Maximo NextGen REST API

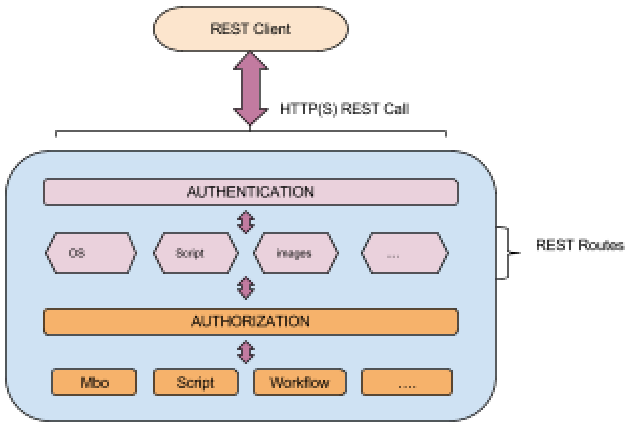
Overview

Maximo NextGen REST APIs are a complete rewrite of the existing REST apis that was released around Maximo version 7.1. Maximo NextGen REST apis are in Maximo releases starting 7602 version. The NextGen apis are often referred to as the REST/JSON apis for the end to end support for JSON data format. There are numerous benefits of using the nextgen apis with a few highlighted here:

- Significantly enhanced support for querying maximo data - subselects, related object queries, multi-attribute text search, custom queries (java/scripting).
- Support for system level actions - bookmarking, notifications, e-sig, image association etc.
- Tight integration to automation scripts - query, actions, custom apis.
- Enhanced REST support for MIF Standard Services - support json data type for those service methods.
- Metadata support using json schema.
- Supports dynamic query views.
- Support for group by queries.
- Supports custom json elements appended to the Object structure JSON.
- Integration with Maximo Cache framework.
- Integration with Maximo formulas.
- Integration with Maximo Federated Mbos/Resources.

This api uses the same code base as the OSLC REST apis (which is used by the Anywhere platform) but is much simpler to setup (no need to setup OSLC resources) and sheds the namespaced json that the OSLC api required. Effectively this api is ready to be used when a vanilla Maximo is installed and setup with users and groups - with no additional setup.

Let's start with a high level overview of the architecture.



As you can see, the REST api call flows through the authentication phase, api routes, authorization and then it interacts the with the Maximo artifacts like Mbo's, Workflows, Automation Scripts etc. API routes are RESTlets (REST handlers) that provide the APIs for interfacing with various Maximo artifacts like - Maximo Business Objects, Automation scripts, Images, Permissions, Schemas etc. This is more for the developers of the API to organically expand the footprint of the REST apis to cover more and more parts of Maximo. We do not see any need for users of the of API to leverage this functionality yet. We expect more routes to get added as we expand the APIs over the next few releases.

Note: It's recommended that you try out the apis, as you walk through this document. You can may want to install the "json viewer" plugin for FF or Chrome which will make easier to view the json documents from the api response. You may want to use a tool like Chrome Postman to make the POST calls with the api.

This document also assumes that you would use the lean json format (json with no namespace). This can be achieved by setting the lean=1 query parameter at login time. Note that it is recommended that you have the lean=1 query parameter set for all requests as it is possible the request might move to another server as part of load balancing or if the original server fails over to a new server. For brevity this query parameter will be ommited in most of the samples show below, but it is recommended that you add that while making the requests.

This document also assumes some familiarity with Maximo Object Structures which form the resources for this REST api. A quick starter for the API demo can be viewed here REST API Quick Start.

Authentication

Below we discuss the most common forms of authentication that we use in Maximo deployments and how to use the REST apis for those authentication schemes.

Maximo native authentication

This is where Maximo owns the user repository along with the user credentials. Maximo is responsible for authenticating the incoming REST call. The REST api expects the HTTP request with a "maxauth" request header that has a base64 encoded userid:password. A sample request would look like below:

```
POST /oslc/login
maxauth: <base64 encoded user:pass>
```

```
<no body required>
```

LDAP based authentications

This is where Maximo does not own the authentication credentials. Its owned by the application server and hence the authentication is validated by the application server.

BASIC

In this scheme the application server expects the authentication credentials to be presented as below

```
POST /oslc/login
Authorization: BASIC <base64 encoded user:pass>

<no body required>
```

FORM

In this scheme the request should look like below

```
POST /j_security_check
Content-type: application/x-www-form-urlencoded

j_username=<userid>&j_password=<password>
```

Note, this being a form encoded POST, the userid and password values needs to be url encoded values. The response for this request will have the jsessionid cookie along with Ltpa token cookies (for Websphere). These cookies need to be re-used for the subsequent api calls.

In fact for all authentication schemes it's recommended that we re-use the authenticated session for subsequent REST api calls by replaying the session and authentication based cookies from a successful authentication response. This helps with performance as the subsequent api calls just re-use the session and does not need to re-authenticate for every request.

Sample JAVA client code for each of these authentication schemes can be referenced from the MaximoConnector.java code (method setAuth(..)) in the Maximo Connector code.

API Home (/oslc)

The api root url is /oslc. A GET call on that would fetch a json object with necessary links to get the details of the current Maximo runtime. Explore the links like systeminfo, whoami, installedProducts, serverMembers, apis along with details like the current date time, language, calendar of the deployed Maximo. Some of the links are described below.

- **systeminfo:** This api provides the system information json (for the deployed Maximo instance. The api is GET /oslc/systeminfo.
- **Whoami:** This api provides the profile json for the logged in user. The api is GET /oslc/whoami.
- **installedProducts:** This api provides the list of installed add-ons for Maximo.The api is GET /oslc/products.
- **serverMembers:** This api is the root api for the **Maximo Management Interface (MMI)**. The response json contains the list of live Maximo servers as per the maximo serversession table. Note that it takes some time for the Maximo runtime to detect a down server and hence it is possible that the response shows some servers that may not be operating at that instant. The api is GET /oslc/members. The resulting json will have links to drill down into individual servers and introspect the various aspects of Maximo runtime like memory, mbo count, Integration caches, threads, database connections etc. This is a pluggable list of data points which we have added over time and will continue to go on adding to help customers figure out Maximo runtime server health.
- **License:** This api provides the list of available license keys for the various Maximo components for this deployment. The api is GET /oslc/license.
- **Apimeta:** This api provides the metadata for all the Object structures that are api eligible (ie usewith value of reporting, integration and oslc). The api is GET /oslc/apimeta.The metadata for each Object structure includes - the schema information as well as the available queries (saved queries). Additionally it provides the creation api url and the (security) authorization application name. We will cover these entities in details later in the document.

REST API Errors

Querying Maximo using the REST API

Let’s start driving into the query aspect of this rest api. A few important aspects were considered while designing this api framework.

- Being able to filter and sort Maximo business objects using a higher level query language which internally will map to the native sql for the corresponding relation db used by Maximo deployment.
- Being able to select the list of attributes that we want the api to fetch.
- Being able to fetch data from related objects leveraging existing Maximo relationships - without needing any additional configuration.
- Being able to page data as needed.

With these basic requirements in mind, we can now talk about some of the query parameters that are used for facilitating these. Query api is always based on a collection uri. All collection uri's (for any resources that are api enabled) support a known set of URI query parameters that help us operate on the collection. The 4 most common ones are listed below. There are many more and we will explore them as we go.

oslc.select: this is used to specify the set of attributes to fetch from the Object structures as well as the related objects.
oslc.where: This is used to specify the where clause which follows the syntax detailed in the next section (Where clause syntax).
oslc.orderBy: this is used to specify the order by clause.
oslc.pageSize: This is used to specify the page size for the

Select Clause

A sample select clause might be help understand it better. Below is a simple select clause from the MXASSET object structure..select=assetnum,location,description,status

```
GET /oslc/os/mxasset?oslc.pageSize=10
```

The response (collection resource) looks like below:

```
{
   “members”:[
    {
     “href”:”uri”
    }
    …
    ]
   “responseInfo”:{
       “nextPage”:{ “href”:”next page uri”},
       “href”:”request uri”,
       “pagenum”:1
    }
}
```

This will result in fetching the 10 members (max) of the MXASSET resource (which is based on the ASSET Mbo in Maximo) - each member pointing to an asset record with a “href” that contains the link to get the details for that MXASSET. Any collection resource in this REST api will follow this same basic structure.

Note that the result is boxed under the **member** json array. Other than the “member” property there is another property called “**responseInfo**” which contains the meta information about the query. The meta information includes the current uri used to get the result (href) as well the url for the next page (**nextPage**) if there is a next page. It will also contain the url for the previous page (**previousPage**) if there is a previous page. This will also contain the current page number (**pagenum**) and total database count (**totalCount**) of the rows that meet the query filter criterion as well as the total number of pages (**totalPages)** available for this query. The **totalCount** and **totalPages** are not displayed by default and will be added only when we add the query parameter/value **“collectioncount=1**” to the request. If you are just interested in getting the total count of records that match the query and not interested in the records itself, you can just simply use the request query parameter **count=1**. This will result in the following json.

```
{
   “totalCount”:<total count of records matching the query>
}
```

There is a system property called mxe.oslc.collectioncount which if set to 1 will always return the **totalCount** and **totalPages** by default as part of the **responseInfo**. However we recommend not to set that property as there will be cases where you may not need those values and will unnecessarily incur the cost of getting those values (which needs an additional sql call to get total count). Its preferred to just request them using the query parameter **collectioncount** as needed.

Just getting the links to the member resources maynot be very exciting or useful. Rather than getting details by traversing individual URI’s, we would leverage the **oslc.select** clause to get more details inlined in this response json.

```
GET
/oslc/os/mxasset?oslc.pageSize=10&oslc.select=assetnum,location,descr
iption,status
```

The resulting json will look like

```
{
   “members”:[
     {
      “href”:”<uri>”,
      “status”:”...”,
      “status_description”:”synonym description of the status”,
              “location”:”..”,
              “assetnum”:”..”,
              “assetmeter_collectionref”:”<uri for assetmeters
collection>”
     }
     …
     ]
   “responseInfo”:{
       “nextPage”:{ “href”:”next page uri”},
       “href”:”request uri”,
       “pagenum”:1
     }
}
```

This will result in a json that contains the 4 attributes as requested for each of the members. Note that by default the api response skips the null value attributes. So for example if location is null for any of the member assets in the selection, that attribute will not appear in the member json. This helps reduce the response payload size. To force the response to add null value attributes, use the query parameter **_dropnulls=0**.

Note that along with the “status” you will also have the “status_description” property which contains the synonymdomain description for that corresponding status value based on the users profile language. The api framework will detect a domain bound attribute (from the Maximo metadata repository) and will use the domain cache to fetch the description for that status.

You will also see the **_rowstamp** property which would be present for every object in the Object Structure for a given resource record. This is used for handling dirty updates. We will cover this in the create/update of resources section.

You will also see the xxxx**_collectionref** properties which contain the links to child objects as defined in the MXASSET Object Structure. The prefix (xxxx) is the name of the child object. As you traverse through those links (ie GET <collectionref link>), you will see the collection of the child objects. We can traverse through that collection resource just like any other collection ie we can page through them (using oslc.pageSize) or filter them (using oslc.where) or get partial views (using oslc.select) etc.

Say if we wanted to get some data from the assetmeter objectwhere assetmeter is a child object as defined in the Object Structure MXASSET. To do this we can leverage the following select clause

```
oslc.select=assetnum,status,description,location,assetmeter{*}
```

The member json will look like

```
{
     “assetnum”:”...”,
     “assetmeter”:[
      {
           ….
      }
      ….
      ]
      ...
}
```

As you can figure out, to get the child object details we are using the notation - <child object name>{comma separated attribute names or * to get all properties}. So assetmeter{*} is going to fetch all properties for the assetmeter. This notation can easily be nested any levels deep. For example you can do obj1{prop1,prop2,obj2{prop21,prop22}} - where obj2 is defined as a child object of obj1.

While this notation works for child objects, you often will need to get more data from a related objects (say locations or workorders) which is not defined in the Object structure. The notation below will help us do this.

oslc.select=assetnum,status,description,location,location.description,location.status

This will result in a member json like.

```
{
     “assetnum”:”...”,
     “location”: {
          “status”:”...”,
          “status_description”:”...”
      }
      "$alias_this_attr$location":”….”,
      ...
}
```

Note the property "**$alias_this_attr$location**". We will cover this in the "Aliasing attributes" section.
Asset to Location is a 1:1 relationship and hence the dot notation format works out great and attaches the json object for the location with the member json for MXASSET at the asset header object. Note that the api framework detected a conflict of names - attribute **location** and the relation named **location.** Note that the dot notation format is <relation name>[.<relation name>]*.<attribute name>. Effectively these relations can be nested too. The api response would bunch attributes at each relation level to form the json object. A sample below might help:

```
oslc.select=rel1.a1,rel1.a2,rel1.rel11.a11,rel1.rel11.a12,rel1.rel21.a21
```

Will result in a json like

```
{
    rel1:{
       a1:..
       a2:...
       rel11:{
          a11:..
          a12:...
      }
      rel21:{
          a21:..
      }
      …..
}
```

As you might have noticed that dot notations produce json objects. But if we need related data that would be 1:*, we would need a slight variation of this. A real example would be Asset:Workorder. An asset may have many open workorders and say we want to get details about all open workorders for my set of Assets. The select clause will look like below

```
oslc.select=assetnum,..,rel.openwo{wonum,description}
```

This "**rel**" notation has the format - "rel.<relation name>". This will result in a json array property named <relation name>. The sample output format would be like below.

```
{
    "openwo":[
     {
         "wonum":..
     }
     ….
    ]
}
```

As like the dot notation, this rel notation can be nested too. The nesting got to happen as part of its attribute set. As usual a sample will be much easier to talk to.

```
oslc.select=rel.rel1{attr1,attr2,rel.rel2{attr21,attr22},rel.rel3{*},
rel4.attr4}
```

Here the rel2, rel3 are samples of nesting the rel notation. The rel4.attr4 showcases the fact that you can embed a dot notation within a rel notation but not the other way round.The rel3 also demonstrates that you can just use * to get all attributes for that target object. Although here it gets tricky as what would * imply - all persistent attributes? Or all persistent and non-persistent attributes combined?. So the rule of thumb we followed is as below:

- If the target object is a persistent object, the * notation will include all persistent attributes for that object. You would need to explicitly request the non-persistent attributes to get them included. For example - rel.openwo{*,displaywonum} where displaywonum is a non-persistent attribute in the target object.
- If the target object is a non-persistent object, the * notation will include all (non-persistent) attributes for that object as it has not persistent attributes anyways.

Note that these dot notation attributes and the rel attributes can be used at any level of the Object structure. For example we count use it in assetmeters like below

```
oslc.select=assetnum,status,assetmeter{*,rel.rel1{attr1,attr2},rel2.a
ttr3}
```

A demo of this can be see here Dynamic Select Clause.

Other forms of related data

So far we have been discussing how to fetch the mbo and related mbo attributes for the object structure. We are now going to cover the other forms to related data and how to request them explicitly or implicitly.

Images

In Maximo we have a image repository (imglib table) that stores the image avatars for the Maximo managed resources (likes Assets, Items, Person etc). The api framework maintains a cache of the image references. While fetching the resource details, if the system detects an image reference, the uri for the image document will be added to the resulting json (_imagelibref). Maximo image repository supports storing the images in Maximo database or in an external repository - provided the repository exposes a simple uri based mechanism to load the images. To facilitate that, the IMGLIB table has 2 attributes - **imguri** and **endpointname**. The endpointname points to the Integration endpoint - which say is the http(s) endpoint and the imguri refers to the url of the image which is used by the http endpoint to fetch the image. It's possible to use a custom endpoint that be leveraged to handle more complex urls. Bulk loading of images can be done using a sql command line tools. Associating images to any Maximo Mbo's can be done using REST apis. The sample REST api below associates an Asset with an image.

```
POST /oslc/os/mxasset/{id}?action=system:addimage
custom-encoding: base64
x-method-override:PATCH
Slug: <maps to image name in imglib>
Content-type: <maps to mime type in imglib>

<HTTP body contains the base64 encoded image bytes>
```

Or

```
POST /oslc/os/mxasset/{id}?action=system:addimage
x-method-override:PATCH
Slug: <maps to image name in imglib>
Content-type: <maps to mime type in imglib>

{
    "imguri":<uri for the externally sourced image>,
    "endpointname":..
}
```

In the same line, we have an api to **delete** the associated image

```
POST /oslc/os/mxasset/{id}?action=system:deleteimage
x-method-override:PATCH
```

Database Aggregation functions

Maximo REST api supports using the database aggregation (max,min,avg,sum,count and exists) functions on related MboSets. For example say we want to apply these functions on the open workorders for an asset. The sample api below will use all the aggregation functions.

```
GET /oslc/os/mxasset/{rest
id}?oslc.select=assetnum,openwo.actlabhrs._dbavg,openwo.actlabhrs._db
sum,openwo.actlabhrs._dbmax,openwo.actlabhrs._dbmin,openwo._dbcount
```

As you can see the format for the sum,avg,max and min are <relation name>.<target attr name>._<operation>. Note that the supported operations are the usual suspects dbsum (for sum), dbavg (for avg), dbmax (for max) and dbmin (for min). Their format is always a dot (.) separated 3 token format which includes a relationship name token followed by an attribute name followed by the underscore prefixed (_) operation to perform on that related attribute.

For the count (operation dbcount) we have a 2 token format which includes the relation name as the first token and the operation name (_dbcount). It will evaluate the count on that related MboSet. The json response would look like below:

```
{
    “assetnum”:..
    “openwo”:{
         “_dbcount”:20,
        “actlabhrs”:{
            “_dbavg”:8,
            “_dbsum”:80
         }
    }
}
```

Bookmarks

Maximo bookmarks can be leveraged with the rest apis.

Cache properties

Formula properties

We can select calculated values without needing to create non persistent attributes. This can be done by leveraging the tight integration between the rest apis and the Object formula feature (introduced in Maximo base 7605 release). An example of the formula properties would help explain it better.

Say we create a object formula called MYREPLACECOST for Asset object (using the Object Formula action from the Database Configuration application). The formula can be defined as purchaseprice/NVL(priority,1). We can then select that formula property associated with the asset object using the api select clause as shown below

```
GET
/oslc/os/mxasset?oslc.select=assetnum,status,exp.myreplacecost,assetm
eter{...}
```

The response would be as if **myreplacecost** was an attribute of the asset mbo.

```
{
     “member”:
      {
          “assetnum”:..
          “myreplacecost”:100
      },
      ….
}
```

Similar approach will work for an individual resource

```
GET /oslc/os/mxasset/{rest
id}?oslc.select=assetnum,status,exp.myreplacecost,assetmeter{...}

{
      “assetnum”:..
      “myreplacecost”:100
}
```

Note that this can be considered as a great alternative to defining non-persistent attributes just for the sake of holding calculated values. This acts like a dynamic attribute which does not need db config or admin mode.

For more information on Maximo Formula feature please visit Maximo Formula.

Federated resource data

Aliasing of attributes

You might have noticed that the name clash of the attribute named “location” with the relation named “location” - both at the Asset object level. In xml world this could have been resolved with namespaces. Here in the json world we want to just rename the property with an alias. Aliasing thus refers to the process of renaming a mbo attribute in the json domain to avoid name conflicts. If there is a name conflict the json response will mark rename attribute with the prefix "**$alias_this_attr$".** To alias this attribute, we can simple use the -- operator in the select clause as shown below

```
oslc.select=assetnum,location--mylocation,location.status
```

This will rename the location attribute to "mylocation" in the json domain. Note that -- operator only works on attributes and not on object names or relation names. So effectively if an attribute name clashes with an object name or a relation name, the attribute name needs to be aliased.

Traversing to related MboSets

Traversing to related MboSets can be done by using the relation name as part of the GET uri call. Sample below shows how we can move from asset to workorder using the relationship name.

```
GET /oslc/os/mxasset/{rest id}/openwo?oslc.select=*
```

As you can see, we are using the openwo relationship name to traverse to the workorder collection from a given asset. Now the resulting json will be a serialized response based on the Workorder mbo whicb by default will not contain any non-persistent attributes. For non-persistent attributes we need to request for them explicitly in the oslc.select clause.

```
GET /oslc/os/mxasset/{rest id}/openwo?oslc.select=*,npattr1,npattr2
```

However in lot of cases, you would prefer getting those response as a Object Structure collection resource. We have a notation for doing just that.

```
GET /oslc/os/mxasset/{rest id}/openwo.mxwodetail?oslc.select=*
```

Note the way we appended the Object Structure name at the end of the relation name with a dot separator. With this request you will get all Workorder records returned as mxwodetail records. Another variation of this api that we support is shown below:

```
GET /oslc/os/mxasset/{rest
id}/openwo?oslc.select=*&responseos=mxwodetail
```

In both cases (ie mbo vs object structure), you can use all the collection api query parameters like oslc.select and oslc.where and paging etc to filter, sort and view the collection as you need. Note that this is recursive ie we can go as deep nested as we want using the relation name and the rest id pair. An example is shown below:

```
GET /oslc/os/mxasset/{rest id}/openwo/{rest
id}/jobplan/?oslc.select=*
```

Note these rest ids are derived from the uris that come back from the server. The client code should not need to generate these ids. It should just leverage the uris and append the relation name token to it to traverse down from the selected record. We have seen in previous sections how related MboSets can be inlined (inside the parent Mbo/set data) using the **rel** notation. This is different in the sense that we are not inlining the related MboSet, rather we are treating it just like another independent collection resource.

Filtering Data Using Where Clause

The most common way to filter a resource set is to use the oslc.where query parameter. This internally maps to the Maximo Qbe framework. We can filter data based on all persistent attributes - at the main mbo (for the os) or any related mbo. For example the where clause below would filter assets based on locations and asset status.

```
GET /oslc/os/mxasset?oslc.where=status="OPERATING" and
location.status="OPERATING"
```

Note here the Locations Mbo is not part of the MXASSET Object Structure. The format for the dot notation (location.status) is <rel1>[.rel2]*.attr1. As you can make out, the dot notation can be deeply nested. The leaf element is always an attribute in the target Mbo.

Before we dive too much into nested queries, let’s just look at the different operators.

Operator	**Description**	**Usage**
=	equals	status=”APPR”
>=	Greater than equals	priority>=1
>	Greater than	startdate>”iso date”
<	Less than	startdate<”iso date”
<=	Less than equals	linecost<=200.5
!=	Not equals	priority!=2
in	In clause	location in [“A”,”B”,”C”], priority in [1,2,3]

This query language is data type sensitive and we would use the double quotes "" for character based attributes and for dates (ISO Format). Numeric values are represented in their corresponding ISO formats (ie non localized format). Boolean values are always represented either as 1/0 or true/false (no quotes)

```
oslc.where=status in ["OPERATING","ACTIVE"] and priority=3 and
statusdate>"ISO date string" and linear=false
```

We can do in clause with numeric values too.

```
oslc.where=priority in [1,2,3]
```

To do **like** clause you can do the following variations.

```
oslc.where=status=”%APPR%”
```

If we want to do starts with:

```
oslc.where=status=”APPR%”
```

If we want to do ends with

```
oslc.where=status=”%APPR”
```

For doing null value queries we can use the star (*) notation as below. For example a not null check can be done using the following format where - **status is not null**.

```
oslc.where=status=”*”
```

If we wanted to do the is null check:

```
oslc.where=status!=”*”
```

We have already shown a sample of the in clause. If we want to do a not in clause - use the format below.

```
location!= “[BR300,BR400]”
```

Note that this not in clause currently only works for ALN attributes.

Range Filters

These are used for supporting range based queries. For example if we want to only get assets in certain date range and priority range we can use this feature.

```
oslc.where=priority>1 and priority<=3 and
installdate>=”1999-02-06T00:00:00-05:00” and
installdate<”2009-02-06T00:00:00-05:00”
```

The date ranges can be easier done with timeline queries which we will cover in a few sections.

SynonymDomain Internal Filters

We can also filter using the internal values for synonymdomains. The sample below shows how we can use that to filter the workorder objects based on the internal values for the status attribute (bound to the WOSTATUS synonymdomain).

```
GET <collection uri>?oslc.where=...&domaininternalwhere=status!=APPR,INPRG
```

This will filter the workorder collection using the “**not in”** clause for the set of external values corresponding to the internal values of APPR and INPRG. We can use a “in” comparison by changing the expression like below:

```
GET <collection uri>?oslc.where=...&domaininternalwhere=status=APPR,INPRG
```

Few things to note here:

- The domaininternalwhere query parameter is independent of the oslc.where and is “anded” to the oslc.where clause (if oslc.where is present in the request).
- The format is <attr name>[=/!=]internal_val1,internal_val2,...
- This feature always generates a sql with “in” or a “not in” operator depending on whether the = or != operator was used.
- The list of internal values need to be comma delimited.
- There can be one or many internal values and null is not allowed in this value set.

MaxTableDomain Based Filters

We can also filter using the maxtabledomain list where clause. This is there more for backward compatibility reasons with the older rest api and in case a client wants to leverage an existing list where clause already created using the table-domain.. The api syntax is shown here.

```
GET <collection uri>?_fd=<maxtabledomain name>&_fdsite=<siteid>&_fdorg=<orgid>
```

This picks up the domains listwhere and applies that to the collection. The site and org are optional and needed if the maxtabledomain is site/org scoped.

Timeline Filters

The timeline filters allow a simpler way to filter collections with time range based queries. An example below shows how we can get all workorders reported in the past 3 months

```
GET /oslc/os/mxwodetail?tlrange=-3M&tlattribute=reportdate
```

This will simply find all the workorders with a reportdate between today and and 3 months back. The query is by default indexed around the current date. Another variation of this query would be to range on a future date just by switching the sign on the tlrange to say +3M instead of -3M. For reportdate a future date range may not be good use case, but it would for date attributes like scheduled date etc (which can be in future). You might also use the current date as an index and filter around that date using the +- notation like below

```
&tlrange=+-3Y
```

This will filter records 3 years past and 3 years to the future indexed on the current date. If current date is not what we want to index on, we can specify the date that we want to use for the index as shown below

```
&tlattribute=reportdate=<some iso date>
```

Classification Attribute Search

Maximo objects like Assets, Locations, Items, Workorders etc can be associated with classifications which provides a way to associate classification metadata (aka classification attributes) to the said objects. Maximo allows searching those objects (assets,locations etc) based on those attribute values. This search capability is what we call as Attribute Search feature. The REST apis support for this feature was added in 7606 version of Maximo base. A sample below describes an attribute search applied to the Asset Mbo using the MXAPIASSET Object Structure.

```
GET oslc/os/mxapiasset?attributesearch=[SPEED:>=50]
```

This searches for all assets that have a class spec attribute called SPEED with a value >=50. If we did not set that :>=50 and run the query as:

```
GET oslc/os/mxapiasset?attributesearch=[SPEED]
```

It would have only fetched assets that have SPEED as a spec attribute. For combining multiple attributes together in the search follow the example below:

```
GET oslc/os/mxapiasset?attributesearch=[SPEED:>=50;AREA;ELEV:=300]
```

This will search for assets that have a class spec attribute SPEED with a value greater than 50 and an attribute named AREA and attribute name ELEV with a value of 300, all and-ed together.

Note this feature is in addition to the oslc.where/savedquery query parameter. So you can use this feature along with the other filtering capabilities that are supported in this api framework.

Sorting

Sorting of collection resource can be done using the oslc.orderBy query parameter. The format for this is

```
GET <collection uri>?oslc.orderBy=-attr1,+attr2
```

The attributes prefixed with the minus sign (-) will be sorted in the descending order and the ones with the plus (+) sign will be sorted in the ascending order. All attributes listed here should be prefixed with a + or - sign. There is not default sort order here. Also related attributes are not supported here. Only persistent attributes from the main object are supported here.

Note that the + sign needs to get encoded - a common mistake made by developers when trying this one out.

Paging

Paging of data is a basic requirement and we have a query parameter **oslc.pageSize** to do just that. That parameter value is used to specify the max records to fetch for a page. A sample uri shown earlier describes how to achieve paging.

```
GET oslc/os/mxasset?oslc.pageSize=10
```

As discussed before, this causes the responseInfo object to have the **nextPage**,**previousPage**,**pagenum**,**totalCount** (optional) and **totalPages** (optional) properties to describe page navigation information.

Auto-paging

We have had long faced this question about how to auto-initiate paging and in the current release candidate (7.6.0.8) we have finally added that feature. This feature is primarily needed if we are accidentally requesting a large data set which can cause resource starvation or OOM errors in Maximo server. To prevent that, we have an attribute in the Object structure application to set the **auto-paging** threshold. For example if we set that to 1000 for MXASSET Object Structure, and the request is as below

```
GET /oslc/os/mxasset
```

Although the oslc.pageSize is not set, the system will start paging the request if the asset count starts exceeding the 1000 limit.

Limit Paging

It is also possible that while querying a client set the page size to be too high, which in turn could also cause an OOM error. To handle that scenario we have a property **mxe.oslc.maxpagesize** which can be set to a positive value for the page size. This will apply to all maximo object structures. We can also set the property specific to an Object Structure by setting the property mxe.oslc.<os name in lower case>.maxpagesize. This will override the global value set by property mxe.oslc.maxpagesize. If the request page size exceeds this value, the system will throw an error and the request will fail.

Stable Paging

Stable paging is a variation of the basic paging. In stable paging we load the MboSet in memory (note the MboSet contains 0 Mbos at this point as the Mbo’s are loaded on demand), and retain a reference to it throughout the paging process. As we page through the MboSet, the Mbo’s are discarded. Effectively at any given point in the paging, only 1 mbo is live. Note, the basic paging will also discard the Mbo’s as we serialize them and there would never be more than 1 mbo in memory for the life of the paging. However in basic paging the MboSet gets loaded for every page request ie the sql query gets fired every time. That's where stable paging scores big as it holds the MboSet reference and does not need to fire the sql for every page. The downside being, the pages expire as they get delivered (as the mbos get discarded and the MboSet is never refreshed). Sample api to initiate stable paging:

```
GET <collection uri>?oslc.pageSize=10&stablepaging=true
```

This creates a stable id which will be embedded as part of the **nextPage** uri (stableId=<some id>). We need to use that uri to page forward. We cannot reload the same page as the page expires at first load. There is no paging backward as pages including the current page have expired.

The stored MboSet will expire if it has not been accessed for 5 minutes or it has be paged to the end of the set. This idle expiry time (of 5 minutes) can be changed using the mxe.oslc.idleexiry value (which is in seconds).

You can see a live demo of the stable paging here Stable Paging and Search Terms

Filtering Child Objects

We have discussed how to leverage the rel notation to inline related MboSets as well as using the basic relation name with the curly brace notation to inline the Object Structure child objects along with the parent data. This section will cover how to limit, filter or sort those child data sets while being inlined within their parent data set. All the query parameters for this child object collection operations follow a naming pattern which is <mbo name>.<relation name>.<operator>. The mbo name token refers to the parent mbo name for the child MboSet we want to operate on. The relation name is the relation name from the parent mbo to the child MboSet. The operator determines what kind of inline operation you are trying to do on the child collection.

Below we list the query parameter types available to do this:

Where Filter

The format of this query parameter is <mbo_name>.<relation_name>.where. This will help us set the where clause filter for the child object identified by the **mbo_name** parent Mbo and the **relation_name** relation name. The syntax for the where clause exactly follows the oslc.where format with the target MboSet in context. For example if we were to filter on the polines in the MXPO Object Structure we will use this feature as below

```
GET
/oslc/os/mxpo?oslc.select=ponum,status,poline{polinenum,itemnum,linec
ost}&po.poline.where=itemnum=”*” and linecost>100.0
```

This will show only the poline’s that have an intemnum and line cost greater than 100. Note this does not in any way impact the selection of the PO records.

To support the **or clause** on the where clause (instead of the default **and**) there is a query parameter <mbo name>.<relation name>.opmodeor which if set to 1, will treat the where clause specified above as a or clause. This will result in fetching all lines that have either a itemnum or a line cost greater than 100.

Limit Filter

The format of this query parameter is <mbo_name>.<relation_name>.limit. This will help us set the limit to the number of rows we want to retrieve for the child collection.

```
GET
/oslc/os/mxpo?oslc.select=ponum,status,poline{polinenum,itemnum,linec
ost}&po.poline.limit=1
```

This will limit the number of polines loaded to only 1 for each PO in the response.

Sorting

The format of this query parameter is <mbo_name>.<relation_name>.orderBy. This will help us sorting the child collection. This follows the oslc.orderBy format and is applied to the context of the child collection.

```
GET
/oslc/os/mxpo?oslc.select=ponum,status,poline{polinenum,itemnum,linec
ost}&po.poline.orderBy=-linecost
```

This will do a descending sort on the linecost for the poline collection.

Timeline queries

The format for these query parameters are <mbo_name>.<relation_name>.tlrange for the time line range (eg -3M or +2Y) and <mbo_name>.<relation_name>.tlattribute (the attribute name of the child object on which to base the time-line on) following in the lines of the Timeline queries as explained earlier.

Note that these query child object filters help sift and limit what child objects we want to see inlined with the root object of the Object Structure. But if we need to show those child objects in a list or tabular form and then page through them and filter them, then we should consider using the relation name in the URI to traverse to that set as explained in the “Traversing to related MboSets” section. That way you will operate on the child collection independently and be able to page the collection.

JSON Schema

The rest apis describe the resource using json schema standards. Maximo metadata contains more information than json schema supports. Hence we extend that schema specification with Maximo specific properties that contains more information from the Maximo metadata. Schema’s can be accessed in couple of ways using the rest api. We have a “jsonschemas” route that provides schema to any Object Structure. An example below shows that:

```
GET /oslc/jsonschemas/mxwodetail
```

This will return the json schema for the root object of the Object structure and will also contain links to the child objects like INVRESERVE etc. To get the schema for all the objects in the OS we need to include the request parameter oslc.select=*. This will fetch all the child objects inline into the root object schema while retaining the hierarchy structure.

Note also that the properties in the schema map to the Mbo attributes (which are included as part of the OS). They also have the json schema type as well as the "subtype" that has the more specific Maximo type.

Additionally you can also specify the oslc.select clause to filter out the part of the Object structure you need. An example is shown below:

```
GET  /oslc/jsonschemas/mxwodetail?oslc.select=wonum,status,invreserve{*}
```

Will give you the details about the workorder wonum and status attributes and all attributes from the invreserve child object.

There is another popular way to get the schema while you are fetching details in a collection query. The query below is a simple collection query

```
GET
/oslc/os/mxwodetail?oslc.select=wonum,status,description,invreserve{itemnum},asset.assetnum
,asset.status
```

Now say in addition to fetching the workorder records, I wanted the schema for this oslc.select clause that is fetching part details from workorder (wonum etc), part from invreserve and part from asset (which is not even part of the OS) using the dot notation. To do that all we need to do is to add the query parameter addschema=1 to the request url. By doing that the response json objects "responseInfo" property will have the schema inlined inside it. This is like getting the data as well as the metadata al in the same rest api call. Note that this schema will not be fetched for the next page request as the next page url will not have the addschema=1 in the uri. However if you add that query parameter explicitly, it will fetch the schema for any page.

In 7609 we will be adding support for mbo schemas too. This is critical for use cases where we fetch a related mboset using the rest api without using the responseos query parameter. For example say we wanted to evaluate the getList api for an attribute - say status for a given workorder. The api below will do the job

```
GET /oslc/os/mxwodetail/{id}/getlist~status?oslc.select=*
```

This will return the possible list of status values for that workorder state. Now the response is a serialized version of the synonymdomain Mbo and is not an Object Structure. If we wanted to have a schema for the response, we can add the query parameter &addschema=1 and that will work for the response mboset without needing to set it as an Object Structure.

This Mbo schema can also be accessed standalone using the route jsonmboschemas. A sample call is shown below:

GET
/oslc/jsonmboschemas/asset?oslc.select=assetnum,status,location.description,location.status,rel.openwo{wonum,status}&addschema=1

This will return the json schema for asset Mbo with the attributes assetnum and status along with the related attributes from location and workorder (using the rel.openwo).

On a similar note, when we access some relation as a Mbo(Set), we can apply json schemas there too without needing to use an Object structure for this. A sample below will show the use case

GET /oslc/os/mxpo/{rest id}/vendor?addschema=1&oslc.select=*

Here we are accessing the related vendor for the PO and we are not using any responseos query param to render the response to as an OS and yet we can access the schema of the "vendor" (which is the companies mbo).

Note this mbo schema is a 7609 feature and is not going to be available in prior releases. The OS schema is however available for releases before 7609.

Creating and Updating resources

Creating resources are almost always done using the collection uri - the same uri you would use to query the resources. For example the api call for creating assets is shown below.

```
POST /oslc/os/mxasset

{
    "assetnum":"ASSET1",
    "siteid":"BEDFORD",
    "description":"my first asset"
}
```

Once you create the asset, you will get a response that contains a **location** header with the URI of the newly created asset. You can now use that “location” uri to fetch the newly created resource. Rather than doing a GET to fetch the newly created resource, you may want the response of the create to contain your newly created resource. For that you can add the request header “**properties**”. The properties header follows the syntax of the “oslc.select” clause. You can use that to fetch all properties, partial set of properties, related mbo attributes, formula properties etc (everything that you can do with the select clause).

```
POST /oslc/os/mxasset
properties: *

{
...
}
```

OR

```
POST /oslc/os/mxasset
properties: assetnum,status

{
...
}
```

Using this properties header, we can remove to the need to do an extra GET for every create/update.

Now that we have created an asset, we will try to update it. For example, say we want to set a location and a description to the asset.

```
POST /oslc/os/mxasset/{rest id for the asset}
x-method-override: PATCH
properties:*

{
    “location”:”BR300”,
    “description”:”test asset desc”
}
```

Note that the url we have a “rest id…” at the end of the collection url. This URI is pointing to a member asset in the collection - hence the “rest id” token after the collection uri. Note that this rest id is not the unique id for the Mbo. It’s a generated id for the Mbo that is created using the primary-key attribute values. Also take note of the x-method-override request header and the value PATCH. This instructs the server side to update the resource. As with create, you can specify the properties request header, to get back the results of an update. The sample here shows a value of *. We would generally recommend avoiding use of * (unless you really want all properties) and instead use the selected set of properties that you care about (works our better for performance).

Note that we needed to fetch the uri of the mxasset reasource in order to update it. Fetching of that URI can be done in 2 ways:

- We can get the uri as part of a GET collection query which will return all select members with their uris.
- As part of the response "location" header when we create a resource.

Although this is the prevalent design for uri interaction in REST paradigm, in some cases you may not have the uri and still need to update the asset based on other key information. We will discuss how to do that shortly.

Next let's try to add some child objects to this. For this example, we will add assetmeters to the newly creates asset. The api below adds 2 assetmeters to an asset..

```
POST /oslc/os/mxasset/{rest id for the asset}
x-method-override: PATCH
patchtype: MERGE
properties:*

{
      "description":"test asset desc",
     "assetmeter":[
      {
               "metername":"TEMP-F",
               "linearassetid":0
      },
      {
               "metername":"ABC",
               "linearassetid":0
      },

     ]
}
```

Note that here we used an extra request header called **patchtype** with a value of MERGE. This instructs the server side to try to match the child objects - like assetmeter with the existing assetmters for this asset. If it finds a match, that assetmeter gets updates with the request assetmeter. If there is no match, it will create a new assetmeter. We call this the MERGE api. Say the asset to be updated had 3 existing assetmeters, and the request contained 1 existing assetmeter and 1 new assetmeter. After the merge call, that asset will be having 4 assetmeters - with one newly created one and another updated assetmeter.

To highlight the difference between a PATCH and MERGE call, we can run the same request but without the patchtype header, on another similar asset with 3 assetmeters. The server side will create a new assetmeter and update the existing assetmeter just as in the MERGE call. Unlike the MERGE call, it will end up deleting the assetmeters that were not in the request. In this example the asset with be left with only 2 assetmeters (that are in the request). Thus in PATCH request, the server side will delete all child objects that are not in the request payload.

Now let's talk about updating the child objects. In this example we will update the assetmeter object to set the meter reading. Make sure the MXASSET Object structure has the np attributes "newreading" and "newreadingdate" included.

```
POST /oslc/os/mxasset/{rest id for the asset}
x-method-override: PATCH
patchtype: MERGE
properties:*

{
    "description":"test asset desc - updating temp meter",
    "assetmeter":[
      {
                "metername":"TEMP-F",
                "linearassetid":0,
          "newreading":"10"
      }
     ]
}
```

Notice that we included the primary keys of the assetmeter - metername and linearassetid for the meter update. Another option that the api allows is to use the href uri for the assetmeter instead of the primary keys. The sample below shows that.

```
POST /oslc/os/mxasset/{rest id for the asset}
x-method-override: PATCH
patchtype: MERGE
properties:*

{
    "description":"test asset desc - updating temp meter with child
uri",
    "assetmeter":[
      {
```

```
                   “href”:”parent uri#encoded_child_keys_anchored”,
             “newreading”:”10”
        }
       ]
}
```

If you are wondering how to selectively delete child object, the request below shows just that.

```
POST  /oslc/os/mxasset/{rest id for the asset}
x-method-override: PATCH
patchtype: MERGE
properties:*

{
    “assetmeter”:[
     {
                   “metername”:”TEMP-F”,
                   “linearassetid”:0,
             “_action”:”Delete”
     }
    ]
}
```

Note the use of child level actions (_action) to delete the assetmeter. You could have also used the child href instead of the primary keys as shown before. You could have also used the http DELETE method to delete the child object using the child object “localuri” URI.

```
DELETE <assetmeter localuri>
```

Note local uri for a child object (in Object Structure) is something you can use to refer directly to the child object. You cannot do the same with the child href as that is an anchored uri. For example you cannot use the child href for http DELETE.

Note that using the POST method with the _action child action, you can do a bulk delete of child objects, which you cannot do using the http DELETE method.

Deleting Resources

In the last section we talked about deleting child objects in an Object Structure resource. Any updation of the resources properties including adding, updating or **deleting** child objects is done under the update api for the resource. Deleting the resource (Object Structure root mbo) can be done using the sample api below:

```
DELETE /oslc/os/<os name>/{rest id}
```

You will get a response of 200 OK if the delete is successful. Starting 7608 we should also be able to do the same using the following api below

```
POST  /oslc/os/<os name>/{rest id}
x-method-override: PATCH

{
     "_action":"Delete"
}
```

Note in certain cases some browsers do not allow the http DELETE and this POST equivalent is a good way to get around that.

It is also important to consider the fact that in Maximo, deleting a Mbo should internally delete the dependent child objects. The REST api merely invokes the delete routine on the Mbo, and is not responsible for what the mbo does internally to delete the dependent child objects.

Additionally there is an action api that can be leveraged to figure out if this Mbo can be deleted. This is needed to handle use cases where we would want to delete the resource in 2 steps ie to mark it for delete and then delete it later. To mark a resource for delete, we will at least need to verify if the resource can be deleted. Fortunately there is a Mbo call back api - canDelete() just for that purpose. This REST api sample below shows how to invoke that

```
GET  /oslc/os/<os name>/{rest id}?action=system:candelete
```

A 200 OK response would indicate that the resource can be deleted. Note this depends on the application Mbo implementing the canDelete method, which is not implemented by default in the Mbo framework. If delete is not allowed, the response would an error json indicating the reason behind it as determined by the application Mbo.

We will talk more about actions in REST api in the next section.

Actions

Sometime back in Maximo we had introduced the functionality of “Standard Services”. The name may sound confusing the but the intent was fairly simple. Maximo has a bunch of application services - aka Appservices which provide a bunch of service methods that act on Mbos to perform some business task. While most of these calls end up modifying the state of the system, some of these calls are merely for getting information. We have leveraged the JSR 181 annotations to expose these methods as “WebMethods” which can be accessed using

SOAP and REST calls. For the purpose of this discussion, we are only going to focus on the RESTFul aspects of these methods.

Simply put, we can add a WebMethod to any existing Maximo App service by extending the service class and adding a method like below. This example is from say extending the Workorder Service (psdi.app.workorder.WOService):

```
@WebMethod
public void approve(@WSMboKey(value="WORKORDER") MboRemote wo, String memo)
{
   wo.changeStatus("APPR",MXServer.getMXServer().getDate(),memo);
}
```

Next you want to make sure that your method is now available for REST api calls. To do that you can select any workorder record and use the rest api to get the set of allowed actions on that resource. For example the query below will do the job.

```
GET /oslc/os/mxwo?oslc.where=wonum="1001"&oslc,select=allowedactions
```

You will see the list of allowed actions, which are nothing but the list of WebMethods for the service corresponding to the root object of the Object Structure along with the list of actions registered with the Object structure (using the application menu "Action Definition"). The resulting json would provide the json schema for web method as well as the HTTP method and the name of the action, which can be used to invoke the action. A sample call below will demonstrate how these methods can be invoked.

```
POST <uri of the workorder 1001>?action=wsmethod:approve
x-method-override: PATCH

{
   "memo":"Testing"
}
```

Note we need to set the request header x-method-override as PATCH for invoking this action api, as this is operating on an existing Mbo (the first parameter to the method). As expected, this mbo reference is derived from the request URI (say - uri of the workorder 1001) and passed into the method as part of the api invocation. The json schema for the payload will reflect the parameter data types (in the method) and the parameter names.

As you can figure out, the query parameter action=wsmethod:approve is the fully qualified name of the action. The format is <action type>:<name> where name is the java method name for the action type **wsmethod**.

You might already have guessed the limitation this has with overloaded methods. Currently overloaded methods are not supported for REST api calls.

There are quite a few of these methods that are available in the out of the box service. I would recommend using the “allowedactions” to check out these methods. We hope to be adding more and more of these methods every release as we evolve our application apis.

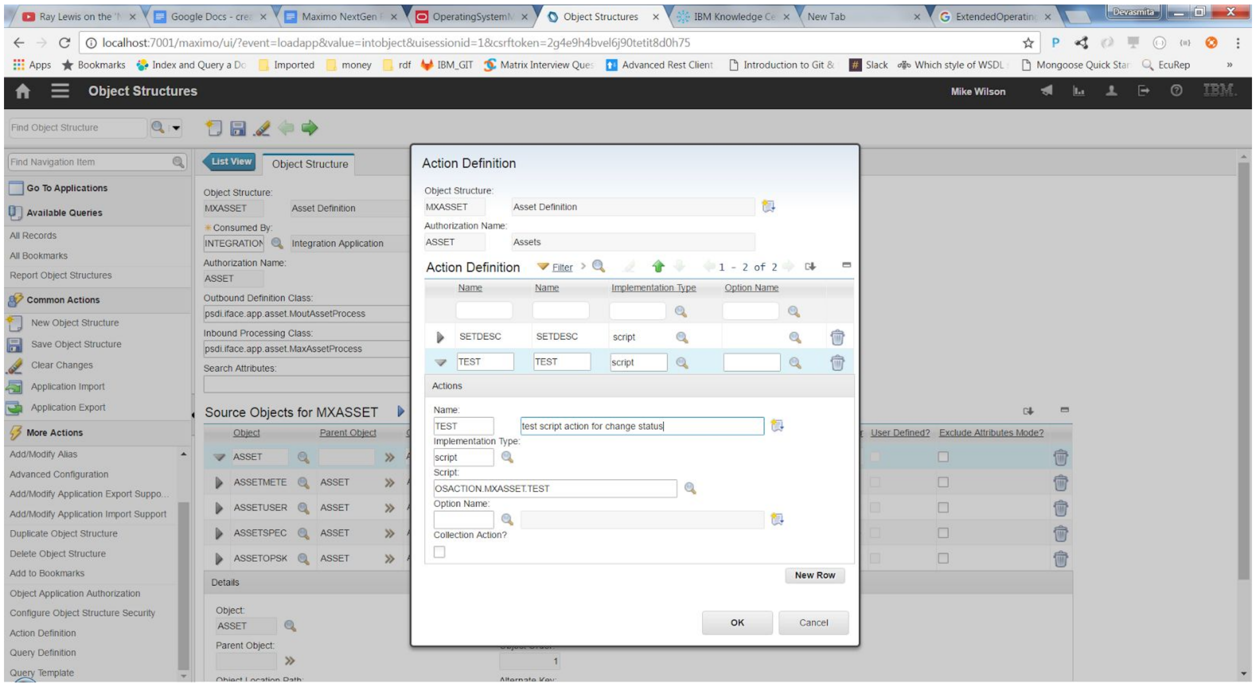
Besides the webmethod based actions, the REST apis support scripted actions too. Effectively you can write a REST action code using automation scripts. A simple example below will explain this concept.

First we need to create an automation script from the Autoscript application using the “Create Screipt For Integration” option. We need to select the Object Structure and then select the type of script as “Action Processing” and give a name for the action. Now you can write a script (in any language - python, js) using the implicit variable **mbo**. For example the script below changes the status of Asset to “OPERATING”,

```
mbo.changeStatus(“OPERATING”,False,False,False,False)
```

Save the script.
Note that you did not write any code to commit the modification to the Asset mbo. The REST api framework commits the transaction after the method completes.

Next you need to register that script with the Object Structure. Goto the Object Structure application and select the “Action Definition” menu action and set the script as an action to the Object Structure as shown below. Name the action same as the name of the script. Choose OK to save the configuration.



Action Definition

Object Structure: MXASSET Asset Definition

Authorization Name: ASSET Assets

Action Definition

Name	Name	Implementation Type	Option Name
SETDESC	SETDESC	script	
TEST	TEST	script	

Actions

Name: TEST test script action for change status

Implementation Type: script

Script: OSACTION.MXASSET.TEST

Option Name:

Collection Action? ☐

Next we can invoke this script with the request like below.

POST <uri of the asset 1001>?action=TEST
x-method-override: PATCH

And you will see that the asset status changed to OPERATING. Note that if you wanted to get the changed Asset in response, you can always add the request header **properties** with your desired value. Note you can also

You can also use this actions rest apis to invoke workflows. A sample below shows how workflow “ABC” for asset can be invoked.

POST <uri of the asset 1001>?action=workflow:ABC
x-method-override: PATCH

For non-interactive workflows, this is all we need to do to initiate it. Interactive workflows are those that require user interaction by placement of assignments, input and interaction nodes. Interactive workflows will be covered in the “Interfacing with the Workflow Engine” section.

Automation Scripts

The rest api’s have a tight integration with the automation scripts. Automation scripts can be leveraged to develop custom apis. A sample call shown below describes how automation scripts interact with rest apis.

Say we want to find the total number of in progress work and service requests in a given site. Since there is no out of the box api for that, you will end up writing one for yourself. At this point in Maximo, let’s say that we prefer to write any custom feature using automation scripting. Since this is rest apis with json response, we would prefer to use the JavaScript language for scripting. Say we want to call this api **countofwoandsr.** The api should look like this

```
GET /oslc/script/countofwoandsr?site=ABC
```

We would like to get a response in json as below

```
{
    “wocount”:100,
    “srcount”:20,
    “total”:120
}
```

Now let us write the script to get this api going.

Script Name: countofwoandsr

```
importPackage(Packages.psdi.server);

var resp = {};
var site = request.getQueryParam("site");
var woset =
MXServer.getMXServer().getMboSet("workorder",request.getUserInfo());
woset.setQbe("siteid","="+site);
var woCount = woset.count();
resp.wocount = woCount;

var srset =
MXServer.getMXServer().getMboSet("sr",request.getUserInfo());
srset.setQbe("siteid","="+site);
var srCount = srset.count();
resp.srcount = srCount;
resp.total = srCount+woCount;
```

```
var responseBody = JSON.stringify(resp);
```

That is pretty much what we should need to get this api going. All we need to do now is to save the script and open up our browser and fire off the GET request and see if the results come up as expected.

We talked about the GET support with automation scripts. We can now talk about how to leverage POST apis with scripting.

Automation scripts are also used for implementing custom queries and custom actions. We will discuss those in their respective sections.

Bulk Operations

With Maximo RESTful JSON API, you can process multiple resources with multiple operations in a single transaction.

The bulk process only supported by using collection url. You will use POST method and x-method override value BULK in the header.
The multiple resources and operations are provided in the message body with a JSON array. Each resource has an element called _data (reserved name) in which the data for the resource is provided. For update and delete, the resource has an element called _meta (reserved name).

Operation	Data Object Example
CREATE	`{"_data":{"assetnum": "test-5", "siteid": "BEDFORD", "description": "TS test 5"}}`
UPDATE	`{"_data":{"description": "New Description"}, "_meta":{ "uri":"resource uri", "method": "PATCH","patchtype":"MERGE"}}`
DELETE	`{"_meta":{ "uri":"resource uri", "method": "DELETE" }}`

Unless there is a syntax type of error in your JSON data, for bulk load, you will always get a response code of 200. However, you need to process the response to determine which resources were updated successfully or not.

Multiple Resources Creation with BULK

After reading this section, you can create multiple resources in a single transaction. As the regular creation, each of the resources will go through the validation process. And the response for each of the resources will be shown in response JSON.

NOTE: The processing performs a Commit for each resource.

```
POST oslc/os/mxasset
X-method-override: BULK
[{
"_data":{
"assetnum": "test-5", "siteid": "BEDFORD", "description": "TS
test 5"}
}, {
"_data":{
"assetnum": "test-6", "siteid": "BEDFORD", "description": "TS
test 6" }
}]
```

If the first asset failed the validation process by having the invalid site, the http response code will still be 200. However, the error message will be shown in response like following,

```
POST oslc/os/mxasset
X-method-override: BULK

[{
"_data":{
"assetnum": "test-5", "siteid": "BEDFORDXXYY", "description":
"TS test 5"}
}, {
"_data":{
"assetnum": "test-6", "siteid": "BEDFORD", "description": "TS
test 6" }
}]
```

Response JSON:

```
[{
"_responsedata": {
     "Error": {
          "message": "BMXAA4153E - [BEDFORDXXYY is not a valid
     site. Enter a valid Site value as defined in the
     Organization Application.]",
          "statusCode": "400",
          "reasonCode": "BMXAA4153E",
          "extendedError": {
               "moreInfo": {"href":"error message uri"}
          }
     }
},{
"_responsemeta": {
     "ETag": "1992365297",
     "status": "201",
     "Location": "oslc/os/mxasset/_VEVTVC0zNy9CRURGT1JE"
     }
}]
```

The first assets are successful and returns a 400 with an error message. The second asset returns a 201 with the URI to the asset resource.

Starting 7608, we have simplified some of the request json structures for bulk request. For example we can now support _action at the data level and we can avoid the _meta and _data altogether.

POST /oslc/os/mxapiasset

properties:*
X-method-override: BULK

```
[
  {
     “assetnum”:”...”,
     “siteid”:”....”,
     “description”:”...”,
     “_action”:”Add”
  },
  {
     “href”:”...”,
     “description”:”...”,
     “_action”:”Update”
  },
  {
     “href”:”...”,
     “_action”:”Delete”
  }
]
```

As you can see, we avoid the _data and _meta with use of _action and href embedded inside the data.

For example we can support bulk change status as shown below:

```
POST /oslc/os/mxapiwodetail?action=wsmethod:changeStatus
X-method-override: BULK

[
  {
      “status”:”APPR”,
      “href”:”....”
  },
  {
      “status”:”INPRG”,
      “href”:”....”
  }
]
```

The response format has not changed.

Multiple Operations with BULK

The examples in first section shows the creation of multiple assets using the BULK processing. You can also use BULK to perform a mix of create, update and delete of asset resources in a single transaction.

To support this, in addition to _data which is used to provide the json data for a resource in a BULK transaction, you can also provide meta data using _meta (another reserved name). The metadata that can be provided are:

Metadata	Description
method	It is the equivalent of the x-method-override header discussed earlier in this document. When POSTing to create a new resource, there is need to provide a method. To perform an update you provide the value PATCH and for a delete you provide the value DELETE.
uri	It is the resource uri when processing an Update or Delete. This is required when updating or deleting a resource.
patchtype	It allows the support of MERGE when processing an update (as described earlier in this document).

Below is an example JSON data that will

- Update an asset with a 'New Description'
- Create a new asset (test-100)
- Delete an asset

```
POST oslc/os/mxasset
X-method-override: BULK

[{
     "_data":{"description": "New Description"},
     "_meta":{"uri":"resource uri","method": "PATCH",
     "patchtype":"MERGE"}
},{
     "_data":{"assetnum": "test-100","siteid": "BEDFORD",
     "description": "New Asset 100"}
},{
     "_meta":{ "uri":"resource uri", "method": "DELETE" }
}]
```

The first asset "_meta" data includes the URI to identify the along with headers identifying that it is a Patch (an update) with a type of Merge.
The second asset provides no 'meta' data since it is a Create and no "_meta" data is applicable.

The third assets provides only the `"_meta"` data to identify the asset to be deleted.

As with the Creation example earlier in this section, the response code will be a 200 but you must examine the response information in the response JSON body to determine if processing of each asset was successful or not.

Handling Attachments

Attachments in Maximo are documents, files or images that are attached to a resource such as Asset or Service Request. The Maximo RESTful API supports the retrieval of attachments that are associated to resources.

To fetch, create, update or delete an attachment for resource by API, take MXASSET as an example, you need,

1. Enable the Maximo attachments feature.
2. Configure the MXASSET object structure with the DOCLINKS MBO as a child to the ASSET object.

Fetch the attachments

When you query a specific resource (using its ID) that has an attachment, you will get a doclinks URL returned for the attachment:

```
GET /oslc/os/mxasset/{rest id}

{
      ...
      "assetnum": "1001",
      "changedate": "1999-03-31T16:53:00-05:00",
      "doclinks": {
            "href": "oslc/os/mxasset/{rest id}/doclinks"
      },
      ...
}
```

Using the doclinks url from the JSON data above, you will be get a list of attached documents (reference to those documents) along with the metadata.

```
GET oslc/os/mxasset/{rest id}/doclinks

{
     "href": "oslc/os/mxasset/{rest id}/doclinks" ,
     "member": [
     {
          "href": "oslc/os/mxasset/{rest id}/doclinks/{id}",
          "describedBy":
          {
               "docinfoid": ..,
               "addinfo": false,
               "weburl": "ATTACHMENTS/Presentation1.ppt",
               "docType": "Attachments",
               "changeby": "WILSON",
               "createby": "WILSON",
               "copylinktowo": false,
               "show": false,
               "format":
               {
                    "label": "application/vnd.ms-powerpoint",
                    "href":"http://purl.org/NET/mediatypes/applicat
               ion/vnd.mspowerpoint"
               },
               "getlatestversion": true,
               "ownerid": ..,
               "printthrulink": false,
               "urlType": "FILE",
               "upload": false,
               "attachmentSize": 101376,
               "modified": "2015-12-04T09:54:24-05:00",
               "title": "ATTACH1",
               "created": "2015-12-04T09:53:43-05:00",
               "description": "Test Attachment 1",
               "fileName": "Presentation1.ppt",
               "ownertable": "ASSET",
               "href":"oslc/os/mxasset/{rest id}/doclinks/
               meta/{id}",
               "identifier": "76"
               }
          }
     ]
     ...
}
```

The .../doclinks/{id} url is the link to the actual attachment file. The content of the attachment could be fetched by,

```
GET oslc/os/mxasset/{rest id}/doclinks/{id}
```

The .../doclinks/meta/{id} url is the link to the metadata for the attachment file.

Create the attachments

The API supports the creation of attachments that are associated to resources. For example, you created an asset and now you want to attach a PDF file that describes the maintenance procedures for that asset.

To create a new attachment, you will need an doclinks URL for resource. As shown in fetching the attachment, take MXASSET as an example, the URL looks like as following,

```
"doclinks":{
      "href": "oslc/os/mxasset/{asset rest id}/doclinks"
},
```

Note: in the current version of the API you can create an attachment for a resource(asset) only after the resource exists in Maximo. You cannot create the attachment at the time of creating the resource.

An attachment is made up of two components,

Attachment file: You create an attachment using HTTP POST with binary content or base64 binary content. There is no support for multi-part messages yet.

Related metadata of the attachment: When creating an attachment for a resource there is a limited set of metadata that can be provided (along with the file) using HTTP Headers:

Header	Value	Description
slug	File Name	The name of the attachment file

encslug	File Name	If the attachment file name has non-ascii characters it can be provided in the header base64 encoded. It is suggested that you always base64 encode your file name using this property if you believe you might have a mix of non-ascii characters
Content-Type	"text/plain"	Based on the type of attachment - text/plain supports a .txt file
x-document-meta	Attachments	Tied to the DOCTYPES domain that defines the supported attachment types
x-document-description	Description	The description of the document
x-document-encdescription	Description	If the description has non-ASCII characters, it can be provided in the header base64 encoded. It is suggested that you always base64 encode your description using this property if you believe you might have a mix of non-ascii characters
custom-encoding	"base64"	This header facilitates testing using a browser client such as RESTClient (for FF). Allows you to paste in a base64 encoded image into the Body of the tool (otherwise you need to test with programmatic tool). You can use public tools to base64 encode your image file

```
POST oslc/os/mxasset/{asset rest id}/doclinks
X-document-meta: FILE/Attachments
Slug: test.txt
X-document-description:test file

Hello this is my first test file
```

The response Location header contains the url for the uploaded attachment (sample shown below).

```
Location: oslc/os/mxasset/{asset rest id}/doclinks/{id}
```

When we GET on that url we will get the attached document that we uploaded before. Along with that, it will also have a response header named Link which will have a URL to the metadata for this attachment.

```
Link: oslc/os/mxasset/{asset rest id}/doclinks/meta/{id}
```

This "meta" link can be used to get the metadata for the attachment. A GET on that link will fetch the json representation of the document description, mimetype etc as shown in a sample below.

To create attachments of WWW (url) type, we can use the following request as a sample.

```
POST /oslc/os/mxasset/{asset rest id}/doclinks
X-document-meta: URL/Attachments
Slug: CNN
Content-location: www.cnn.com
X-document-description:cnn web site
```

And in the response you will get a Location header with the url of the newly created url attachment. Note that the url was set on the content-location request header. The slug request header is used as the name of the attachment.

Another important thing to note: the x-document-meta request header has 2 parts - the url type/document type. The url type is a synonymdomain in Maximo and hence hardcoding the values FILE or URL maybe a problem in case those values have been modified at the customer installation. You could potentially do one of the 2:
1. Avoid specifying the url type altogether. The api framework would default the url type based on your request. For example if the request has the content-location header, it will be treated as a url type with internal value of WWW. Otherwise it will treated as a url type of FILE. In each of these cases the system will use the default external value for these internal values (FILE or WWW).
2. The other option would be to fetch the external values of the FILE and WWW types and then use that in the client side code to set the x-document-meta.

We tend to prefer the first approach as its simpler of the client.

Update the attachments

There is no support for updating an attachment, you would need to delete the current version and create a new version.

Delete the attachments

The attachments could be deleted by using HTTP POST with the URL of the attachment and providing the x-method-override header with a value of DELETE.

```
POST oslc/os/mxasset/_MTAwMS9CRURGT1JE/doclinks/80
x-method-override: DELETE
```

Handling attachments as part of the resource json

Starting 7606 we support handling attachments as part of the Object structure json. Here is a sample that adds 2 attached documents as part of asset creation.

POST /oslc/os/mxapiasset

```
{
  "assetnum":"TEST299",
  "siteid":"BEDFORD",
  "doclinks":[
  {
    "urltype":"FILE",
    "documentdata":"aGV5IGhvdyBhcmUgeW91",
    "doctype":"Attachments",
    "urlname":"greetingsabcd.txt"
  },
  {
    "urltype":"FILE",
    "documentdata":"aGV5IGhvdyBpcyB0aGF0",
    "doctype":"Attachments",
    "urlname":"howisthatfor.txt"
  }
  ]
}
```

As you can see the “documentdata” attribute has the base64 encoded document.

Aggregation

Aggregation API provides the aggregation function based on restful API. After reading this section, you will know how to use it getting the aggregated results back. The results also provide collection links which only give the records in that group.

We mainly have the following query parameters for aggregation,

Query Parameters	Description	Example
gbcols	Define the attributes and aggregation function	gbcols=siteid, min.budgetcost, max.budgetcost,avg.totalcost
gbfilters	Provide the ability to filter the aggregation result to a smaller set	gbfilters=siteid="BEDFORD"
gbsortby	Provide the ability to sort the aggregation result	gbsortby=-siteid
gbrelprop	Provide the ability to get the related property back	
gbrange	Provide the ability to get the ranged aggregation result *(only support count)*	gbrange=assethealth

Aggregation Column

The aggregation API is running based on the attribute and aggregation function we define in **gbcols.** For example, we are trying to get the minimum, maximum budget cost, the average total cost and the count number of the assets for all of the site.
We must provide at least one attribute as the grouping on attribute. In our case, we use **siteid**.

Here is the table of syntax, along with the example,

Aggregation	Description	Example
max.attributename	The maximum value of the attribute	max.budgetcost
min.attributename	The minimum value of the attribute	min.budgetcost

avg.attributename	The average value of the attribute	avg.totalcost
count.attributename	The count number of record	count.*

Finally, gbcols=siteid,max.budgetcost,min.budgetcost,avg.totalcost,count.*

In the result set, the attribute like siteid will show the grouped value. The aggregation attributes like max.attributename will show as max_attributename. The result will also include the data as well as a Resource Collection link which will give you the resources in current group.

```
GET
/oslc/os/mxapiasset?gbcols=siteid,count.*,min.budgetcost,max.budgetcost,avg.tot
alcost
```

Expected result:

```
  {
     "count": 75,
     "max_budgetcost": 2765.00,
     "collectionref":
".../oslc/os/mxapiasset?&oslc.where=siteid%3D%22BOSTON%22",
     "avg_totalcost": 30901.17333333333333333333,
     "siteid": "BOSTON",
     "min_budgetcost": 390.00
  },
```

The gbcols also support multiple level aggregation. We could build the following parameters to get the aggregation results for organization and site.

```
GET
/oslc/os/mxapiasset?gbcols=orgid,siteid,count.*,min.budgetcost,max.budgetcost,a
vg.totalcost
```

Expected result:

```
  {
     "count": 551,
     "orgid": "EAGLENA",
     "max_budgetcost": 25000.00,
     "collectionref": collection ref
     "avg_totalcost": 436.87392014519056261343012,
     "siteid": "BEDFORD",
     "min_budgetcost": 0.00
  },
```

Aggregation Filter

After you get the aggregation result set, it is possible that you want to get the smaller set based on the requirements. In this case, we introduce the **gbfilter** which is the having clause in SQL term. The value for this query parameter must follow the SQL clause format. For example, you want to get the grouped result only for BEDFORD. Then we will have,

```
GET /oslc/os/mxapiasset?gbcols=siteid,count.*&gbfilter=siteid='BEDFORD'
```

Expected result:

```
{
      "count": 551,
      "sum_totalcost": 240717.53,
      "collectionref":
".../oslc/os/mxapiasset?&oslc.where=siteid%3D%22BEDFORD%22",
      "siteid": "BEDFORD"
}
```

Aggregation Sort By

In aggregation API, we can define the **gbsortby** value to get the sorted result back. For example, you want to sort the result set by Site in descending order.

```
GET /oslc/os/mxapiasset?gbcols=siteid,count.*&gbsortby=-siteid
```

For ascending order, the item should be gbsortby=+siteid. However, when we test in browser, it is necessary to encode the value for gbsortby. There is online site such as http://meyerweb.com/eric/tools/dencoder/ helping you to do it. Copy the value +*siteid* from URL and encode it. Bring it and copy to url. Finally, we will get gbsortby= %2Bsiteid for showing result by Site in ascending order.

```
GET /oslc/os/mxapiasset?gbcols=siteid,count.*&gbsortby=%2Bsiteid
```

Aggregation Range

In aggregation API, it is also possible to show the result set in different group in one range with **gbrange**. In Maximo 7606, we only support count.

String(ALN) value:
For example, you are trying to get the total count for work orders which is in WAPPR or APPR status. You need to append the following term to the URL.
In result set, the group for APPR and WAPPR will be ranged to one new group with the total count and the collection link. All the other result, like status="CAN" will be shown as the regular grouped result.

```
GET
/maximo/oslc/os/mxwodetail?gbcols=status,count.*&gbrange=status={[APP
R:WAPPR]}
```

Expected result:

```
  {
      "count": 2388,
      "status": [
           "APPR",
           "WAPPR"
      ],
      "collectionref":
".../oslc/os/mxwodetail?oslc.where=status+in+%5B%22APPR%22%2C%22WAPPR
%22%5D"
  }
```

Numeric value:
We support numeric value as well for gbrange as well. For example, you want to get the count for workorder. The first range is 1<=worpriority<=3, the second range is 4<=wopriority<7
The rules to build the range is following the mathematics. "[" and "]" means greater or less than including the side value. "(" and ")" means greater or less than exclude the side value. In our case, we build,

```
gbrange=wopriorty={[1..3],[4..7)}
```

We will see the priority=7 group has been excluded from the ranged result.

```
GET /oslc/os/mxwodetail?gbcols=wopriority,count.*&gbrange=wopriority={[1..3],[4..7)}
```

Expected result:

```
{
      "count": 32,
      "collectionref":
".../oslc/os/mxwodetail?&oslc.where=wopriority%3D7",
      "wopriority": 7
 },
 {
      "count": 390,
      "collectionref":
".../oslc/os/mxwodetail?&oslc.where=wopriority%3E%3D4+and+wopriority%
3C7",
      "wopriority": [
           4.0,
           7.0
      ]
   },
   {
      "count": 66415,
      "collectionref":
".../oslc/os/mxwodetail?&oslc.where=wopriority%3E%3D1+and+wopriority%
3C%3D3",
      "wopriority": [
           1.0,
           3.0
      ]
   }
```

Selecting Distinct Data

Selecting distinct data can be done by replacing the oslc.select clause in a collection url with query parameter distinct=<attribute name>. A sample is shown below

GET /os/mxapiasset?distinct=siteid&oslc.where=...

Will respond with an json array of sites that match the said where clause.

[“BEDFORD”,”NASHUA”...]

Dealing With Hierarchical Data

Maximo has a lot of hierarchical objects - Locations, Workorders, Assets, Failure codes etc. There are subtle differences between each of these hierarchies and hence we would like to talk about each one of these separately.

Location Hierarchy

The location hierarchy is Maximo is always scoped under the LOCSYSTEM object. Effectively a given location can belong to multiple systems, and hence different hierarchies. The api is designed such a way that we start off with the list of systems that is available for the user. And then we dive into the hierarchy under the scope of that system.

We have an object structure - MXAPILOCSYSTEM that can be leveraged to get a list of systems. The api below shows how to do just that.

GET /os/mxapilocsystem?oslc.select=systemid,description

For there, we would like to go to the top (root) location under that system. You should select the system that you want to drill down using the href of the system record. The api call below shows how to do that.

GET /os/mxapilocsystem/{id}/topleveloc.mxapioperloc?oslc.select=systemid,description

Here "toplevelloc" is the name of the relation from LOCSYSTEM to LOCATIONS table. Notice that we have appended that object structure name to the relation name to get the response as the MXAPIOPERLOC os. This helps us jump from one object structure to the other using the rest apis.

Next we would like to drill down under that top level location. For that we take the href of the location object that we got and append the relation name **syschildren** with the os name mxapioperloc appended to it (such that we stay in the context of the mxapioperloc os). The api for that is shown below

GET /os/mxapioperloc/{id}/syschildren.mxapioperloc?ctx=systemid=<systemid>

You will also note that we used a query parameter ctx with the value of systemid=<systemid of the system that we are drilling down into>. This is needed as a location can belong to multiple systems and hence multiple hierarchies. So while we drill down into the hierarchy, we need to provide the systemid context.

Now you can take any of the locations in the response and drill down by following the api described above. Always remember to set the ctx query parameter to the right systemid or else the api will default to the primary system for that location for the drill down.

You will also notice that these collection responses can be filtered, sorted or paged just like any other os query response. So if you are building a tree structure in the UI using these apis, you can introduce a sorting or filtering function at each tree node (which is a location).

General Ledger Component Hierarchies

GL component hierarchies provides another flavor of Maximo hierarchies. The general ledger account consists of segments (gl components) which follow a certain hierarchy as defined in the chartofaccounts and glcomponents table. The glcomponents table defines all the components and their gl order. In order for an end user to specify a general ledger account we would need to provide an api for looking up the segments in a hierarchical way (following the gl order). The set of apis below describes just that.

GET /oslc/glcomp?lean=1&oslc.select=*

The response will be a list of gl segments at the top level ie gl order 0. For each of the records look for the childcompref uri. If we do a GET on that url we will get the child records for that segment. Note that the json also has a responseInfo that provides some metadata about the current segment (glsegmentcurrent) as well as the total number of segments (glsegmentcount). The **glcompsofar** describes the account that has been selected so far. At the start it will just use the metacharacters ? (as configured) and the segment separators to represent the account. The GET on the childcompref would look like the api call below.

GET /oslc/glcomp?glcomp=<comp0>&oslc.select=*

The collection of records you get would be the next set of segments that are valid for the segments selected so far as described in the glcomp query parameter value. Note the glcomp query parameter value gets updated to point to the next set of segments as we drill down. Internally it uses the | separator for the segments and hence the childcompref url for the 3rd set of segments will look like

GET /oslc/glcomp?glcomp=<comp0|comp1>&oslc.select=*

You can instead also specify the **glcompsofar** value to drill down too. The sample api call below shows how to do that.

GET /oslc/glcomp?glvalue=<comp0-comp1-???>&oslc.select=*

Note the use of glvalue query parameter to get the values. This will give the exact same results as the glcomp=comp0|comp1 api call.

As is the case with the other hierarchies, we can sort (oslc.orderBy) and filter (oslc.where) with these apis.

Interfacing with the Workflow Engine

Initiating a workflow for a given Mbo can be done by using the api shown below:

```
POST /oslc/os/<os name>/{rest id}?action=workflow:<workflow name>
X-method-override: PATCH
```

This will invoke the named workflow in the context of the mbo identified in the URI.

Handling Task Nodes

After initiation the workflow might end up into a task node, which generates an assignment. The apis below shows how you can handle assignments.

The first step would be for the user to fetch his/her assignments. For an user to fetch assignments use the api below

```
GET /oslc/os/mxapiwfassignment?oslc.select=*
```

This will fetch all the assignments for that user. Each assignment will have a positive action and a negative action to take. The sample json is shown below

```
[
   {
      “description”:...
      “wfassignmentid”:..
      “href”:”....”,
      “wfaction”:[
         {
            “instruction”:”....”,
            “Ispositive”:false
        },
         {
            “instruction”:”....”,
            “Ispositive”:true
        }
      ]
   },
   {
     …
   }
]
```

Note that the wfaction json contains the positive and negative actions. The user is supposed to take up one or the other.

The api call below shows how to take the positive action.

```
POST <href of the mxapiwfassignment>?action=wsmethod:completeAssignment
x-method-override: PATCH

{
  “memo”:”some memo”,
  “accepted”:true
}
```

To take up the negative route you can just set the accepted flag to false in the json and POST to the same href.

Handling Input Nodes

Input nodes provide the user with interactive options to choose from in a workflow path. The user may not choose anything, in which case the workflow stays in that same state. If the workflow framework looks ahead and sees an input node as the next node, the rest api response for the current node (say that was a task assignment that the user accepted or rejected) will return

1. A response json which will have the details of the options that the input node provides. The consuming client code is supposed to use those options to let the end user decide the option to chose.
2. A response location header with the url to POST the users choice to.

The response json may look like this

```
{
   “member”:[
   {
     “actionid”:..,
     “Instruction”:”.....”
   },
   {
     “actionid”:..,
     “Instruction”:”.....”
```

```
    }

    ]
    “nodetype”:”INPUT”,
    "internalnodetype": "WFINPUT"
}
```

Note that the input node type says that its “WFINPUT”. This information can be leveraged by the consuming code (say a mobile app) to display a generic UI to represent these options.

The api call below describes how to choose an option:

```
POST <location uri>

{
     “actionid”:”choose one of the action id from the json above”,
     “Memo”:”....”
}
```

Note if this call is not made, the workflow stays with the current node (ie the node previous to the input node) and does not move to the next node. In essence the input node is a transient node which is only available for processing within that context of the previous node.

Handling Interaction Nodes

Interaction nodes are Maximo UI dialogs and applications/tabs that are presented to the user for him/her to take an action using that UI artifact. Now unlike an input node this one is not a transient node. This implies that the workflow engine has moved to the this node from the previous node.

When the workflow lands into this node, the response json from the previous call should indicate that details of the interaction node, presenting the information from the WFINTERACTION table for that node. This should help identify (using the json property “internalnodetype” with a value of WFINTERACTION) the client code to provide an equivalent interface for the Maximo dialog/app for the user to act upon. Like the case with the input node, the rest framework will generate a URI (set the in the response location header) for the client code to respond back to the interaction such that the workflow instance can move to the next node in the path.

Now if the user ignores this node the engine just moves on to the next node. But the engine needs to know that the interaction node job is complete one way or the other. To do that the end user in Maximo application will re-route the workflow by pressing the workflow route button. To simulate that in the api realm, the client code needs to make the api call shown below

```
POST <location uri>

{
    “interactioncomplete”:1
}
```

This will indicate to the workflow engine that the interaction is complete.

Handling Wait Nodes

Wait nodes are listeners to the Mbo (that is being workflowed) event. Effectively the workflow waits on this event and when the event eventually happens, it moves to the next node. We do not need any special handling with apis for this node as this is backend event driven. So if an event comes from any api calls/ MIF call or UI call - for that mbo, the workflow will listen for that and if the condition is met, it will move to the next node in the path.

Handling Condition Nodes

Condition nodes are automatically evaluated by the workflow engine and the engine will move to the next node in the path after condition evaluation.

Permissions in Maximo

API Routes

Dynamic API Documentation (Swagger)

In Memory Processing Of Resources

Saved Queries

Maximo supports a feature called a Saved Query where a pre-built query for an application, such as Work Order Tracking, which allows a user to easily retrieve a common set of data (example: a list of Approved Work Order). After reading this section, you can use the saved query capability to query records based on defined filter criterion with RESTful API call.

Available Queries for Object Structure

For each Object Structure, you can find all authorized (for the requesting user) saved queries using the apimeta api call listed below (with MXASSET Object Structure as an example)

Talk about ispublic, name, title and href?

```
GET oslc/apimeta/mxasset
{
..
"queryCapability": [
      {
            "ispublic": true,
            "name": "All",
            "href": ".../oslc/os/mxasset"
      },
      {
            "ispublic": true,
            "name": "publicAssets",
            "javaMethod": true,
            "href": ".../oslc/os/mxasset?savedQuery=publicAssets"
      },
      {
            "title": "IT Stock in Stock Locations (non-Storeroom)",
            "ispublic": true,
            "name": "ITSTOCK",
            "href":".../oslc/os/mxasset?savedQuery=ITSTOCK"
      },
      {
            "title": "X",
            "ispublic": true,
            "name": "LINKED-ASSETS",
            "href": ".../oslc/os/mxasset?savedQuery=LINKEDASSETS"
      },
      {
            title: "Life to date cost is 80% of replacement cost",
            ispublic: true,
            name: "ASSET:Bad Actor - LTD Cost",
          href:
          "/oslc/os/mxapiasset?savedQuery=ASSET%3ABad+Actor+-+LTD+Co
          st"
     },
     ]
     ..
}
```

In Maximo, we have 4 types of saved queries for Object Structure

Query Method (method, java method)

This query is defined in Object Structure's definition class. It is sourced from an annotated method name.
This option would be used if a method was implemented for query purposes. There are no default query methods provided, this would be a custom code implementation.

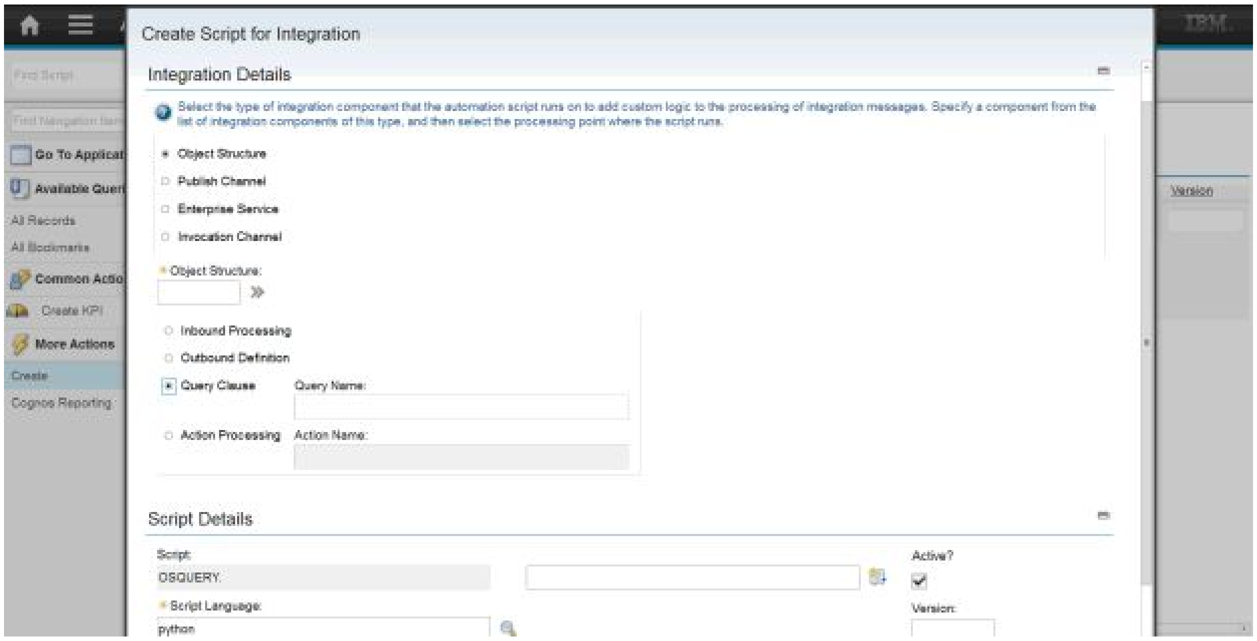
The code example is as following,

```
@PreparedQuery("http://maximo.nextgen.asset#publicAssets")
public void publicAssets(MboSet assetSet) throws MXException,
RemoteException
{
     String whereusercust="assetnum not in (select assetnum from
assetusercust)";
     assetSet.setUserWhere(whereusercust);
}
```

Automation Script (script)

This query is run with a predefined automation script. This configuration allows for more complex queries than are normally supported by a query clause.

The creation of a script for an object structure can be defined as a query clause. When you define a script as a query clause, the script can be configured as an object structure query for use with the JSON API.


Create Script for Integration

Integration Details

Select the type of integration component that the automation script runs on to add custom logic to the processing of integration messages. Specify a component from the list of integration components of this type, and then select the processing point where the script runs.

- [✓ Object Structure]
- Publish Channel
- Enterprise Service
- Invocation Channel

Object Structure:

- Inbound Processing
- Outbound Definition
- [✓ Query Clause] Query Name:
- Action Processing Action Name:

Script Details

Script: OSQUERY.

Script Language: python

Active? ✓

Version:

Object Structure Query Clause (osclause)

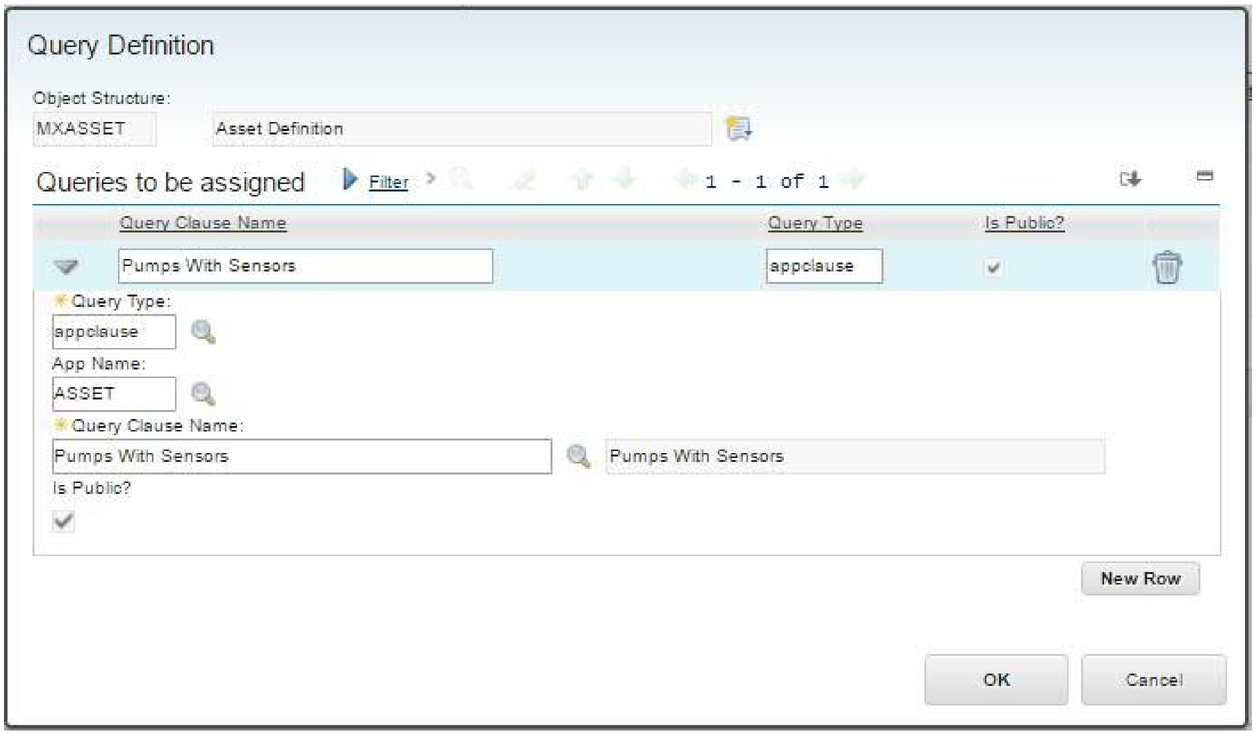
The where clause for this query is defined in this query definition. For this type, you enter a Where clause, provide a name and description for the query, and flag whether the query is public or not. The Where clause format is similar to a Where clause that is used in an application List tab. Public queries are available to everyone to use. Non-public queries are only available to the query owner.

Applications Query (appclause)

The query is sourced from a Public Saved Query of an application. It could be associated with Object Structure in following ways,

Using Asset and MXASSET as an example,

1. Open Object Structure APP. Select Query Definition from Action List, set type = appcluase, select the query from the list.


Query Definition

Object Structure: MXASSET Asset Definition

Queries to be assigned

Query Clause Name	Query Type	Is Public?
Pumps With Sensors	appclause	✓

Query Type: appclause

App Name: ASSET

Query Clause Name: Pumps With Sensors — Pumps With Sensors

Is Public? ✓

2. Set the authorization name of MXASSET as ASSET. In apimeta, the saved query names will be listed as original name.
3. Set the authorization name of MXASSET as MXASSET, then check load queries from all apps. If you have the access to ASSET, in apimeta, the saved query name will show as ASSET:QueryName

For the detail of OSLC Query, see Ability to set query definitions and action associations in the Object Structures application.

Execute Saved Query for Object Structure

In Maximo RESTful API, the query parameter for all of the saved query is **savedQuery.** Note, this parameter is case-sensitive. If you apply SAVEDQUERY or savedquery, it will be ignored as invalid query parameter without any error.

In the queryCapability Section of APIMeta, the links for saved queries are already provided. Take ITSTOCK as an example:

```
GET /oslc/os/mxasset?savedQuery=ITSTOCK
```

Execute KPI clause for Object Structure

In Maximo RESTful API, we could get more detail for KPI by calling its where clause with saved query as following,

```
GET /oslc/os/mxasset?savedQuery=KPI:ASSETKPI
```

API will take the where clause from KPI and apply it to MXASSET Object Structure.
Note, KPI clause will not be available in APIMETA. And, you have to make sure the where clause in KPI could be applied to the main object of the Object Structure. Otherwise you will get the SQL error.

Query Template

Query Template is an object structure-based template including the query related definition. In collection level, we can define the page size, search attributes and timeline attribute in the template. In attribute level, we could add selected attribute, give the ordered info, and override the title of attribute.
After reading this section, you will be able to create a Query Template for Object Structure (We will mainly take MXASSET as an example). Apply it to Object Structure and get selected attribute and ordered collection back.
Currently, the query template can be created by JSON API. Let's start with create one for MXASSET. Here is the definition, suppose we have an object structure named as MXAPIQUERYTEMPLATE for querytemplate:

Set up query template

Page Size = 5;
Search Attributes = assetnum, description;
Timeline Attribute = changedate;

```
POST /oslc/os/mxapiquerytemplate
{
      “pagesize”:”5”,
      “searchattributes”:”assetnum,description”,
      “timelineattribute”:”changedate”,
      “intobjectname”:”MXASSET”
}
```

Assume the query template name for this new created query template is 1001. Since we haven't defined any attribute yet, the result set will only contain the reference links for each of the record. When we apply the query template to object structure, in this case, MXASSET, we will use following query parameter to generate the restful call with query template, querytemplate=1001

```
GET /oslc/os/mxasset?querytemplate=100&collectioncount=1
```

Expected Response Info:

```
  "responseInfo": {
    "nextPage": {
      "href": "next page link"
     },
     "totalCount": 1152,
     "pagenum": 1,
     "href": "current page link",
     "totalPages": 231
  },
```

From result set, you will see there are only 5 records included in the first collection page. Since the searchAttribute and timeline attribute already been defined in query template. We could use oslc.searchTerm and tlrange to filter the result set.

GET
```
/oslc/os/mxasset?querytemplate=1001&oslc.searchTerm="PUMP"&tlrange=-3
M&collectioncount=1
```
Please refer to ***search attribute and search term, timeline section, collectioncount*** to see more detail about oslc.searchTerm, tlrange and collectioncount(what is the good way to do refer)

Expected Response Info:

```
  "responseInfo": {
    "nextPage": {
      "href": "next page link"
     },
    "totalCount": 43,
    "pagenum": 1,
    "href": "current page link",
    "totalPages": 9
  },
```

Set up query template with attributes

After you define the basic configuration for query template. You want to create several attributes for query template. The syntax for attribute is shown in following table (The examples are based on MXASSET):

Basic Format		
Format	Description	Example
attribute	The attribute name from the object	assetnum
relationship.attribute	The attribute name from dynamic relationship	allwo.wonum

```
POST /oslc/os/mxapiquerytemplate
{
      "pagesize": 5,
      "intobjectname": "MXASSET",
      "querytemplateattr": [{
                  "selectattrname": "assetnum",
                  "selectorder": 1
            },
            {
                  "selectattrname": "status",
                  "selectorder":2
            },
            {
                  "selectattrname": "siteid",
                  "Selectorder":3,
            "sortbyon":true,
                  "sortbyorder":0,
                  "ascending":true
            }]
}
```

Assume the templatename is 1002, after you apply the query template with the query. The result set should return in 5 records per page, each of the object will include the assetnum, status and siteid. Besides, the records will be sorted by Site in ascending order.

```
GET /oslc/os/mxasset?querytemplate=1002
```

Expected Result:

```
{
        "assetusercust_collectionref": "link to assetusercust",
        "assetnum": "1001",
        "_rowstamp": "1195406",
        "status_description": "Not Ready",
        "assetopskd_collectionref": "link to assetopskd",
        "assetmeter_collectionref": "link to assetmeter",
        "status": "NOT READY",
        "assetmntskd_collectionref": "link to assetmntskd",
        "siteid": "BEDFORD",
        "assetspec_collectionref": "link to assetspec",
        "href": "link to current record"
},
```

We also provide the capability to use the complex syntax, with * notation after the attribute.

Advanced Fromat (*)		
Format	Description	Example
rel$relationship.attribute*	The attribute name from dynamic relationship (1:n)	rel$allwo.wonum*
rel$relationship.exp$formula*	The formula for dynamic relationship (1:n)	rel$allwo.exp$formula*
rel$relationship.relationship.attribute*	The attribute name from multiple level relationship (1:n)	rel$allwo.pm.pmnum*
childobjectname.attribute*	The attribute name from child object (1:n)	location.location*
exp$formula*	The formula for the object	exp$formula

Suppose we are trying to build the following clause,
oslc.select=rel.allwo{wonum,siteid,exp.formula,pm{pmnum,description}},location{location,description,allwo.wonum},assetnum,allwo.wonum

The x.y.z* syntax is:

rel$allwo.wonum*	rel$allwo.pm.description*	assetnum
rel$allwo.siteid*	location.location*	allwo.wonum
rel$allwo.exp$formula*	location.descrption*	

rel$allwo.pm.pmnum*	location.allwo$wonum*	

Differences between basic and advanced format

The differences between relName.AttrName, objName.attrName* and rel$relName.attrName*, taking MXPO as an example:

Suppose we are trying to deal with this relationship: Name = VENDOR, Parent = PO, Child = COMPANIES

And we are trying to get the **name** for company, we have following notations:

vendor.name: There is no specific action for it. Usually it’s used when we don’t have companies as our child object in MXPO, it will only take the first record back even it could be one to many.

companies.name*: It will convert to companies{name} by Query Template. Usually it’s used when we **have COMPANIES** as our child object in MXPO (and relationship could be VENDOR), then we have to use objectname instead of relationship name to get the record back, and it will return multiple records if the result is one to many.

rel$vendor.name*: It will convert to rel.vendor{name} by Query Template. Usually it’s used when we **don’t have COMPANIES** as our child object in MXPO but we still want to get multiple records back if the result is one to many

Troubleshooting the REST api

The REST api leverages primarily the “integration” and “oslc” loggers for the api framework part. Enabling those 2 loggers to DEBUG or INFO will provide a good amount of debugging information. However, the rest apis will always interface with Maximo business objects and other maximo artifacts - security, scripting etc which will have their own loggers. On top of that, we will often need to enable the sql loggers whenever we feel the query result is not what the filter clause described.

One good way to debug this thing may be to use the thread logging functionality which is integrated with the rest api framework.

Thread logging is enabled on a per user basis within the rest api scope. This can be done from the maximo logging application->Configure Custom logging->Thread logging. Choose the context name as “OSLC” and the user name as the “personid” of the user whose REST requests we want to track/debug. Next we can enable the logging with say - sql,oslc and integration loggers to start with.

This same step of enabling the thread logging can also be done by the user himself with the help of the rest api.

```
POST /oslc/log/enablelogs

[“log4j.logger.maximo.sql”,”log4j.logger.maximo.oslc”,”log4j.logger.m
aximo.integration”]
```

This will enable the thread logging for the current logged in user for the loggers sql,oslc and integration. There is another api /oslc/log/enablealllogs that enables all logger for this user. We recommend not setting that one right away as it would generate a tonne of log which may make it hard to debug. We can disable this logging by making the call below:

```
POST  /oslc/log/disablealllogs

<no request body needed>
```

As we make the requests with this setup for the desired user, the system keeps track of all the oslc, integration and sql logs generated for that user only. It will not mix the logs with other users or other contexts (other than OSLC) that may also generate logs.

This log can then be accessed using a rest api call GET /oslc/log. This will stream the log to the browser. This log is only for the user who was targeted with thread logging setup and can be accessed by only that user and only for that logged in user session. Once the session is done, this log can still be accessed by the server admin from the server's working directory. That would be a manual process - no rest api for that.

Password Management using REST API

Note this feature is only supported when Maximo is configured to use Maximo native authentication scheme. If Maximo is configured to use the application server security (ie mxe.useAppServerSecurity property is set to 1), then the security provider (for example the LDAP user registries) are responsible for password management and hence their admin tools should be leveraged to complete this task.

Change Password

Change password needs to get done when the password expires for an user or the user deems necessary to change the password for some other security concerns. If the password has expired and the user attempted to login using the old password he/she will get an 403 error with the BMX id of BMXAA2283E. The client code can detect that error and then throw up the password change dialog for the end user to change the password. The api to change password is shown below. Note the maxauth header will have the expired password.

```
POST /changepassword
maxauth:<base64 encoded user:password>

{
  "passwordold":"blah-old",
  "passwordinput":"blah-new",
  "passwordcheck":"blah-new",
  "pwhintquestion":"<password hint question> (optional)",
  "pwhintanswer":"<password hint answer> (optional - goes with the
hint question)"
}
```

Forgot Password

This api is used to reset password in Maximo.This would be initiated by the end user when the end user has forgotten the password.

```
POST /forgotpassword

{
   "primaryemail":"the email id where you will get the reset
password",
   "pwhintquestion":"the hint question",
   "pwhintanswer":"the hint answer",
   "loginid":"the login id of the user"
}
```

This should send an email with the reset password. Note unlike the change password api, here you do not have to provide the maxauth header as in this case the user does not remember his password.

Supporting BIRT reports in the REST api (in 7609)

Supporting File Import (CSV/XML) and Import Preview using REST api (in 7609)

From 7609, we support uploading data from flat file or xml file to Maximo by restful API action=importfile, which leverages with Application Import capability in classic Maximo.

Prepare Object Structure

For import using the XML format, we do not need any special configuration to the Object Structure. For Flat file support, similar with the Application Import in Maximo, we need to configure Object Structure to support flat structure. This implies there shouldn't be any alias conflict for that specific Object Structure.
To verify that, find your target Object Structure, like MXAPIMETER in Object Structure Application. Make sure Support Flat Structure is checked and no alias conflict is detected by the system (this should be indicated by Alias Conflict checkbox is unchecked).
If there are any conflicted fields, find Add/Modify Alias from action list. Add new alias to conflicted field starting from first child object.
The rest of this section will focus on flat files. XML files would be exactly similar with a different header value.

Security

There is no special requirement on security - it just follows the normal Object structure security concepts. This implies that you got to have support for INSERT/SAVE/DELETE sigoptions (for your corresponding security application for the Object structure) to be able to import csv files.

Prepare CSV

The CSV format is exactly the same as what you would use in maximo classical application import. This implies when we try to import file from CSV, the attribute name of each field must match with attribute alias in Maximo.

For example, the csv file content can be as following

METERNAME,METERTYPE,READINGTYPE,DESCRIPTION,MEASUREUNITID
RUNHOURS2,CONTINUOUS,DELTA,Run Hours2,HOURS
TEMP-F2,GAUGE,,Temperature in Fahrenheit2,DEG F

Preview

Before processing the data into the database, it is always better to check if there is any error by using preview functionality. It is supported in import file API. Here is the sample:

```
POST oslc/os/mxapimeter?action=importfile&lean=1
maxauth:<base64 encoded user:password>
preview:1

METERNAME,METERTYPE,READINGTYPE,DESCRIPTION,MEASUREUNITID
RUNHOURS2,CONTINUOUS,DELTA,Run Hours2,HOURS
TEMP-F2,GAUGE,,Temperature in Fahrenheit2,DEG F
```

After we make the POST request to server, maximo will return the preview response which is including the validation information and warning messages.

Here is the sample with error message:

```
{
    "invaliddoc": 2,// how many invalid records
    "totaldoc": 2,//how many records in csv file in total
    "validdoc": 0,//how many valid records
    "warningmsg": "\nBMXAA5598E - Processing of an inbound
transaction failed. The processing exception is identified in
document 1.\n\tBMXAA0024E - The action ADD is not allowed on object
METER. Verify the business rules for the object and define the
appropriate action for the object.\nBMXAA5598E - Processing of an
inbound transaction failed. The processing exception is identified in
document 2.\n\tBMXAA0024E - The action ADD is not allowed on object
METER. Verify the business rules for the object and define the
appropriate action for the object."//the error messages for each
record.
}
```

We can tell what is the problem from warning messages then fix it. In the sample error responses, we can tell the issue is caused by missing the security setup. After granting the sigoptions and reprocess the call, the successful preview will look like following,

```
{
      "invaliddoc": 0,
      "totaldoc": 2,
      "validdoc": 2,
      "warningmsg": ""
}
```

File Import

After preview, we can import the file to maximo by removing preview header from request:

```
POST oslc/os/mxapimeter?action=importfile&lean=1

METERNAME,METERTYPE,READINGTYPE,DESCRIPTION,MEASUREUNITID
RUNHOURS2,CONTINUOUS,DELTA,Run Hours2,HOURS
TEMP-F2,GAUGE,,Temperature in Fahrenheit2,DEG F
```

And we can get response as following:

```
{
    "validdoc": 2
}
```

We also support other file type like XML, customized delimiter and textqualifier. You can easily configure them with the following headers when you do the POST call.

Header	Description	Default value
filetype	The type of the uploading file, it can be FLAT or XML	FLAT
delimiter	The delimiter of csv file	,
textqualifier	When the data include any delimiter, it will be wrapped by textqualifier	"
preview	If the importfile API is running in preview mode. It can be 0 or 1	0

Supporting File Export

File exporting can be done by using query parameter _format=csv. We can download data from any Object Structure, for example, MXAPIASSET in Maximo to a flat file. Similar with File Import, before exporting data, support flat file, no alias conflict and security set up are required for the target Object Structure.

File Export

_format=csv is always used with other query parameters together for different tasks.
Here are some common ones.

oslc.select: it can be used to control which columns are included in the csv file. For example, if we are trying to export data from ASSET with assetnum, siteid, description and location field, we can add these four field to oslc.select as oslc.select=assetnum,siteid,description, location.
oslc.pageSize: it is used to to control how many records we are going to download from server.
oslc.orderBy: it can be used to define the orderby column in csv file.
oslc.where: it can be used to filtering the data.

Most of querying or filtering capability are available for file exporting. Please refer to other chapter due to the requirement.

Here is the sample rest call:

```
GET
/oslc/os/mxapiasset?lean=1&oslc.select=assetnum,siteid,description,lo
cation&oslc.pageSize=10&oslc.orderBy=-assetnum
```

By this restful call, we are going to download data from MXAPIASSET which including assetnum, siteid, description and location. The maximum records number is 10 and the result will be sorted by assetnum.

Creating a Maximo User using the REST api

Creating a maximo user can be done using the MXPERUSER (or MXPERAPIUSER) Object Structure. This will create the maximo user and person at the same time. The rest call is shown below:

POST /oslc/os/mxperuser

```
{
  "personid":"TESTADMIN1",
  "firstname":"ABC",
  "lastname":"XYZ",
  "primaryemail":"abc_xyz@yahoo.com",
  "primaryphone":"999 999 9999",
  "city":"Boston",
  "addressline1":"crazy road",
  "stateprovince":"MA",
  "postalcode":"01111",
  "country":"US",
  "language":"EN",
  "maxuser":[
  {
          "loginid":"testadmin1",
          "passwordcheck":"Helloabc11",
          "passwordinput":"Helloabc11",
          "defsite":"BEDFORD",
          "type":"TYPE 1",
          "userid":"TESTADMIN1"

  }
  ]
}
```

Note that you have to set the passwordinput and passwordcheck to be not restricted in the Object Structure as they are set to restricted at the object level by default. Also this is an example of creating a user when Maximo is the owner of authentication. If authentication is handled by the application server, we do not need to provide the passwordinput and passwordcheck attributes (or any other password management details like emailpswd, generatepswd, password hint question, force expiration etc) as Maximo does not manage passwords when mxe.useAppServerSecurity is set to 1.

Creating a Maximo MT (Multi tenant) Tenant using the REST api

We need to create an object structure (say MXAPITENANTREG) with the tenantdbuserid attribute restricted. We also need to include the newusergroup and the docroot non-persistent attributes.

The sample below shows a POST request to create a tenant:

```
POST /oslc/os/mxapitenantreg

{
  "tenantcode":"MYTEST00",
  "description":"my test 00 tenant",
  "company":"test00comp",
  "firstname":"mytest00",
  "lastname":"Bhat",
  "primaryemail":abc@us.ibm.com",
  "tenantloginid":"myabc123",
  "tenantdbuserid":"T11",
  "tenantlangcode":"EN",
  "status":"ACTIVE"
}
```

This will create the tenant and the tenant admin with the loginid of myabc123. It will also send an email to the primary email address with the generated password. We need to make sure that the smtp host is setup for this. If the app server security is turned on, then this smtp setup maynot be needed as the password management is done outside of Maximo. Note a MT tenant cannot be created without the tenant admin. Also note that this api needs to get invoked in the context of MT landlord.

Handling Interactive Logic in Maximo Using the REST APIs

Maximo business logic is filled with Yes/No/Cancell/OK interactions that needs specific user inputs to execute the corresponding business logic. For this logic to be accessible from REST apis, we need to make the rest api request interactive. By default all rest requests are non interactive. This will make the server side logic choose the default option and execute the default logic. This may not be desirable in all cases. To keep the choice at the users hand we have introduced the concept of interactive requests in this api. The example below shows how to make that request:

```
POST /oslc/os/mxapiwodetail...?interactive=1
```

This will mark that request as interactive and hence now execute the interactive logic on the server side. However you need to somehow set the desired user input for the interactive logic. To do that we need to set the request header **yncuserinput.**

the value can be a ; separated list of name value pairs - each name will correspond to the YesNo key - for example in the FldWoAssetnum class one of the interactions is shown below

Say u were dealing with this one:
MXApplicationYesNoCancelException.getUserInput("woassetchange",...

your request header would look like

```
yncuserinput: woassetchange:<value>
```

where the value is one of

OK = 2
CANCEL = 4
YES = 8
NO = 16
NULL = -1

An example shown below:

yncuserinput: woassetchange:8

Now say you had YNC nested - like one YNC leads to the other - you can solve all of them by providing the values in sequence - like:

yncuserinput: woassetchange:8;<someotherkey>:<someothervalue>

Handling duplicate requests in REST api

Often we will land up into a situation where the REST call committed successfully on the server, but the communication channel broke and the client got a 500 error. The client will think that the server rolled back the transaction and will end up re-sending the request. This in some cases can result in erroneous or duplicate data. To avoid this the REST api framework provides a mechanism to catch this double-dipping issue. The request for create/update/delete can contain a request header called **transactionid** which the api framework will validate for duplication. If it finds no matches, the transaction is good to go. If however a match is found, the request is rejected with a HTTP 409 Conflict error.

Note that the transactionid header value is client generated and hence is the responsibility of the client code to make sure it is unique enough that is does not clash with another valid request. If the server does not find a match, it stores it as part of the request transaction commit so that in can reject future transactions with the same transaction id. The default life of the transaction id is 5 minutes, controlled by the escalation OSLCTXN. However this can be modified as per the installation need.

Note that this feature is primarily useful when we are operating the REST client in an asynchronous or disconnected mode (much like the Anywhere platform). This feature may not make much sense for in the connected/interactive mode.