `requiredURLPrefix` are specified the actual URL must match it prefix with at least one of them. Prefix matching does not allow actual URL to use path components like ".." to go outside of the prefix. As alternative, URL may be restricted using tag `requiredURL`. In that case exact equality is required. Multiple tags `requiredURL` are possible.

Username and password in the HTTP/S client are interpreted as HTTP Basic authentication credentials. Passwords allow encryption (same encryption as the passwords in the `csa.properties` file). They can also use syntax ${property.name} to refer to properties in `csa.properties` file. Tag `baseURL` (not visible in example above) allows administrator to specify URL that is validated in same way as single occurrence of tag `requiredURLPrefix`, but changes meaning of actual URL parameter passed to client libraries so that it is only relative to this `baseURL` (useful to avoid repeating hostname in the configuration and in the script).

Other protocols that are recognized:

- **jdbc** - when used with SQLClient.
- **file** - when used with FileStorage.

Attributes for controlling unrestricted access

The Resource alias XML file starts with the root element <*resource-alias*>. The <*resource-alias*> element has three optional attributes:

```
<resource-alias unrestrictedHTTPClient="false"
allowProcessEngineAccess="false"
allowResourceProviderAccess="false">
```

All of these resource attributes default to secure choices (value "false").

Meaning of the attributes:

- **unrestrictedHTTPClient** - If false, a HTTPClient can be used only with valid resources alias. If true, JS dynamic property scripts can make unrestricted connections using HTTPClient.
- **allowProcessEngineAccess** - if true, a resource alias returned by `CSAIntegrationHelper.getAccessPointForProcessEngine` will be allowed to be used with HTTPClient to make unrestricted connections to any process engine.
- **allowResourceProviderAccess** - if true, a resource alias returned by `CSAIntegrationHelper.getAccessPointForResourceProvider` will be allowed to be used with HTTPClient to make unrestricted connections to any resource provider.

CSAIntegrationHelper

Object **CSAIntegrationHelper**: provides access to usual tasks that dynamic properties might need. `CSAIntegrationHelper` contains these functions `getProcessEngineNames`, `getOrgName`, `csaReportingUserId`, `getUserEmail`, `getAccessPointForProcessEngine`, `getAccessPointForResourceProvider`, `getUsername`, `getResourceProviderIds`, `getCsaReportingUserId`.

```
var user = CSAIntegrationHelper.getUsername('90d96588360da0c701360da0f1d600a1');
```

The functions `getAccessPointForProcessEngine` and `getAccessPointForResourceProvider` return a token that can be used to configure HTTP/S client and call provider endpoint, but they do not expose passwords. Arguments to these two functions are IDs of respective database objects.

Function	ResourceAlias **getAccessPointForProcessEngine**(String processEngineName) Given a process engine name, this method returns the token for HTTPClient.
Parameters	processEngineName - a String object representing the name of the process engine
Returns	Token that can be used to configure HTTP/S client and call process engine endpoint

Function	ResourceAlias **getAccessPointForResourceProvider**(String resourceProviderId) Given a resource provider id, this method returns the token for HTTPClient.
Parameters	resourceProviderId - a String object representing the id of the resource provider
Returns	Token that can be used to configure HTTP/S client and call resource provider endpoint

Function	String **getUsername**(String userId) Given a user id, this method returns the username of the user.
Parameters	userId - A String object representing the id of the user.
Returns	A String object containing the username

Function	String **getCsaReportingUserId**() CSA Reporting User a pre-seeded user who has READ_ONLY permissions for ALL artifacts.
Parameters	None
Returns	A String object representing the id of the CSA Reporting User

Function	String **getOrgName**(String orgnizationId) Given an organization ID, this method returns the organization name.
Parameters	orgId - A String object representing an organization ID.
Returns	A String object representing an organization name.

Function	String[] **getResourceProviderIds**() Returns the ALL resource provider **IDs**.
Parameters	None
Returns	A String array containing all the IDs of the resource providers in the system.

Function	String[] **getResourceProviderIds**(String providerTypeName) Returns the resource provider IDs by provider type.
Parameters	providerTypeName - a String object representing the name of the provider type, such as, VMWARE_VCENTER, HP_SITESCOPE, HP_UCMDB, HP_SA, etc.
Returns	A String array containing the IDs of resource providers of the given provider type.

Function	String[] **getProcessEngineNames**(String processEngineTypeName) Returns the process engine names by process engine type.
Parameters	processEngineTypeName - a String object representing the name of the given process engine type, such as. INTERNAL, HP_OO, HP_CDA.
Returns	A String array containing the names of process engines of the given process engine type.

JavaScript Helper API

Input/output functions

- Function **print**: prints all arguments to response body.
- Function **println**: prints all arguments to response body followed by a new line.
- Variable **request**: contains request arguments (from HTTP POST body) in the form of a JavaScript object with string keys and string values.

Debugging functions

- Function **debug**: print all arguments to debug output (an XML-comment section preceding normal output in response body) in a way that may be helpful for debugging. If invoked without arguments all variables in scope are printed. To view the debug output, send a POST request on the CSA propertysources API, while authenticated as csaReportingUser.

```
POST <csa_endpoint>/csa/propertysources/<js_file_name>
```

- Function **trace_enable**: enable script tracing with output to script debug output. Can be turned on with trace_enable and off with trace_disable dynamically for a selected part of code.
- Function **trace_disable**: disable script tracing.

Script loading function

Function **load**: will load other JS scripts from the same directory. All changes to the global variables (function definitions, global variables) will be visible in the calling script environment. The load function accepts more scripts in variable arguments.

```
load('other_script.js');
```

Limited access to allowed Java classes

Object **Java**: contains the function to access Java types. Only safe classes in the whitelist are allowed.

```
var map = new Java.type('java.util.HashMap');
```

HTTP/S client

HTTP/S client is accessible through the object **HTTPClient** which provides a single function named "call". This function has a single argument which is the JS object that contains the configuration for the HTTP/S client. The JS object configuration includes fields specifying the URL, HTTP method, resource alias to refer to passwords stored in `resource-alias.xml`, request headers, and POST body.

Result value is the response that contains these fields:

- "body" (string with response body)
- "error" (undefined if request was performed successfully)
- "statusCode"
- "status" (status message)
- "headers"

```
var request = {
    url: 'http://httpbin.org/basic-auth/user/passwd',
    method: 'GET',
    resourceAlias: new ResourceAlias('basic auth'),
    params: {
        my: 'param',
        another: 'param'
    },
    headers: {
        'Content-Type': 'application/json',
        'My-Header': 'someValue'
    },
    data: 'some body message'
};
var response = HTTPClient.call(request);

var responseBody;
if (response.error) {
    responseBody = "error: "+response.error;
} else {
    responseBody = "ok: "+response.body;
}
```

Support for Advanced Authentication

There are cases when basic authentication is not enough and the credentials need to be posted as an API endpoint in the request body. To avoid confidential values in text files, CSA supports a set of placeholders that can be used inside of the message body and nowhere else. The placeholders are then replaced with the actual credentials just before the message is sent to the API endpoint. The values are retrieved from a configured resource alias being resource provider defined in CSA in this case.

Using the request property `autoHttpBasic` set to false, instructs the server to find the password to the resource provider using its ID to replace the placeholder `${PASSWORD}` in `dataTemplate,` which is then used to create a body of the request.

To use this approach the request should contain the following fields except the standard ones:

- url - contains just a relative part to that specified as access point url inside the resource provider.
- method – must be POST or PUT to get use of the Request body.
- ResourceAlias – must use statement `CSAIntegrationHelper.getAccessPointForResourceProvider()`.
- templateVariables – should contain any variables not directly returned by calling `CSAIntegrationHelper.getAccessPointDetailsForResourceProvider()`. For processing these values from the details of the resource provider, write your own methods in this JavaScript.
- templateFormat – specify the format of `dataTemplate` ('JSON' – default, 'XML').
- autoHttpBasic – must be set to false.
- dataTemplate – specify body of the request in Json or XML using the format documented by the resource provider using a placeholder for the password (`${PASSWORD}`) and for other variables specified in the `templateVariables` field.

To use this approach for some resource providers, it must be supported by the provider and the provider must document the format of the body.

Below is an example for a One View resource provider.

```
var request = {
url: '/rest/login-sessions',
method: 'POST',
resourceAlias: CSAIntegrationHelper.getAccessPointForResourceProvider(request.oneViewId),
params: {
my: 'param',
another: 'param'
},
headers: {
'Content-Type': 'application/json',
'X-API-Version': "120",
'Accept': 'application/json'
},
templateVariables: {
myUserName: details.username,
myDomain: myGetDomain(details.username)
},
templateFormat: 'JSON',
dataTemplate: '{"authLoginDomain":"LOCAL","password":"${PASSWORD}","userName":"${myUserName}"}',
autoHttpBasic: false
};
```

URI handling

Object **URI**: represents a Uniform Resource Identifier (URI) reference. It is a delegate for Java URI class with all methods available except for toURL. For more information on URI, see the following link:

http://docs.oracle.com/javase/8/docs/api/java/net/URI.html

XML parsing and querying

For XML parsing and querying there is a DOM parser with a DOM-like (read-only) interface, which also provides XPath queries.

Object **DOMParser**: to be instantiated to parse XML. Optional single boolean argument specifies whether parser is namespace-aware. An instance of it has the method `parseFromString` to parse XML in a string. Return value is the DOMNode type. Parser is configured to use secure processing and avoid external entity attacks.

```
new DOMParser().parseFromString('<test></test>');
```

Object **DOMNode**: provides access to parsed XML Document Object Model (DOM) Node field using fields: `attributes`, `childNodes`, `firstChild`, `hasAttributes`, `hasChildNodes`, `lastChild`, `localName`, `namespaceURI`, `nextSibling`, `nodeName`, `nodeValue`, `parentNode`, `prefix`, `previousSibling`, `textContent`.

Or same access using functions: `getAttributes`, `getChildNodes`, `getFirstChild`, `getLastChild`, `getLocalName`, `getNamespaceURI`, `getNextSibling`, `getNodeName`, `getNodeValue`, `getParentNode`, `getPrefix`, `getPreviousSibling`, `getTextContent`.

Additionally, XPath 1.0 query evaluation on DOMNode objects is possible using functions: `evaluate`, `evaluateBoolean`, `evaluateBooleanSet`, `evaluateNode`, `evaluateNodeSet`, `evaluateNumber`, `evaluateNumberSet`, `evaluateString`, evaluateStringSet. Functions ending in 'Set' return an array containing elements. Functions with type in the name return the value of such type. Function evaluate accepts in the second argument one of the constants from **XPathConstants** and returns **XPathResult** object.

```
var d = new DOMParser().parseFromString('<a foo="bar">hmm</a>');
[ d.evaluateNode('/a/@foo').nodeValue,
  d.evaluateString('/a') ]
// returns value [ "5", "hmm" ]
```

```
var d = new DOMParser().parseFromString('<a foo="5">hmm</a>');
[ d.evaluate('/a/@foo', XPathConstants.STRING).stringValue(),
  d.evaluate('/a/@foo', XPathConstants.NUMBER).numberValue(),
  d.evaluate('/a/@foo', XPathConstants.NODE).singleNodeValue().nodeValue ]
// returns value [ "5", 5.0, "5" ]
```

```
var d = new DOMParser().parseFromString('<a><b/><c/></a>');
[ d.firstChild.nodeName,
  d.firstChild.firstChild.nodeName,
  d.firstChild.lastChild.nodeName,
  d.firstChild.firstChild.parentNode.nodeName ]
// returns value [ "a", "b", "c", "a" ]
```

- Function **XPathConstants** contains QName constants: `DOM_OBJECT_MODEL`, `NUMBER`, `NODESET`, `NODE`, `STRING`, `BOOLEAN`.
- Object **XPathResult** (returned by evaluate function on DOMNode) contains functions: `booleanValue`, `iterateNext`, `numberValue`, `resultType`, `singleNodeValue`, `snapshotItem`, `snapshotLength`, `stringValue`.
- Object **XMLSerializer** allows to serialize DOM to XML. It contains functions: `isIndent`, `serializeToString`, `setIndent` and field indent.

```
var d = new DOMParser().parseFromString('<a><b/><c/></a>');
var x = new XMLSerializer();
x.setIndent(false);

x.serializeToString(d);
// returns a string "<?xml version=\"1.0\" encoding=\"UTF-8\"?><a><b/><c/></a>"
```

Note: Mozilla Rhino JavaScript E4X extension is not available.

File reading and writing

File access is allowed in predefined locations that are configured in the `resource-alias.xml` file. Entries in this configuration file define "file storage" resources. Each resource is mapped to a directory in the filesystem. This directory and its subdirectories are then accessible using the `FileStorage` object in JavaScript.

The following example defines two file storages, one read-only and one writable:

```
<entry name='readonly storage'>
        <requiredProtocol>file</requiredProtocol>
        <baseURL>file:///opt/hp/csa46/jboss-
as/standalone/deployments/csa.war/propertysources/fsreadonly/</baseURL>
</entry>
<entry name='writable storage'>
        <requiredProtocol>file</requiredProtocol>
        <baseURL>file:///opt/hp/csa46/jboss-
as/standalone/deployments/csa.war/propertysources/fswritable/</baseURL>
        <writable>true</writable>
</entry>
```

Object **FileStorage**: provides access to files in the storage. Use `ResourceAlias` with the name of the storage to create the `FileStorage`:

```
var storage = new FileStorage(new ResourceAlias('writable storage'));
```

Functions for reading files are: `readAllBytes`, `readAllLines`, `readAll`, `newInputStream`. All read functions take the name of the file as the argument.

Functions for writing to files are: `writeAllBytes`, `writeAllLines`, `writeAll`, `newOutputStream`. Functions that write all at once take the name of the file as the first argument, and the data to write as the second argument. Function `newOutputStream` takes the name of the file as the only argument.

All write functions optionally take one or more **OpenOption** arguments. Usable OpenOptions are:

- APPEND - Bytes will be written to the end of the file rather than the beginning.
- TRUNCATE_EXISTING - If the file already exists, its length is truncated to 0.
- CREATE - Create a new file if it does not exist.
- CREATE_NEW - Create a new file, failing if the file already exists.
- DELETE_ON_CLOSE - Delete on close.

If no options are present, then the write functions work as if CREATE and TRUNCATE_EXISTING are present.

If the file 'myfile.txt' contains text 'hello', then the following examples show possible ways to read the whole file:

```
var bytes = storage.readAllBytes('myfile.txt'); // returns [ 104, 101, 108, 108,
111 ]
var lines = storage.readAllLines('myfile.txt'); // returns [ 'hello' ]
var content = storage.readAll('myfile.txt'); // returns 'hello'
```

The following examples write the whole data (which is the text 'hello') to the file:

```
storage.writeAllBytes('myfile.txt', [ 104, 101, 108, 108, 111 ]);
storage.writeAllLines('myfile.txt', [ 'hello' ]); // writes also new line
character after each line in the array
storage.writeAll('myfile.txt', 'hello', OpenOption.APPEND); // appends 'hello' to
the file
```

The `newInputStream` and `newOutputStream` functions return objects that are in fact Java objects of **BufferedInputStream** and **BufferedOutputStream** types respectively. For a list of methods, see the following documentation:

http://docs.oracle.com/javase/8/docs/api/java/io/BufferedInputStream.html#method.summary

http://docs.oracle.com/javase/8/docs/api/java/io/BufferedOutputStream.html#method.summary

The following example reads the file and writes its copy:

```
var inStream = storage.newInputStream('myfile.txt');
var outStream = storage.newOutputStream('copy-myfile.txt');
while((data = inStream.read()) != -1) {
    outStream.write(data);
}
inStream.close();
outStream.close();
```

The function for deleting the file is called delete and it takes the name of the file as the argument:

```
storage.delete('myfile.txt');
```

SQL client

SQL client is accessible through the object **SQLClient** which provides a single function named "call". Initialization of SQL client has one parameter. This parameter represents a ResourceAlias instance. This alias is located in the `resource-alias.xml` file in the same directory as the dynamic property JavaScript file. This alias instance must be a JDBC type and should contain a user name (mandatory), base url (mandatory), and password (optional).

Resource Alias example:

```
<entry name='postgres database'>
        <username>csa</username>
        <requiredProtocol>jdbc</requiredProtocol>
        <baseURL>jdbc:postgresql://localhost:5432/csa</baseURL>
        <password>ENC(3oKr9eADA7bE53Zk2t9wIA==)</password>
</entry>
```

SQL initialization example:

```
var client = new SQLClient (new ResourceAlias(' postgres database'));
```

The 'call' function can execute only select query. If the query parameter contains insert or update query, the method throws an error. The function has two parameters. The first parameter is an SQL query in JDBC PreparedStatement format (values of query parameters should be replaced by question marks). The second parameter is an array of parameter values in the order that should be placed to the query. These values will be placed into the query before execution. If you need to use Date, Time or Timestamp objects, you must use the SQLDate, SQLTime or SQLTimestamp object instead.

Query execution example:

```
var response = client.call('SELECT * FROM csa_action where artifact_id = ? and consumer_visible =
? and version_number = ? and updated_on > ?',
[ '8fe9ce204ffeb8b4014ffebc521d09f6', true, 1, new SQLTimestamp('2015-09-24 11:00:00.000') ]);
```

Executing query returns response object. This object contains objects metadata, rows (number of returned rows) and data. Metadata is a HashMap object containing the column name as key and column type as value. Rows contain returned rows count. Data is an array with the size of the return rows count and contains HashMap objects. Every map contains the column name as key and column value as value.

Call method response format:

```
{
    metadata: {
        column_name: column_type,
        column_name: column_type
    },
    rows: result_rows_count,
    data: [ {
        column_name: column_value,
        column_name: column_value,
        column_name: column_value
    }, {
        column_name: column_value,
        column_name: column_value,
        column_name: column_value
    } ]
}
```

This object can be processed by any JavaScript code and JSON object representing available values can be built from it.

Build available values from response example:

```
var availableValues = [];
for(var row in response.data) {
    var availableRow = {
        'value': response.data[row].get('uuid'),
        'displayName': response.data[row].get('display_name'),
        'description': response.data[row].get('description'),
        'initialPrice': response.data[row].get('initial_cost'),
        'recurringPrice': response.data[row].get('monthly_cont')
    };
    availableValues.push(availableRow);
}
```

LDAP client

LDAP client is accessible through object `LDAPClient` which provides a single function named "call". Initialization of the LDAP client is without any parameter. The `ResourceAlias` instance is passed by parameter in request. This alias is located in the `resource-alias.xml` file in the same directory as the dynamic property JavaScript file. This alias instance must be an LDPA or LDPAS type. Any other types are not allowed. It can contain the user name (optional), base url (mandatory), and password (optional).

Resource Alias example:

```
    <entry name="LDAP on hpeswlab">
        <username></username>
        <requiredProtocol>ldap</requiredProtocol>
        <baseURL>ldap://ldap-server:10389</baseURL>
        <password></password>
    </entry>
```

LDAP initialization example:

```
var client = new LDAPClient().
```

The 'call' function executes request in JSON. This JSON contains information of various parameters. If some parameters are not set, default values are used. The following table shows default values and constraints of the parameters.

Paremeter	Default value	Constrains
protocol	ldap	Only ldap or ldaps are allowed.
securityAuthentication	none	
resourceAlias		Example: new ResourceAlias('Resource Name')
readTimeout	60000 ms	
connectTimeout	60000 ms	
params.userSearchBase		Example: dc=example,dc=com
params.userSearchFilter		Example: (objectclass=person)
searchControls.scope	SUBTREE_SCOPE	OBJECT_SCOPE, ONELEVEL_SCOPE, SUBTREE_SCOPE
searchControls.timeLimit		number
searchControls.countLimit		number

Authentication Mechanisms (parameter **securityAuthentication**)

Different versions of the LDAP support different types of authentication. Here are some examples:

Value	Description
none	Use no authentication (anonymous)
simple	Use weak authentication (clear-text password)

Request (JSON) execution example:

```
var request = {
    protocol: 'ldap',
    securityAuthentication: 'none',
    resourceAlias: new ResourceAlias('Resource Name'),
    params: {
       userSearchBase: 'dc=example,dc=com',
       userSearchFilter: '(objectclass=person)'
    },
    searchControls: {
        scope: 'SUBTREE_SCOPE',
        timeLimit: 1000
    }
};
var response = client.call(request);
```

Executing request JSON returns a response object that is based on an entry from LDAP. An entry from LDAP consists of a set of attributes. An attribute has a name (an attribute type or attribute description) and one or more values. These attributes are transformed to `HashMap`, where key is attribute's name and value is attribute's value. When attribute has more values, they are separated by comma in one String.

Call method response format:

```
{
     givenname=givenname: consumer,
     sn=sn: consumerA,
     userpassword=userpassword: password,
     ou=ou: Users,
     manager=manager: uid=manager,ou=ConsumerUsers,ou=CSAUsers,dc=example,dc=com,
     member=member:cn=ServiceConsumer,ou=ConsumerGroup,ou=CSAGroups,dc=example,dc=com,
     mail=mail: name.surname@mail.com,
     uid=uid: User28,
     objectclass=objectClass: top, inetOrgPerson, person, organizationalPerson,
uidObject, extensibleObject,
     cn=cn: User28
}
```

This object can be processed by any JavaScript code and JSON object representing available values can be built from it.

Build available values from response example:

```
var response = client.call(request);

var availableValues = [
    {
        'displayName': 'UID',
        'value': response.data[0].get('uid')
    }, {
        'displayName': 'Given name',
        'value': response.data[0].get('givenname')
    }, {
        'displayName': 'Surname',
        'value': response.data[0].get('sn')
    }
];
```

Appendix A – Developing the JavaScript dynamic scripts

JavaScript for dynamic properties is stored in the database. This is a change compared to pre 4.70 versions of CSA. Having the script stored in a database makes it available on all nodes in a cluster environment without the need of additional synchronization (such as using a shared filesystem for all nodes).

The JavaScript files existing in previous versions of CSA are backed up in a ZIP archive file `CSA_HOME/jboss-as/standalone/deployments/csa.war/propertysources/js-backup<timestamp>.zip`. You can unpack the files and view them for reference but keep in mind that JavaScript files on a filesystem no longer have any effect on dynamic properties in CSA. All relevant JavaScript files are kept in the database. This is different from JSP files (see Appendix D – CSA JSP scripts reference).

If you are a Service Designer, you can upload new scripts using a new CSA script management UI (see Dynamic loading) or you can import scripts as part of a service design. You cannot overwrite existing scripts though. The only way to do it is to delete the old version and upload the new version of the script.

To test a script, create a design (can be topology or sequence), go to the subscriber options tab, create an option set, add an option to it, and add a property to it. Set the property to be a list property and click **switch to dynamic entry**.

You can select the script name, add parameters and press the "refresh" icon to run the script and see returned dynamic list property values. A limitation of this is that some dynamic properties may be tied to an identity of the user that invoked this action (for example: dynamic property with list of images on OpenStack may return a different list for different users). To avoid this limitation you would need to publish the design and offering and use the property from the Marketplace Portal under the identity of the intended user.

Another option to test the script is to use any HTTP client. Set authentication user to csaReportingUser (using HTTP Basic authentication) and its password. Do POST request to `https://hostname:8444/csa/propertysources/js-example1.js` (or respective JS script). For JS files this has the advantage of showing debug output from scripts that are not visible in the design UI (unless the script fails). Debug output (if present) is at the start of the response enclosed in the XML-style comment section. Possible errors are indicated with the `<error>` tag.

Scripts can write to the debug output using the **debug** function. This function outputs all arguments in some detailed form which describes the value and also methods that are available to invoke on each object. When invoked without arguments: debug() it outputs information about variables in the current scope (includes all functions and objects).

Another helpful tool is tracing. Tracing can be dynamically enabled and disabled by invoking functions **trace_enable** and **trace_disable**. Tracing writes details on each function that is entered including values of all arguments, each functions exited (either return value or exception is displayed) and variable states on each line.

Using functions that output to this debug output is recommended only to develop scripts. It may slow down scripts significantly and produces output that actual dynamic list properties ignore.

Appendix B – CSA JavaScript reference

CSA uses Mozilla Rhino as JavaScript implementation. The JavaScript engine is configured to run with JavaScript version 1.8. Integration between Java and JavaScript (also known as LiveConnect) is strongly restricted to maintain security and integrity of CSA. To implement common use-cases where JavaScript has to interact with outside systems, additional objects and functions are provided.

Basic JavaScript data types

String

Strings constants are created surrounding string data by a pair of single or double quotes. Character set of JavaScript source code, input and output is assumed to be unicode. There is no separate character type. By addressing character in a string (using []), the return value is a string with one character. Byte and unicode escape sequences are available. Examples:

- 'string 1'
- "string 2"
- '\u2665\x41'

Number

Numbers are implemented as double-precision binary 64-bit IEEE 754 floating point number. Numbers have one bit sign, 11 bits exponent and 52 bits of mantissa. This type can precisely represent all 32bit integer numbers. For non integer numbers or very large integer numbers the number will be "rounded" to nearest representable number.

Boolean

Boolean constants are true and false. They are useful in conditions. Usual operators for boolean values are available.

Regular expression

JavaScript 1.8 has regular expression as a native data type. Regular expression is written between pair of forward slash characters. Regular expression can be used to find patterns in strings and to access groups for easy text extraction. Examples:

- 'testing'.search(/st/) evaluates to 2
- 'axbxc'.split(/x/) evaluates to array ['a','b','c']

Array

An array can hold objects and allows fast access to them through their index. Arrays in JavaScript have variable size and do not restrict object type. Example: [1, 4, 5.5, 'asdf'] Type-restricting arrays are also available.

Object

Objects can be created by enclosing field descriptions inside curly braces. Object system of JavaScript is based on prototypes. Object prototype can be accessed through 'prototype' field. Objects can be also used as maps associating keys to values. To create object based on a prototype, use new keyword.

For more details on JavaScript language you can refer to various books on this topic. Be aware that JavaScript environments even for same version of JavaScript differ by amount of supported built-in functions and objects. For example in browsers it is common to have global variable window which provides a lot of functionality related to browser window interaction, however this one is not available in CSA.

List of global names

The following names are bound to values in the global JavaScript environment:

```
Function, Object, Error, CallSite, decodeURI, decodeURIComponent, encodeURI, encodeURIComponent,
escape, eval, isFinite, isNaN, isXMLName, parseFloat, parseInt, unescape, uneval, NaN, Infinity,
undefined, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError, InternalError,
JavaException, Array, String, Boolean, Number, Date, Math, JSON, With, Call, Script, Iterator,
StopIteration, RegExp, Continuation, ArrayBuffer, Int8Array, Uint8Array, Uint8ClampedArray,
Int16Array, Uint16Array, Int32Array, Uint32Array, Float32Array, Float64Array, DataView, print,
println, debug, trace_enable, trace_disable, request, load, Java CSAIntegrationHelper,
ResourceAlias, DOMParser, XMLSerializer, XPathConstants, URI, FileStorage, OpenOption, HTTPClient,
SQLClient, SQLDate, SQLTimestamp, SQLTime
```

List of functions available with non-CSA specific data types

Functions available in String objects

```
constructor, toString, toSource, valueOf, charAt, charCodeAt, indexOf, lastIndexOf, split,
substring, toLowerCase, toUpperCase, substr, concat, slice, bold, italics, fixed, strike, small,
big, blink, sup, sub, fontsize, fontcolor, link, anchor, equals, equalsIgnoreCase, match, search,
replace, localeCompare, toLocaleLowerCase, toLocaleUpperCase, trim, trimLeft, trimRight, length
```

Functions available in Boolean objects

```
constructor, toString, toSource, valueOf
```

Functions available in Number objects

```
constructor, toString, toLocaleString, toSource, valueOf, toFixed, toExponential, toPrecision
```

Functions available in RegExp objects

```
constructor, compile, toString, toSource, exec, test, prefix, multiline, ignoreCase, global,
source, lastIndex
```

Functions available in Date objects

```
constructor, toString, toTimeString, toDateString, toLocaleString, toLocaleTimeString,
toLocaleDateString, toUTCString, toSource, valueOf, getTime, getYear, getFullYear, getUTCFullYear,
getMonth, getUTCMonth, getDate, getUTCDate, getDay, getUTCDay, getHours, getUTCHours, getMinutes,
getUTCMinutes, getSeconds, getUTCSeconds, getMilliseconds, getUTCMilliseconds, getTimezoneOffset,
setTime, setMilliseconds, setUTCMilliseconds, setSeconds, setUTCSeconds, setMinutes, setUTCMinutes,
setHours, setUTCHours, setDate, setUTCDate, setMonth, setUTCMonth, setFullYear, setUTCFullYear,
setYear, toISOString, toJSON
```

Functions available in Array objects

```
constructor, toString, toLocaleString, toSource, join, reverse, sort, push, pop, shift, unshift,
splice, concat, slice, indexOf, lastIndexOf, every, filter, forEach, map, some, find, findIndex,
reduce, reduceRight, length
```

Indexing access is available.

Functions and values available in Math object

```
toSource, abs, acos, asin, atan, atan2, ceil, cos, exp, floor, log, max, min, pow, random, round,
sin, sqrt, tan, E, PI, LN10, LN2, LOG2E, LOG10E, SQRT1_2, SQRT2
```

Functions avilable in ArrayBuffer objects

```
byteLength, slice
```

Indexing access is available.

Appendix C – JavaScript sample code

Sample list

```
println('<Property>');
println('    <availableValues>');
println('        <value>Standard</value>');
println('        <displayName>Standard</displayName>');
println('        <description>Standard</description>');
println('        <initialPrice>12</initialPrice>');
println('        <recurringPrice>2</recurringPrice>');
println('    </availableValues>');
println('    <availableValues>');
println('        <value>Enterprise</value>');
println('        <displayName>Enterprise</displayName>');
println('        <description>Enterprise</description>');
println('        <initialPrice>23</initialPrice>');
println('        <recurringPrice>3</recurringPrice>');
println('    </availableValues>');
println('</Property>');
```

Sample list with availableValues array

```
availableValues = [ {
    'value': 'Standard',
    'displayName': 'Standard',
    'description': 'Standard',
    'initialPrice': '12',
    'recurringPrice': '2'
}, {
    'value': 'Enterprise',
    'displayName': 'Enterprise',
    'description': 'Enterprise',
    'initialPrice': '23',
    'recurringPrice': '3'
} ];
```

Load content from a text file

Create the file storage entry in the `resource-alias.xml` file as shown here:

```
<resource-alias>
    <entry name="csa files”>
        <requiredProtocol>file</requiredProtocol>
        <baseURL>file:///opt/csa46/files</baseURL>
    </entry>
</resource-alias>
```

Then use this resource alias in the script:

```
var storage = new FileStorage(new ResourceAlias('csa files'));

// Read the whole file to an array
var lines = storage.readAllLines('options.txt');

var availableValues = [];

// Loop through the lines
for(var l = 0; l < lines.length; l++) {
    var line = lines[l];
    availableValues.push({
        'displayName': l+': '+line,
        'value': line,
        'description': line
    })
}
```

Load content from the database using SQL client

Database driver jars should be placed under: `CSA_HOME/jboss-as/standalone/deployments/csa.war/WEB-INF/lib/.`

Create the database connection entry in the `resource-alias.xml` as shown here:

```
<resource-alias>
    <entry name='sqlserver auth'>
        <username>csauser</username>
        <requiredProtocol>jdbc</requiredProtocol>
        <baseURL>jdbc:jtds:sqlserver://localhost:1433/csa</baseURL>
        <password>secret</password>
    </entry>
</resource-alias>
```

Note: The format of the URL varies greatly between drivers, check the documentation for the relevant driver you are using, such as MSSQL, Oracle, PostgreSQL:

MSSQL = `jdbc:jtds:sqlserver://<ipaddress>:1433/<dbname>`

Oracle = `jdbc:oracle:thin:@<ipaddress>:1521:<SID>`

PostgreSQL = `jdbc:postgresql://<ipaddress>:5432/<dbname>`

Then use this resource alias in the script:

```
// Get a client for the database
var client = new SQLClient(new ResourceAlias('sqlserver auth'));

// Execute the query
var response = client.call(
    'SELECT uuid, display_name, discriminator FROM csa_category where
discriminator = ?',
    [request.discriminator != null? request.discriminator: 'CATALOG_CATEGORY']);

var availableValues = [];

// Loop through the result set
for(var row in response.data) {
    var availableRow = {
        'value': response.data[row].get('uuid'),
        'displayName': response.data[row].get('display_name'),
        'description': response.data[row].get('discriminator')
    };
    availableValues.push(availableRow);
}
```

Load data from an HTTP resource

The simplest GET call:

```
// Make GET request
var response = HTTPClient.call({
    method: 'GET',
    url: 'http://httpbin.org/xml'
});
debug(response);

var availableValues = [];

if(response.statusCode == 200) {
    // Show the first 100 characters of the response
    var line = response.body.substring(0, 99);
    availableValues.push({
        value: line,
        displayName: line
    });
}
```

To authenticate an HTTP request with Basic Authentication, create resource alias entry:

```
<resource-alias>
    <entry name='basic auth'>
        <username>user</username>
        <requiredProtocol>http</requiredProtocol>
        <requiredURL>http://httpbin.org/basic-auth/user/passwd</requiredURL>
        <password>passwd</password>
    </entry>
</resource-alias>
```

Then use it within the request:

```
// Make GET request with basic authentication
var request = {
    url: 'http://httpbin.org/basic-auth/user/passwd',
    method: 'GET',
    resourceAlias: new ResourceAlias('basic auth')
};
var response = HTTPClient.call(request);
debug(response);

var availableValues = [];

if(response.statusCode == 200) {
    // Show the first 100 characters of the response
    var line = response.body.substring(0, 99);
    availableValues.push({
        value: line,
        displayName: line
    });
}
```

If HTTP response is returning XML, the following parsing logic can be applied:

```
var response = HTTPClient.call({
    method: 'GET',
    url: 'http://httpbin.org/xml'
});
var availableValues = [];

function value(name, value) {
    return {
        value: value,
        displayName: name,
        description: value
    }
}
if(response.statusCode == 200) {
    // Parse the body to DOMNode
    var xml = new DOMParser().parseFromString(response.body);
    // Show some nodes
    availableValues.push(value(xml.firstChild.nodeName,
xml.firstChild.nodeValue));
    var sibling = xml.firstChild.nextSibling;
    availableValues.push(value(sibling.nodeName, sibling.hasChildNodes()));
    // Show some attributes
    if(sibling.hasAttributes()) {
        var attrs=sibling.attributes;
        for(var a in attrs) {
            availableValues.push(value(a, attrs[a]));
        }
    }
    // Get all <title> elements with XPath
    var titles = xml.evaluateStringSet('//title');
    for(var t in titles) {
        availableValues.push(value('title '+t, titles[t]));
    }
}
```

Example of `CSAIntegrationHelper`:

```
var username =
CSAIntegrationHelper.getUsername('90d96588360da0c701360da0f1d600a1');
availableValues = [{
    value: username,
    displayName: username
}];
```

Load content from Operations Orchestration (OO)

Get the Operations Orchestration certificate and import it to your trust store:

```
keytool -importcert -keystore js-example.truststore -storepass secret -file OO.cert -alias oo
```

Create the resource alias to authenticate against the Operations Orchestration certificate, use the trust store with the certificate:

```
<resource-alias>
    <entry name='oo auth'>
        <username>admin</username>
        <requiredProtocol>https</requiredProtocol>

<requiredURLPrefix>https://oohost:8445/PAS/services/rest/run/</requiredURLPrefix>
        <password>admin</password>
        <trustStore>
            <fileName>/full/path/to/js-example.truststore</fileName>
            <storePassword>secret</storePassword>
        </trustStore>
    </entry>
</resource-alias>
```

Pass the flow name as an input parameter to the script like this:

```
ooFlowURL=https://oohost:8445/PAS/services/rest/run/Library/CSA/Topology_Generated_Flows/04.50.00
00/design/1.0.0/End-to-End%2520Deployment%2520%282eb4d182-57a2-40d1-8660-1c6ff391bb3b%29
```

The XML that is read from Operations Orchestration contains a text in one of the `<item>` elements that looks like this:

```
{Result=;vCenterServer0001-ipAddressList=null;vCenterServer0001-hostname=null;vCenterServer0001-
cpuCount=null;vCenterServer0001-ipAddress=null;vCenterServer0001-vmID=null;vCenterServer0001-
macAddress=null;vCenterServer0001-result=Provisioning of the VM failed.;vCenterServer0001-
memorySize=null;vCenterServer0001-vmState=null;}
```

The script reads these name/value pairs and puts them to `availableValues`:

```
var response = HTTPClient.call({
    method: 'GET',
    url: request.ooFlowURL,
    resourceAlias: new ResourceAlias('oo auth')
});
var availableValues = [];

if(response.statusCode == 200) {
    // Parse the XML
    var respXml = new DOMParser().parseFromString(response.body);
    // Find the flowResult item
    var flowResult = respXml.evaluateString('//item[name="flowResult"]/value');
    // Extract name=value pairs
    var pattern = /{?([^=]+)=([^;]*);/g;
    var array;
    while((array = pattern.exec(flowResult)) != null) {
        availableValues.push({
            'displayName': array[1],
            'value': array[2],
            'description': array[0]
        });
    }
} else {
    availableValues.push({
        'displayName': response.statusCode+' '+response.status,
        'value': response.body
    });
}
```

Appendix D – CSA JSP scripts reference

There are two ways to implement dynamic properties in the option model: JSP files or JavaScript files to compute values of the dynamic properties. JSP files were used prior to 4.70, but have been deprecated because of security and technical issues. JavaScript is the new and preferred way to implement dynamic properties.

Deprecating the JSP files in 4.70, means that no enhancements will be made in the product regarding managing JSPs. The existing content will still be supported and JSPs will continue to work. However, it is highly recommended that you only use JavaScript to build new content.

The JSP files are stored in the CSA database, but they cannot be executed from the database. Because of that, CSA copies the JSP files from the database to the particular node filesystem from which they are being executed. It is possible to replace a JSP with a custom version on the node filesystem. Such a custom JSP is not propagated back to the database however.

JSPs in a cluster HA system

In a cluster HA system, the JSP files are being synchronized from the database to the filesystem on all nodes. Customizing a JSP file on a filesystem will override the JSP for one node but the change is not propagated to the other nodes. That can cause a script execution to fail in the cluster. You must copy the customized JSP files onto each node's filesystem in the cluster for the JSP execution to succeed, or use the following workaround.

When a service design is imported, it puts the nested JSPs into the CSA database. This import behavior can be leveraged for JSP development. You can put the JSP on the filesystem of a standalone CSA node. Then use the JSP in a design and export that design. When the design is imported onto any CSA cluster HA system, the design brings the nested JSP, which is put into the database, and shares the design (with the JSP) among all the nodes.

JSP registration in the database

- **import** - when the service design/offering/catalogue is exported, it also contains assigned JSPs. When this archive is imported to another CSA service design/offering/catalogue, the JSP files are stored in the database.
- **capsule import** - the capsule can contain auxiliary files (JSP, xml, txt, jar, and so on). All those auxiliary files are put into the database along with the JSPs and are synchronized to the nodes in the same way that the JSPs are synchronized to the nodes.

Send documentation feedback

If you have comments about this document, you can send them to clouddocs@hpe.com.

Legal notices

Warranty

The only warranties for Hewlett Packard Enterprise products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Hewlett Packard Enterprise shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.

Restricted rights legend

Confidential computer software. Valid license from Hewlett Packard Enterprise required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright notice

Trademark notices

Adobe® is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

UNIX® is a registered trademark of The Open Group.

RED HAT READY™ Logo and RED HAT CERTIFIED PARTNER™ Logo are trademarks of Red Hat, Inc.

The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation in the United States and other countries, and are used with the OpenStack Foundation's permission.

Documentation updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for recent updates or to verify that you are using the most recent edition of a document, go to the following URL and sign-in or register: https://softwaresupport.hpe.com.

Select Manuals from the Dashboard menu to view all available documentation. Use the search and filter functions to find documentation, whitepapers, and other information sources.

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your Hewlett Packard Enterprise sales representative for details.

Support

Visit the Hewlett Packard Enterprise Software Support Online web site at https://softwaresupport.hpe.com.