Table of Contents

---

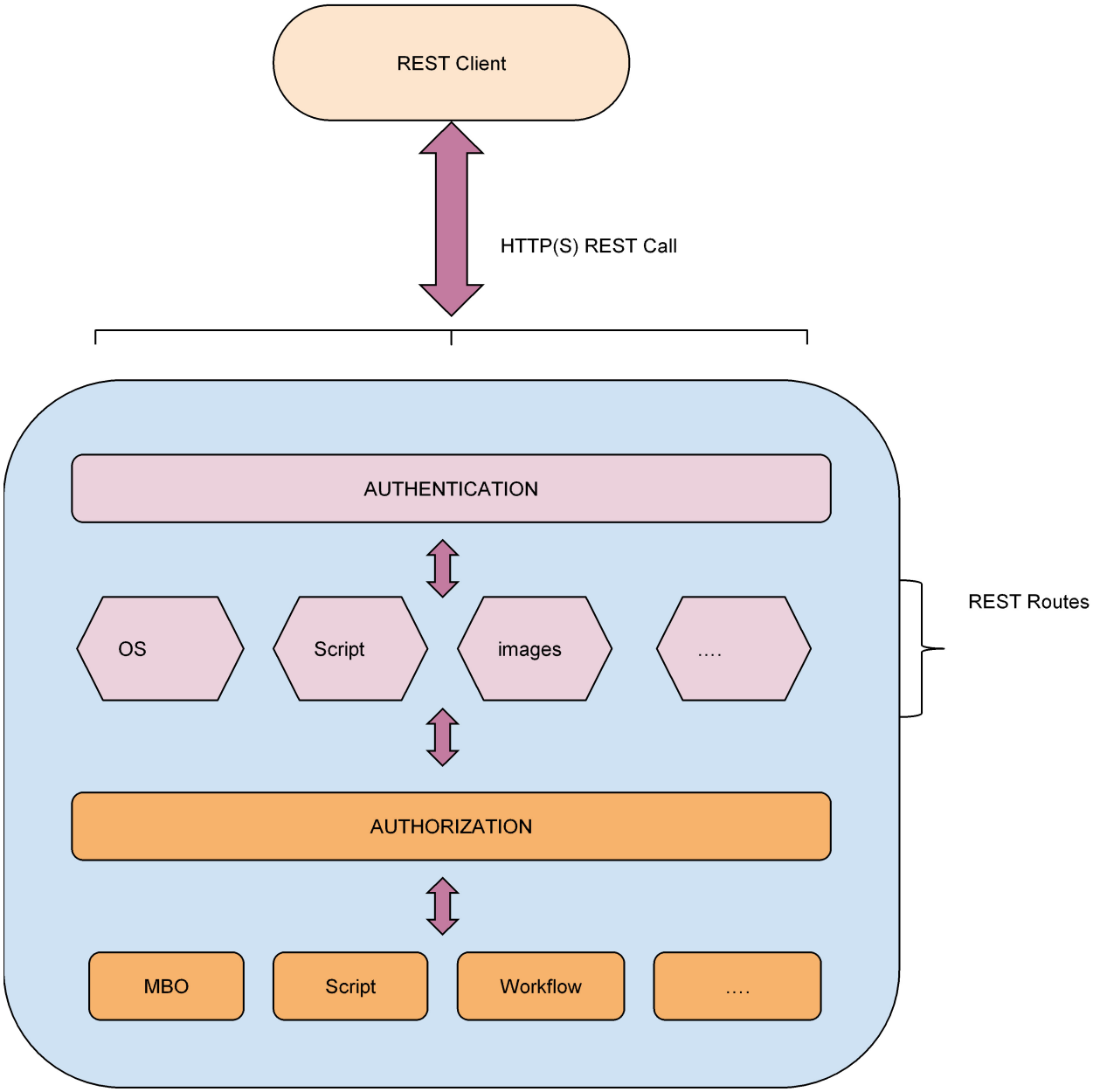
1. Overview

The new REST APIs for IBM® Maximo® Asset Management are a rewrite of the existing REST APIs that were released after Maximo Asset Management version 7.1, and are integrated in Maximo Asset Management releases starting with version 7.6.0.2. The new REST APIs are sometimes referred to as the REST/JSON APIs because of the end-to-end support for JSON data format. The benefits of using the new REST APIs include:

- 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 and so on.
- Tight integration to automation scripts - query, actions, custom APIs.
- Enhanced REST support for Maximo Integration Framework (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 that are appended to the object structure JSON.
- Integration with the Maximo Asset Management cache framework.
- Integration with Maximo Asset Management formulas.
- Integration with federated Maximo business objects (MBOs).

The new REST APIs uses the same code base as the OSLC REST APIs, which are used by the IBM Maximo Anywhere platform. The new REST APIs are simpler to set up because you don’t need to set up OSLC resources and sheds the namespaced JSON that the OSLC APIs require. Effectively, the new REST APIs are ready to be used when a Maximo package that is not customized is installed and setup with users and groups.

A high-level overview of the architecture is illustrated in the following figure:



The REST API call flows through the authentication phase, API routes, authorization, and then it interacts the with the Maximo artifacts, such as MBO's, Workflows, and automation scripts. API routes are RESTlets (REST handlers) that provide the APIs for interfacing with various Maximo artifacts, such as MBOs, Automation scripts, Images, Permissions, and Schemas. This is more for the developers of the API to organically expand the footprint of the REST APIs to cover more parts of Maximo Asset Management.

Note: Try out the APIs as you read through this document. You can install the JSON viewer plug-in for Firefox or Chrome to easily view the JSON documents from the API response. You can also use a tool, such as Chrome Postman, to make the POST calls with the API.

This document also assumes that you use the lean JSON format (JSON with no namespace) by setting the lean=1 query parameter at login. Note: It is recommended to set the lean=1 query parameter for all requests so that the request can 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 is omitted in most of the examples, but it is recommended that you add that while making the requests.

This document also assumes familiarity with Maximo object structures, which form the resources for the REST APIs. For more information about the new REST APIs, you can view the REST API Quick Start (https://www.youtube.com/watch?v=S81i0NQg3C8&feature=youtu.be) video.

2. Authentication

This section describes the most common forms of authentication that is used in deployments for Maximo Asset Management and how to use the REST APIs for those authentication schemes.

2.1. Maximo Asset Management native authentication

With native authentication, Maximo Asset Management is configured to manage the user repository along with the user credentials. Maximo Asset Management 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. The following example shows a sample request:

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

<no body required>
```

2.2. LDAP based authentications

With LDAP based authentication, authentication for Maximo Asset Management is managed and validated by the application server.

2.2.1. BASIC authentication

With BASIC authentication, authentication credentials in the application server are presented in the following format:

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

<no body required>
```

2.2.2. FORM authentication

With FORM authentication, authentication credentials are presented in the following format:

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

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

Since this FORM request is a form encoded POST, the user ID and password values need 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 reused for the subsequent API calls.

It is recommended for all authentication schemes that the authenticated session is reused 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 reuse the session and does not need to reauthenticate 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 (https://github.com/ibmmaximorestjsonapis/maximorestclient/blob/master/src/com/ibm/maximo/oslc/MaximoConnector.java) code.

3. API home (/oslc)

The API root URL is /**oslc**. A GET call on the API root fetches a JSON object with necessary links to get the details of the current Maximo runtime environment. Explore the links like **systeminfo**, **whoami**, **installedProducts**, **serverMembers** APIs along with details like the current date time, language, and the calendar of the deployed Maximo system. The following list describe these API links:

- **systeminfo:** The `GET /oslc/systeminfo` API provides the system information JSON for the deployed Maximo instance.
- **Whoami:** The `GET /oslc/whoami` API provides the profile JSON for the user that is logged in.
- **installedProducts:** The `GET /oslc/products` API provides the list of installed add-ons for Maximo Asset Management.
- **serverMembers:** The `GET /oslc/members` 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 environment 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 resulting JSON will have links to drill down into individual servers and examine the various aspects of the Maximo runtime server health, such as memory, MBO count, integration caches, threads, and database connections.
- **License:** The `GET /oslc/license` API provides the list of available license keys for the various Maximo components in this deployment.
- **Apimeta:** The `GET /oslc/apimeta` API provides the metadata for all the object structures that are API eligible, that is the usewith value of reporting, integration and oslc.

The metadata for each object structure includes the schema information as well as the available queries (saved queries). Additionally, the metadata provides the creation API URL and the (security) authorization application name.

4. Querying Maximo Asset Management by using the REST API

When designing the the REST API framework, the following functionality was considered:

- Being able to filter and sort Maximo business objects by using a higher-level query language that internally maps to the native SQL for the corresponding relation DB that is used by the Maximo deployment.
- Being able to select the list of attributes that you want the API to fetch.
- Being able to fetch data from related objects leveraging existing Maximo relationships without additional configuration.
- Being able to page data.

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 for any resources that are API enabled. All collection URIs support a known set of URI query parameters that operates on the collection. The following most common query parameters are listed

oslc.select: This query parameter specifies the set of attributes to fetch from the object structures as well as the related objects.

oslc.where: This query parameter specifies the WHERE clause for filtering the result set of a query.

oslc.orderBy: This query parameter specifies how the results of a query are ordered.

oslc.pageSize: This query parameter specifies the resources for each page of a query.

4.1. Select Clause example

The select clause example is based on the `MXASSET` object structure.

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

The following response (collection resource) is an example of what you might see from the request:

JSON

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

This results in fetching the 10 members (max) of the `MXASSET ` resource, which is based on the `ASSET` MBO, and 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 follows the 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` that contains the meta information about the query. The meta information includes the current URI that is used to get the result (href) and the URL for the next page (`nextPage`), if there is a next page. The meta information also includes the URL for the previous page (`previousPage`), if there is a previous page. 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`) that are available for this query are also included.

The `totalCount` and `totalPages` are not displayed by default. You can enable `totalCount` and `totalPages` by including the query parameter `collectioncount=1` to the request. If you want to see only the total count of records that match the query and not the records, you use the request query parameter `count=1`. This results in the following JSON response:

JSON

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

If you set the `mxe.oslc.collectioncount` system property to `1`, the `totalCount` and `totalPages` parameters are included by default as part of the `responseInfo`. However, we recommend not to set that property to `1` because 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). It is preferred to just request them using the query parameter `collectioncount` as needed.

Just getting the links to the member resources may not be very exciting or useful. Rather than traversing individual URIs for details, the `oslc.select` clause is used to get more details inlined in this JSON response.

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

The following JSON will look like

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

This results in a JSON that contains the four attributes as requested for each of the members. Note that by default the API response skips the null value attributes. For example, if the location is null for any of the member assets in the selection, that attribute will not appear in the member JSON and 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 that is based on the users profile language. The API framework detects a domain bound attribute (from the Maximo metadata repository) and uses the domain cache to fetch the description for that status.

The `_rowstamp` property is present for every object in the object structure for a given resource record and is used for handling dirty updates. See Creating and updating resources.

The `xxxx_collectionref` properties includes the links to child objects as defined in the `MXASSET` object structure. The prefix `xxxx` is the name of the child object. The `GET <collectionref link>`) shows 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.

To get data from the `assetmeter` object where `assetmeter` is a child object as defined in the object structure `MXASSET` , you can use the following select clause:

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

The member JSON looks like:

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

To get the child object details, the notation - <child object name>{comma separated attribute names or **to get all properties} is used. So** `assetmeter{ }` is going to fetch all properties for the `assetmeter` . This notation is applicable for multiple levels. For example, you can define ` 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 need to get more data from related objects, such as locations or work orders, which are not defined in the object structure. The following notation is an example of calling more data from related objects:

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

The member JSON looks like:

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

Note the property `$alias_this_attr$location` . For more information, see the section Aliasing attributes.

Asset to location is a 1:1 relationship where the dot notation attaches the JSON object for the location with the member JSON for `MXASSET` at the asset header object. Note that the API framework detects 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 groups attributes at each relation level to form the JSON object. For example:

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

Results in a JSON like:

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

Dot notations produce JSON objects. However, for related data that is `1:*` , a variation is required. For example, for an `Asset:Workorder` relationship, an asset can have many open work orders. If you want to get details about all open work orders for the set of Assets, you can use the following select clause:

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

This `rel` notation has the format - `rel.<relation name>` and results in a JSON array property named `relation name` . The following example shows the sample output format:

```JSON
{
    “openwo”:[
     {
         “wonum”:..
     }
     ....
     ]
}
```

This rel notation can also be nested. The nesting occurs as part of it’s attribute set, for example:

```
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` shows that you can embed a dot notation within a rel notation but not the other way round. The `rel3` also demonstrates that you can use **to get all attributes for that target object. To determine whether** implies all persistent attributes or all persistent and non-persistent attributes combined, use the following guidelines:

- If the target object is a persistent object, the **notation includes all persistent attributes for that object. You need to explicitly request the non-persistent attributes to include them. 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.

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.attr3}
```

To learn more about dynamic select clause, see the video Dynamic Data Retreival (https://www.youtube.com/watch?v=DXePvOSNhyc&feature=youtu.be).

4.1.1. 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 Asset Management, an image repository (imglib table) stores the image avatars for the managed resources, such as assets, items, person. The API framework maintains a cache of the image references. If the system detects an image reference while fetching the resource details, the URI for the image document is added to the resulting JSON (`_imagelibref`). The Maximo image repository stores the images in the Maximo database or in an external repository provided that the repository exposes a simple URI based mechanism to load the images. To facilitate that, the `IMGLIB` table has two attributes - `imguri` and `endpointname`. The `endpointname` attribute points to the integration endpoint, which is the `http(s)` endpoint, and the `imguri` attribute refers to the URL of the image that is used by the http endpoint to fetch the image. It is possible to use a custom endpoint to handle more complex URLs. Bulk loading of images can be done by using SQL command line tools. Associating images to any Maximo MBOs can be done by using REST APIs. The following sample REST API associates an asset with an image.

```HTTP
POST /oslc/os/mxasset/{id}?action=system:addimage
custom-encoding: base
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

```
HTTP
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”:..
}
```

The following API deletes the associated image:

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

Domain internal values

Maximo REST API supports fetching the synonymdomain internal value corresponding to the external value which will be in the payload by default as part of the mbo attribute value. Most use cases need the internal values for some client side logic as the external values are customer specific and the internal values are base provided. To get the internal values use the query parameter `internalvalues` with the value set to 1. The internal value would be set using the <attribute name>_maxvalue property. An example is shown below

```
HTTP
GET /oslc/os/mxapiasset?oslc.select=assetnum,status&internalvalues=1

{
   "members":[
    {
        "status":"...",
        "status_maxvalue":"..."
    },
    ...
}
```

Database aggregation functions

Maximo REST API supports using database aggregation (`max` , `min` , `avg` , `sum` , `count` , and `exists`) functions on related MboSets. For example, to apply these functions on the open work orders for an asset, all the aggregation functions are used in the following API:

```
HTTP
GET /oslc/os/mxasset/{restid}?
oslc.select=assetnum,openwo.actlabhrs._dbavg,openwo.actlabhrs._dbsum,openwo.actlabhrs._dbmax,openwo.actlabhrs.
_dbmin,openwo._dbcount
```

The format for the `sum` , `avg` , `max` , and `min` functions is `<relation name>.<target attrname>.`*`<operation>`*. *Note that the supported operations are* `dbsum` *(for* `sum` *),* `dbavg` *(for* `avg` *),* `dbmax` *(for* `max` *) and* `dbmin` *(for* `min` *). The format is always a dot (.) separated by a three token format that includes a relationship name token that is followed by an attribute name and the underscore prefixed ()* operation to perform on that related attribute.

The `count` function (operation `dbcount`) has a two token format that includes the relation name as the first token followed by the operation name (`_dbcount`) and evaluates the count on that related MboSet. The JSON response looks like:

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

Bookmarks

Maximo bookmarks can be leveraged with the rest APIs.

Formula properties

You can select calculated values without creating non-persistent attributes by using the integration between the REST APIs and the object formula feature, which was included in Maximo Asset Management 7.6.0.5.

For example, you create a object formula called `MYREPLACECOST` for asset object by using the **Object Formula** action in the **Database Configuration** application. The formula is `purchaseprice/NVL(priority,1)`. You can then select that formula property that is associated with the asset object by using the following API select clause:

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

The response is as if `myreplacecost` was an attribute of the asset MBO.

```JSON
{
     “member”:
      {
          “assetnum”:..
          “ myreplacecost”:
      },
      ....
}
```

A similar approach applies to an individual resource:

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

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

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 that does not need db config or admin mode.

For more information about the formula feature, see the [Maximo Formula community] (https://www.ibm.com/developerworks/community/groups/service/html/communityoverview?communityUuid=ad185f90-ca8d-4416-99a6-22fe7f50e4d1).

Federated resource data

4.1.2. 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, this clash is resolved by using namespaces. With JSON, you can rename the property with an alias. Aliasing refers to the process of renaming a MBO attribute in the JSON domain to avoid naming conflicts. Where there is a naming conflict, the JSON response marks the renamed attribute with the prefix `$alias_this_attr$`. To alias an attribute, you use the `--` operator in the select clause. The following renames the `location` attribute to `mylocation` in the JSON domain:

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

Note that the `--` operator works only on attributes and not on object names or relation names. Therefore, if an attribute name clashes with an object name or a relation name, the attribute name needs to be aliased.

4.1.3. Traversing to related MboSets

Traversing to related MboSets is achieved by using the relation name as part of the `GET` URI call. The following example shows how to move from an asset to a work order using the relationship name.

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

The `openwo` relationship name is used to traverse to the work order collection from a given asset. The resulting JSON is a serialized response that is based on the work order MBO, which by default does not contain any non-persistent attributes. You request non-persistent attributes explicitly in the `oslc.select ` clause.

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

If you prefer getting the response as a object structure collection resource, you can use the following clause:

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

The object structure name is added at the end of the relation name with a dot separator. With this request, you get all work order records returned as mxwodetail records. The following example is a variation of the API:

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

In both cases, (that is MBO and object structure), you can use all the collection API query parameters like `oslc.select` and `oslc.where` and use paging etc to filter, sort, and view the collection as required. Note that this is recursive and you can go as deep nested as needed by using the relation name and the rest ID pair. For example:

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

These rest IDs are derived from the URIs that come back from the server. The client code does not need to generate these IDs but uses 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.

5. Filtering data using WHERE clause

The most common way to filter a resource set is to use the `oslc.where` query parameter. This parameter internally maps to the Maximo QBE framework. You can filter data that is based on all persistent attributes, at the main MBO (for the OS) or any related MBO. In the example, the `where clause` filters assets that are based on locations and the asset status.

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

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 nested. The leaf element is always an attribute in the target Mbo.

The following table describes the different operators that you can use for filtering data:

Table 1. Operators

Operator	Description	Usage
=	equals	status=”APPR”
>=	Greater than equals	priority>=
>	Greater than	startdate>”iso date”
<	Less than	startdate<”iso date”
⇐	Less than equals	linecost⇐200.
!=	Not equals	priority!=
in	In clause	location in [“A”,”B”,”C”], priority in [1,2,3]

This query language is data type sensitive and the double quotes `""` is used for character based attributes and for dates (ISO Format). Numeric values are represented in their corresponding ISO formats (that is 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
```

You can use the in clause with numeric values too.

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

For the LIKE clause, you can use the following variations:

```
oslc.where=status=”%APPR%”
```

If you want to use starts with:

```
oslc.where=status=”APPR%”
```

If you want to use ends with:

```
oslc.where=status=”%APPR”
```

To use null value queries, you can use the star (*) notation. For example, a not null check is completed by using the following format where - `status is not null` .

```
oslc.where=status=”*”
```

If you want to use the is null check:

```
oslc.where=status!="*"
```

If you want to ise a not in clause, you can use the following format:

```
location!= "[BR300,BR400]"
```

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

5.1. Range filters

Range filters are used to support range based queries. For example, if you want to only get assets within a certain date range and priority range, you 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"
```

You can also filter with date ranges by using timeline queries.

5.2. SynonymDomain internal filters

You can also filter by using the internal values for synonymdomains. The following example shows how you can filter the work order objects that are 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 work order collection using the *not in* clause for the set of external values that correspond 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
```

There are a few things to note:

- 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 an 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.

5.3. MaxTableDomain based filters

You can also filter using the `maxtabledomain` list WHERE clause. The `maxtabledomain` is more useful for backward compatibility reasons with the older rest API and also in case a client wants to use an existing list WHERE clause that was created by using the table-domain. The API syntax is shown:

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

This API picks up the domain's `listwhere` and applies that to the collection. The site and organization are optional and needed if the `maxtabledomain` is site/org scoped.

5.4. Timeline filters

The timeline filters allow a simpler way to filter collections with time range based queries. The following example shows how you can list all work orders reported in the past three months:

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

This query finds all the work orders with a reportdate between today and the previous three months. The query is by default indexed around the current date. Another variation of this query is to range on a future date 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 that can be in the future. You might also use the current date as an index and filter around that date using the +- notation as shown in the following example:

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

This query filters records from three years in the past and three years in the future that is indexed on the current date. If you do not want to index on the current date, you can specify the date that you want to use for the index. The following example shows how you can specify the date:

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

6. Classification attribute search

Maximo objects, such as assets, locations, items, and work orders can be associated with classifications to provides a way to associate classification metadata (aka `classification attributes`) to the objects. These objects (assets,locations etc) are searched for based on those attribute values. This search capability is known as the attribute search feature. The REST APIs support for this feature was included in Maximo Asset Management 7.6.0.6. The following example shows an attributesearch that is applied to the asset MBO by using the MXAPIASSET object structure.

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

All assets that have a class spec attribute called `SPEED` with a value `>=50` are searched for. If the value :>=50 is not defined, then assets that have `SPEED` as a specific attribute is returned.

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

To combine multiple attributes in the search, you can use the following example:

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

This query searches for assets that have a class spec attribute `SPEED` with a value that is greater than 50, an attribute named AREA, and attribute name ELEV with a value of 300, all ANDed together.

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.

7. Sorting

You can sort collection resources by using the 'oslc.orderBy' query parameter in the following format:

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

The attributes that are prefixed with the minus sign (-) are sorted in descending order and the attributes with the plus (+) sign are sorted in ascending order. All attributes that are listed must be prefixed with a + or - sign. There is no default sort order. Additionally, only persistent attributes from the main object are supported. Related attributes are not supported.

The `+` sign needs to be URL encoded. This is a common mistake made by developers when trying this one out.

8. Paging

You can apply paging data by using the `oslc.pageSize` query parameter. The parameter value specifies the maximum number of 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.

8.1. Auto-paging

Auto-initiated paging was added to Maximo Asset Management 7.6.0.8. This feature is primarily needed if requesting a large data set, which can cause resource starvation or OOM errors in the Maximo server. To prevent these issues, you can set the auto-paging threshold in the Object Structure application. For example, if auto-paging is set to 1000 for the MXASSET object structure, and you request `GET /oslc/os/mxasset` and not define the `oslc.pageSize`, the system starts paging the request if the asset count exceeds the 1000 limit.

8.2. Limit paging

It is also possible that while querying a client set the page size to be too high, which can also cause an OOM error. To handle this scenario, you can set a positive value for the page size to the `mxe.oslc.maxpagesize` property. This applies to all Maximo object structures. You can also set the property to a specific object structure by setting the `mxe.oslc.<os name in lower case>.maxpagesize` property. This value change overrides the global value set by the `mxe.oslc.maxpagesize` property. If the request page size exceeds this value, an error occurs and the request fails.

8.3. Stable paging

Stable paging is a variation of the basic paging. In stable paging the MboSet in memory is loaded (note the MboSet contains 0 MBOs at this point as the MBOs are loaded on demand), and a reference to the MboSet is retained throughout the paging process. As the paging for the MboSet is processed, the MBOs are discarded. Effectively, at any given point in the paging, only one MBO is live. Note, the basic paging also discards the MBOs as they are serialized and there would never be more than one MBO in memory for the life of the paging. However, in basic paging the MboSet is loaded for every page request, that is the SQL query is applied every time. With stable paging, the MboSet reference is retained and does not need to process the SQL for every page but the pages expire as they are delivered because the MBOs are discarded and the MboSet is never refreshed. The following API example initiates stable paging:

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

This API creates a stable ID, which is embedded as part of the `nextPage` URI (`stableId=<some id>`). The URI is used to page forward. The same page cannot be reloaded because the page expires. Also, there is no backward paging because the pages including the current page also expire.

The stored MboSet expires if the set is not accessed after five minutes or if the set is paged to the end. You can changed the idle expiry time of five minutes by updating the `mxe.oslc.idleexiry` value, which is measured in seconds.

For more information about stable paging, you can view the video Stable Paging and Search Terms (https://www.youtube.com/watch?v=NQAV8YyWZLE&feature=youtu.be)

9. Filtering child objects

This section explains how to limit, filter, or sort child data sets while being inlined within their parent data set. All the query parameters for the child object collection operations follow a naming pattern that is `<mbo name>.<relation name>.<operator>`. The MBO name token refers to the parent MBO name for the child MboSet. The relation name is the relation name from the parent MBO to the child MboSet. The operator determines the type of inline operation that you are trying to apply on the child collection.

The following are the types of query parameters that are available:

9.1. Where filter

The format of this query parameter is `<mbo_name>.<relation_name>.where`. This query sets the WHERE clause filter for the child object that is identified by the *mbo_name* parent MBO and the *relation_name* relation name. The syntax for the WHERE clause follows the `oslc.where` format with the target MboSet in context. For example, to filter on the `polines` in the MXPO object structure, use the follow format:

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

The poline's that have an itemnum and line cost that is greater than 100 is shown and does not impact the selection of the PO records.

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

9.2. Limit filter

The format for this query parameter is `<mbo_name>.<relation_name>.limit`. This query sets the limit to the number of rows to be retrieved for the child collection.

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

The results show the limit the number of polines loaded to only 1 for each purchase order in the response.

9.3. Sorting

The format of this query parameter is `<mbo_name>.<relation_name>.orderBy`. This query sorts the child collection, 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,linecost}&po.poline.orderBy=-
linecost
```

The results shows a descending sort list on the linecost for the poline collection.

9.4. 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.

These query child object filters examines and limits the child objects in the inlined with the root object of the object structure. If you need to show those child objects in a list or tabular form and then page through and filter them, then you can use the relation name in the URI to traverse to that set. For more information, see the section Traversing to related MboSets. By traversing to the set, you can operate on the child collection independently and be able to page the collection.

10. JSON Schema

The rest APIs describe the resource using JSON schema standards. Maximo metadata contains more information than JSON schema supports. Therefore, the schema specification is extended with Maximo specific properties that contains more information from the Maximo metadata. Schema's can be accessed in couple of ways using the REST APIs. A `jsonschemas` route provides schema to any object structure. The following example shows that:

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

The JSON schema for the root object of the object structure is returned and contains links to the child objects, such as `INVRESERVE`. To get the schema for all the objects in the OS, you need to include the request parameter `oslc.select=*`. This fetches 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 specify the `oslc.select` clause to filter out the part of the object structure that you need. An example is shown below:

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

Details about the `workorder`, `wonum`, and `status` attributes and all attributes from the `invreserve` child object are provided .

There is an alternative method to get the schema while you are fetching details in a collection query. The following query is a simple collection query:

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

In addition to fetching the work order records, if you want the schema for this `oslc.select` clause that fetches part details from `workorder` (`wonum` etc), part from `invreserve`, and `part` from asset (which is not even part of the OS), by using the dot notation, you add the query parameter `addschema=1` to the request URL. When you add the parameter, the response JSON objects `responseInfo` property will have the schema inlined inside it. This returns the data and the metadata in the same REST API call. Note that this schema will not be fetched for the next page request beacause the next page URL will not have the `addschema=1` in the URI. However, if you add that query parameter explicitly, it fetches the schema for any page.

Support for MBO schemas was added in Maximo Asset Management 7.6.0.9. This support is critical for use cases where you need to fetch a related MboSet using the REST API without using the `responseos` query parameter. For example, if you want to evaluate the `getlist` API for an attribute, such as the status for a given work order, you might use the following API:

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

This API returns the possible list of status values for that work order state. The response is a serialized version of the `synonymdomain` MBO and is not an object structure. If you want to have a schema for the response, you can add the query parameter `&addschema=1` , which works for the response MboSet, without needing to set it as an object structure.

This MBO schema can also be accessed standalone by using the `jsonmboschemas` route. The following example shows a sample call:

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

The sample call returns 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`).

Similarly, when you access some relation as a Mbo(Set), you can apply JSON schemas without using an object structure. The following example shows the use case:

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

The related vendor for the `PO` is accessed and you do not use any `responseos` query parameter to render the response to as an OS, but you can access the schema of the `vendor` (which is the `companies` MBO).

Note that this MBO schema feature was introduced in Maximo Asset Management 7.6.0.9 and is not available in previous releases. The OS schema is available in releases before Maximo Asset Management 7.6.0.9.

11. Creating and updating resources

Creating resources are typically completed by using the collection URI, which is the same URI you use to query the resources. . For example, the API call for creating assets is shown in the following example:

HTTP

```
POST /oslc/os/mxasset

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

After you create the asset, you get a response that contains a `location` header with the URI of the new asset. You can now use that `location` URI to fetch the new resource. Rather than doing a `GET` to fetch the resource, you may want the response of the create to contain the new 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).

HTTP

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

{
...
}
```

OR

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

{
...
}
```

Using the `properties` header removes the need to do an extra `GET` for every create and update.

Now that an asset is created, you can update the asset. The following example shows how you can set a `location` and a `description` to the `asset`.

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

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

The URL has a `rest id` at the end of the collection URL. This URI is pointing to a member asset in the collection, which is why the `rest id` token is after the collection URI. Note that this rest id is not the unique id for the MBO but a generated id for the MBO that is created by using the primary-key attribute values. Also, the `x-method-override` request header and the value `PATCH` instructs the server side to update the resource. As with create, you can specify the properties request header to receive the results of an update. This example shows a value of `*`. However, to maintain performance you can define the set of specific properties that you need.

Note that we needed to fetch the URI of the `mxasset` resource in order to update it. The fetching of that URI can be done in two ways:

- We can get the URI as part of a GET collection query, which returns 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.

To add some child objects, the following API example adds two `assetmeters` to the new asset.

```HTTP
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
},

     ]
}
```

An extra request header called `patchtype` is used with a value of `MERGE` . This instructs the server to match the child objects - like `assetmeter` with the existing `assetmters` for this asset. If a match is found, that `assetmeter` gets updates with the request assetmeter. If a match is not found, an `assetmeter` is created. This action is called the `MERGE` API. For example, if the asset to be updated has three existing `assetmeters` , and the request contained one existing `assetmeter` and one new `assetmeter` , that asset will be having 4 `assetmeters` with one newly created one and another updated `assetmeter` after the merge call.

To highlight the difference between a `PATCH` and `MERGE` call, you can run the same request without the `patchtype` header on another similar asset with three ` assetmeters`. The server side creates a new `assetmeter` and update the existing `assetmeter` like the `MERGE` call. Unlike the `MERGE` call, the `assetmeters` that were not in the request are deleted. In this example, only two ` assetmeters` (that are in the request) remains in the asset. Thus in a `PATCH` request, the server side deletes all child objects that are not in the request payload.

To update the child objects, the following example updates the `assetmeter` object to set the meter reading. Make sure that the `np` attributes, `newreading` and `newreadingdate` are included in the `MXASSET` object structure.

```
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 the primary keys of the `assetmeter - metername` and `linearassetid` for the meter update are included. Another option that the API allows is to use the href URI for the `assetmeter` instead of the primary keys, which is shown in the following example:

```
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”
}
     ]
}
```

The following request below shows how to selectively delete a child object:

```
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 the 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` .

By 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.

12. Deleting resources

In the last section we talked about deleting child objects in an object structure resource. Any updating 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 by using the following sample API:

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

You get a response of 200 OK if the delete is successful. Starting 7608 you are able to do the same by using the following API:

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

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

In certain cases, some browsers do not allow the http `DELETE` and this `POST` equivalent is a good alternative.

It is also important to consider the fact that in Maximo Asset Management deleting an MBO internally deletes the dependent child objects. The REST API 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 used to determine if this MBO can be deleted. This is helpful in cases where you want to delete the resource in two steps, firstly to mark the resource for deletion and then to later delete the resource. To mark a resource for deletion, you need to verify if the resource can be deleted by using the MBO call back API nameed `canDelete()`. The following REST API exmple shows how to invoke `canDelete()`:

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

A `200 OK` response indicates that the resource can be deleted. This depends on the application MBO implementing the canDelete method, which is not implemented by default in the MBO framework. If deletion is not allowed, an JSON error would indicate the reason, which is determined by the application MBO.

13. Actions

The functionality of `Standard Services` is available to get information about application services. Maximo Asset Management has multiple application services - aka Appservices, which provides service methods that act on MBOs to perform a business task. While most of these calls end up modifying the state of the system, some of these calls are just to get information. You can use 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, you can add a `WebMethod` to any existing Maximo application service by extending the service class and adding a method. This example is from extending the `Workorder` Service (`psdi.app.workorder.WOService`):

JAVA

```
@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 available for REST API calls. You can select any work order record and use the rest API to get the set of allowed actions on that resource as shown in the following query:

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

You will see the list of allowed actions, which are 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 provides 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. The following sample call demonstrates how these methods are invoked:

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

{
    “memo”:”Testing”
}
```

Note you 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). This MBO reference is derived from the request URI (the URI of the work order 1001) and passed into the method as part of the API invocation. The JSON schema for the payload reflects the parameter data types (in the method) and the parameter names.

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` .

In additon to this, we also support the concept of streaming action apis where one can directly access the REST request input stream. In this model you can define a java method with one and only one method parameter of type `com.ibm.tivoli.maximo.oslc.provider.OslcRequest` . An example is shown below

JAVA

```java
@WebMethod
public void loadFile(OslcRequest request)
{
    //you can now use the request to get the query params as well as the inputstream like
    String params1 = request.getQueryParam("param1");
    InputStream is = request.getInputStream();
    //use the input stream to read the blob which is in the body of the request
    ....
}
```

However there are limitations with overloaded methods, where overloaded methods are not supported for REST API calls.

There are methods available in the out of the box service and you can use `allowedactions` to see these methods.

In addition to `webmethod` based actions, the REST APIs also support scripted actions. You can write a REST action code by using automation scripts. A simple example below will explain this concept.

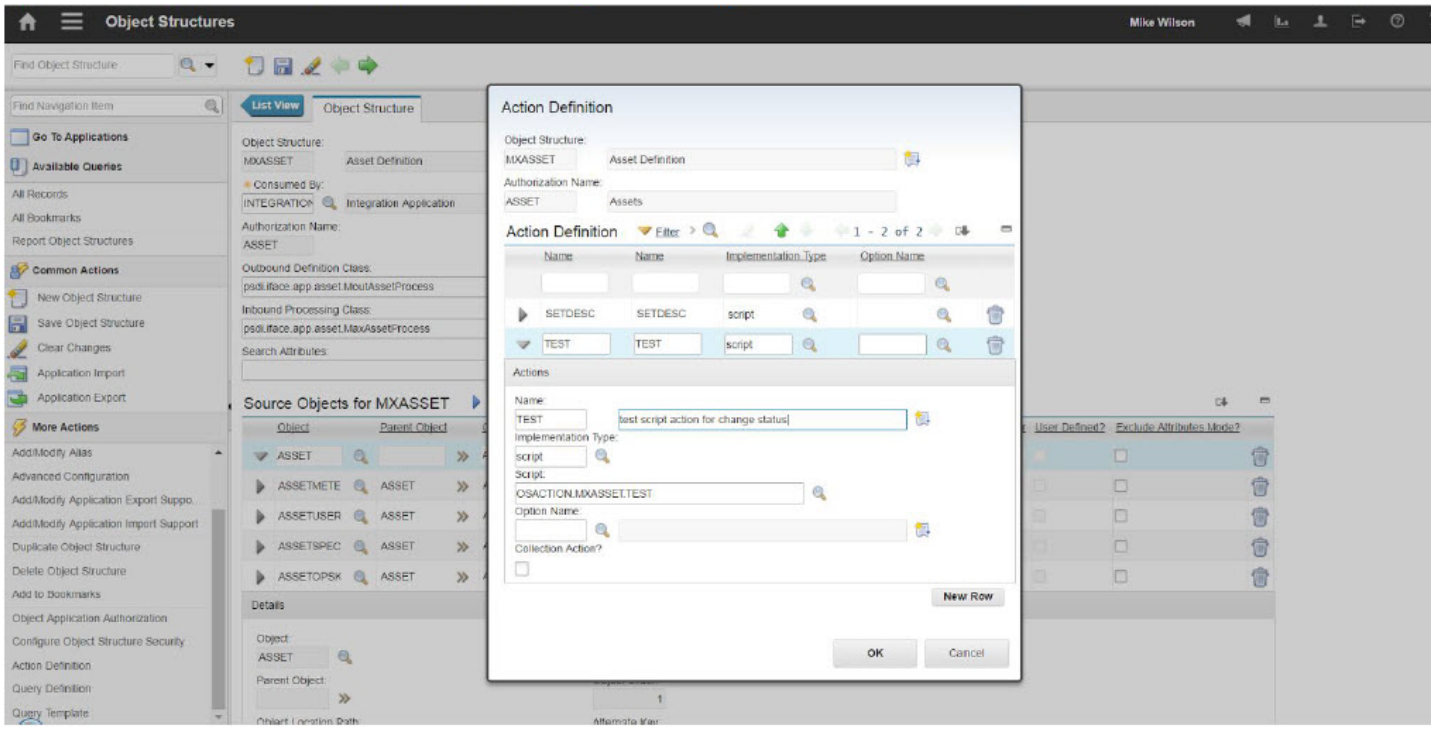
Firstly, you create an automation script in the Automation script application by using the **Create** > **Script For Integration** option. You select the object structure, then select the type of script as `Action Processing` , and give a name for the action. You can write a script (in any language - `python` , `js`) by using the implicit variable `mbo` . For example, the following script changes the status of Asset to `OPERATING` :

JAVASCRIPT

```
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 register that script with the object structure in the Object Structure application. Select the **Action Definition** action and set the script as an action to the object structure. Name the action same as the name of the script and click **OK** to save the configuration.



Next you can invoke this script similar to the following request:

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

The asset status changes to `OPERATING`. Note that if you want to get the changed Asset in response, you can add the request header `properties` with your desired value. You can also use this actions rest APIs to invoke `workflows`. The following example shows how the `ABC` workflow for the asset is invoked:

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

For non-interactive workflows, this is all you need to do to initiate it. Interactive workflows require user interaction by the placement of assignments, input, and interaction nodes. For more information about interactive workflows, see Interfacing with the workflow engine.

14. Automation scripts

The REST APIs have a tight integration with the automation scripts. Automation scripts can be used to develop custom APIs. The following example call describes how automation scripts interact with REST APIs.

To find the total number of work that is in progress and service requests in a given site, you need to create an API since there is no out-of-the-box API for this task. Since this is a REST API with a JSON response, you can use the JavaScript language for scripting. You call this API *countofwoandsr*.

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

The response in JSON is shownn:

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

You can write the following script for this API:

Script Name: `countofwoandsr`

```javascript
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);
```

After you save the script, open your browser and initiate the GET request to validate the results.

You can also use automation scripts for implementing custom queries and custom actions.

15. Bulk operations

With Maximo RESTful JSON API, you can process multiple resources with multiple operations in a single transaction. The bulk process is supported only by using collection URLs. You can use the 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.

15.1. Creation of multiple resources with BULK

You can create multiple resources in a single transaction. As the regular creation, each of the resources go through the validation process. And the response for each of the resources are 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 is still be 200. However, the error message similar to the messages in the following example is shown in the response:

Response JSON:

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 asset errored out because of invalid site and hence returns a 400 with an error message. The second asset returns a 201 with the URI to the newly created asset resource. The overall HTTP response code would still be 200. In the above case, even if both the asset json faced a business validation error, the overall response code would still be 200. The only cases you will see an error status is when the request is structurally invalid (say invalid json) or maybe for some request level error like the authentication failed.

In Maximo Asset Management 7.6.0.8, the request JSON structures for bulk request is simplified. For example, _action at the data level is supported, where the _meta and _data are no longer needed.

```
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, the `_data` and `_meta` is not required with the use of `_action` and href embedded inside the data.

The following example shows the support for bulk change status:

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

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

The response format has not changed.

15.2. 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 by 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).

The following example JSON data will complete these tasks:

- 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 it, 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, the response code is a `200` but you must examine the response information in the response JSON body to determine if processing of each asset was successful.

16. Handling attachments

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

To fetch, create, update, or delete an attachment for a resource by using API, such as `MXASSET`, complete the following tasks:

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

16.1. Fetch the attachments

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

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

JSON

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

If you use the `doclinks` URL from the JSON data in the previous example, you receive a list of attached documents (reference to those documents) with the metadata.

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

JSON

```
{
"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 can 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.

16.2. Creating attachments

The API supports the creation of attachments that are associated to resources. For example, after creating an asset, 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. You cannot create the attachment at the time of creating the resource.

An attachment is made up of two components:

1. Attachment file: You create an attachment by using the HTTP POST with binary content or base64 binary content. There is currently no support for multi-part messages.
2. 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-asciicharacters 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 (as shown in the following example).

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

The GET on that URL will get the attached document that was uploaded before. Along with that, it also has a response header that is named Link, which has 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 the following example.

To create attachments of a WWW (URL) type, you can use the following request as an example:

```
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
```

In the response, you 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 and document type. The URL type is a synonymdomain in Maximo Asset Management and hence hardcoding the values `FILE` or `URL` maybe a problem in case those values have been modified during installation. You could potentially do one of the 2

1. Avoid specifying the URL type altogether. The API framework defaults to 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.

16.3. Updating attachments

There is no support to update an attachment. To revise an attachment, you need to delete the current version and create a new version.

16.4. Deleting attachments

You can delete attachments by using the 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
```

16.5. Handling attachments as part of the resource JSON

From Maximo Asset Management 7.6.0.6, the handling of attachments as part of the Object structure JSON is supported. The following example adds two 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"
   }
   ]
}
```

In this example, the `documentdata` attribute has the base64 encoded document.

Starting 7611, we also support gettinmg attachments inline as part of a json document. To do this just use the query parameter inlinedoc=1. An example shown below

```
GET /oslc/os/mxapiasset/{rest id}?inlinedoc=1

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

17. 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.

The following table describes the query parameters for aggregation:

Query Parameters	Description	Example
gbcols	Define the attributes and aggregation function	gbcols=siteid, min.budgetcost,max.budgetcost,avg.totalcost
gbfilter	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

17.1. Aggregation column

The aggregation API is running based on the attribute and aggregation function that is defined in `gbcols`. For example, to get the minimum, maximum budget cost, the average total cost, and the count number of the assets for all of the site, you must provide at least one attribute as the grouping on attribute. In this case, you use `siteid`.

The following table describes the syntax and examples:

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` shows the grouped value. The aggregation attributes like `max.attributename` shows as `max_attributename`. The result also includes the data as well as a Resource Collection link, which gives you the resources in current group.

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

Expected result:

JSON

```
{
"count": 75,
"max_budgetcost": 2765.00,
"collectionref":
".../oslc/os/mxapiasset?&oslc.where=siteid%3D%22BOSTON%22",
"avg_totalcost": 30901.1733333333333333333333,
"siteid": "BOSTON",
"min_budgetcost": 390.00
  },
```

The `gbcols` also support multiple level aggregation. You can 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,avg.totalcost
```

17.1.1. Expected result:

JSON

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

17.2. Aggregation Filter

After you get the aggregation result set, you may want to get the smaller set based on the requirements. In this case, you can use 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 there is:

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

Expected result:

JSON

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

17.3. Aggregation Sort By

In aggregation API, you 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 tested in the browser, it is necessary to encode the value for `gbsortby`. There is an online site, such as http://meyerweb.com/eric/tools/dencoder/, to help you to do it. Copy the value `+siteid` from URL and encode it. Bring it and copy to URL. Finally, you can get `gbsortby=%2Bsiteid` to show the results by Site in ascending order.

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

17.4. Aggregation Range

In aggregation API, you can show the result set in different groups in one range with `gbrange`. In Maximo Asset Management 7.6.0.6, `count` is supported.

17.4.1. String(ALN) value:

For example, you are want the total count for work orders that are in `WAPPR` or `APPR` status. You need to append the following term to the URL. In the 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 results, like `status="CAN"`, are shown as the regular grouped result.

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

Expected result:

JSON

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

17.4.2. Numeric value:

Nnumeric value for `gbrange` is supported. 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 this example:

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

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:

JSON

```
{
"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
]
  }
```

18. Selecting Distinct Data

Selecting distinct data is done by replacing the `oslc.select` clause in a collection URL with query parameter `distinct=<attribute name>`. The following example:

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

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

```
["BEDFORD","NASHUA"...]
```

19. Dealing With hierarchical data

Maximo Asset Management has many hierarchical objects, such as `Locations`, `Workorders`, `Assets`, `Failure codes`, etc and there are subtle differences between each of these hierarchies.

19.1. Location hierarchy

The location hierarchy is always scoped under the `LOCSYSTEM` object. Effectively, a given location can belong to multiple systems, and hence different hierarchies. The API is designed where the list of systems that is available for the user is first with the hierarchy under the scope of that system.

For example, you can get a list of systems for the `MXAPILOCSYSTEM`object structure by using the following API:

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

With this API, you go to the top (`root`) location under that system and select the system that you want to drill down using the href of the system record. The following API call 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. The object structure name is added to the relation name to get the response as the `MXAPIOPERLOC` os. This helps to jump from one object structure to the other using the rest APIs.

To drill down under that top level location, you can use the href of the location object and append the relation name **syschildren** with the object structure **mxapioperloc** (to keep within the context of the `mxapioperloc` OS). The following API shows this format:

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

The a query parameter `ctx` is also used with the value of `systemid=<systemid>` for the system that is being drilled down into. This is needed because a location can belong to multiple systems and hence multiple hierarchies. So when drilling down into the hierarchy, the systemid context is required.

You can take any of the locations in the response and drill down by following the API described in the previous example. Always remember to set the `ctx` query parameter with the right `systemid` or the API defaults to the primary system for that location for the drill down.

These collection responses can be filtered, sorted, or paged just like any otheOSos 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).

19.2. General Ledger component hierarchies

General Ledger (GL) component hierarchies provides another flavor of hierarchies in Maximo Asset Management. 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. To specify a general ledger account, you need an API to find the segments in a hierarchical way (following the gl order). The following APIs describes how to find segments:

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

The response for this API is a list of GL segments at the top level, that is GL order 0. For each of the records, you can look for the childcompref URI. If you do a GET on that URL, you 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, the metacharacters ? (as configured) and the segment separators to represent the account are used. The GET on the childcompref would look like the following API call:

```
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 is updated to point to the next set of segments. Internally, the | separator for the segments are used and hence the childcompref URL for the third set of segments look like

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

You can also specify the `glcompsofar` value to drill down too as shown in the following API call:

```
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, you can sort (`oslc.orderBy`) and filter (`oslc.where`) bu using these APIs.

20. Interfacing with the workflow engine

Initiating a workflow for a given MBO can be done by using the following API:

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

This invokes the named workflow in the context of the MBO that is identified in the URI.

20.1. Handling task nodes

After initiation, the workflow may end up in a task node, which generates an assignment. The following APIs shows how you can handle assignments and where you can fetch assignments.

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

All the assignments for that user are fetched. Each assignment has a positive action and a negative action to take as shown in the following example JSON:

JSON

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

Note that the `wfaction` JSON contains the positive and negative actions and you choose one of those actions.

The following API call 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.

20.2. Handling input nodes

Input nodes provide you with interactive options to choose from in a workflow path. You do not need to 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 you accepted or rejected) is returned.

1. A response JSON that has the details of the options that the input node provides. The consuming client code is supposed to use those options to let you decide which option to chose.
2. A response location header with the URL to POST the users choice to.

The response JSON may look like:

JSON

```
{
    “member”:[
    {
       “actionid”:..,
       “Instruction”:”.....”
    },
    {
       “actionid”:..,
       “Instruction”:”.....”
    }
    ]
    “nodetype”:”INPUT”,
    "internalnodetype": "WFINPUT"
}
```

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

The following API call 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.

20.3. Handling interaction nodes

Interaction nodes are Maximo UI dialogs, applications, or tabs that are presented for you to take an action by using that UI artifact. Unlike an input node, an interaction node 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 indicates the details of the interaction node, presenting the information from the `WFINTERACTION` table for that node. This identifies (using the JSON property `internalnodetype` with a value of `WFINTERACTION`) the client code that provides an equivalent interface for the dialog or application. Like the case with the input node, the rest framework generates 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.

If you ignore this node, the system just moves on to the next node. To indicate that the interaction node job is complete, you need to re-route the workflow by pressing the workflow route button in the Maximo Asset Management application. To simulate that in the API realm, the client code needs to apply the following api call:

```
POST <location uri>

{
     “interactioncomplete”:1
}
```

This indicates to the workflow engine that the interaction is complete.

20.4. 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. There is no handling of APIs for this node as this is backend event driven. So if an event comes from any API calls, MIF calls, or UI call for that MBO, the workflow moves to the next node in the path if the condition is met.

20.5. Handling condition nodes

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

21. Saved queries

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

21.1. Available queries for object structures

For each object structure, you can find all authorized (for the requesting user) saved queries by using the apimeta API call. See the following MXASSET object structure example:

```
GET oslc/apimeta/mxasset
```

JSON

```
{
..
"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"
},
]
}
```

There are four types of saved queries for Object structures in Maximo Asset Management.

21.1.1. 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 is used if a method was implemented for query purposes. Since there are no default query methods provided, this method would be a custom code implementation by using the following code example, `@PreparedQuery("http://maximo.nextgen.asset#publicAssets")`

JAVA

```
public void publicAssets(MboSet assetSet) throws MXException, RemoteException
{
   String whereusercust="assetnum not in (select assetnum from assetusercust)";
   assetSet.setUserWhere(whereusercust);
}
```

21.1.2. 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.

21.1.3. 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.

21.1.4. Applications query (appclause)

The query is sourced from a Public Saved Query of an application. Using `Asset` and `MXASSET` as an example, the query can be associated with object structures in following ways:

1. In the the Object Structure application, select the Query Definition action, set `type = appcluase` and select the query from the list.
2. Set the authorization name of `MXASSET` as `ASSET`. In apimeta, the saved query names is listed as original name.
3. Set the authorization name of `MXASSET` as `MXASSET`, then check load queries from all applications. If you have the access to `ASSET`, in apimeta, the saved query name shows as `ASSET:QueryName`

For information about OSLC Query, see Ability to set query definitions and action associations in the Object Structures application (http://www-01.ibm.com/support/docview.wss?uid=swg21972876).

21.2. Execute saved query for object structures

For Maximo Asset Management RESTful APIs, the query parameter for all of the saved query is `savedQuery`. This parameter is case-sensitive. If you apply `SAVEDQUERY` or `savedquery`, the parameter is ignored as an invalid query parameter without any errors.

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
```

21.3. Executing KPI clause for object structures

By using RESTful APIs, you can get more detail for a KPI by calling its where clause with saved query as follows@

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

The API takes the where clause from `KPI` and apply it to the `MXASSET` object Ssructure. The KPI clause is not be available in `APIMETA` and you have to make sure the where clause in `KPI` can be applied to the main object of the object structure. Otherwise, you will get the SQL error.

22. Query templates

A query Template is an object structure-based template that includes the query related definition. In the collection level, you can define the page size, search attributes, and timeline attribute in the template. In the attribute level, you can add selected attributes, give the ordered information, and override the title of attribute.

After reading this section, you will be able to create a query template for object structures, apply the template to object structure, and get selected attribute and ordered collection back.

Currently, the query template can be created by JSON API. Using MXASSET as an example, you complete the following tasks where the object structure is named `MXAPIQUERYTEMPLATE` for `querytemplate:`

22.1. Setting up the 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 query template is 1001. Since attributes are not defined yet, the result set only contains the reference links for each of the record. When you apply the query template to object structure, in this case, `MXASSET` ,you can use the following query parameter to generate the restful call with query template called `querytemplate=1001` :

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

Expected Response Info:

JSON

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

From the result set, there are only five records that are included in the first collection page. Since the searchAttribute and timeline attribute are already defined in query template, you can use `oslc.searchTerm` and `tlrange` to filter the result set.

```
GET /oslc/os/mxasset?querytemplate=1001&oslc.searchTerm="PUMP"&tlrange=-3M&collectioncount=1
```

Expected Response Info:

JSON

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

22.2. Setting up query templates with attributes

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

Table 2. 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 five records per page, where each of the objects include the `assetnum`, `status` and `siteid`. The records are sorted by Site in ascending order.

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

Expected Result:

JSON

```
{
      "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"
},
```

You can also use more complex syntax, with * notation after the attribute.

Table 3. 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

For example, if you want 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*	

22.3. Differences between basic and advanced format

The differences between `relName.AttrName`, `objName.attrName*` and `rel$relName.attrName*`, using `MXPO` as an example with the relationship: `Name = VENDOR, Parent = PO, Child = COMPANIES`.

To get the `name` for company, there are 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*:` Converts to companies{name} by the query template. Usually, it's used when you **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*:` Converts to `rel.vendor{name}` by the query template. Usually, it's used when you **don't have COMPANIES** as our child object in `MXPO` but you still want to get multiple records back if the result is one to many.

23. Troubleshooting the REST API

The REST API uses primarily the `integration` and `oslc` loggers for the API framework part. Enabling those two loggers to `DEBUG` or `INFO` provides debugging information. However, the rest APIs always interface with Maximo business objects and other Maximo artifacts, such as security, scripting etc, which have their own loggers. Additionally, you can enable the SQL loggers if the query result is not what the filter clause described.

One option to debug is 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 in the Logging application by selecting **Configure Custom logging** > **Thread logging**. Choose the context name as “OSLC” and the user name as the “personid” of the user whose REST requests you want to track or debug. Then you can enable the logging with - sql,oslc and integration loggers to start with.

You can also enabling the thread logging by using the following REST API:

```
POST /oslc/log/enablelogs

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

This API enables the thread logging for the current user that is logged in for the loggers `sql,oslc` and integration. There is another api /oslc/log/enablealllogs that enables all loggers for the user. It is recommended not to set that one right away as it would generate a multiple logs, making is difficult to debug. You can disable this logging by using the following call:

```
POST  /oslc/log/disablealllogs

<no request body needed>
```

As you make the requests with this setup for the desired user, the system keeps track of all the oslc, integration and SQL logs that are 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 by using the REST API call `GET /oslc/log`. This API call streams 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, which is a manual process. There is no REST API for that task.

24. Password management using REST API

You can manage passwords only when Maximo Asset Management is configured to use the Maximo native authentication scheme. If Maximo Asset Management 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) is responsible for password management and hence their admin tools should be leveraged to complete this task.

24.1. Changing passwords

You change passwords when the password expires for a 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, the user receives a `403` error with the `BMX id` of `BMXAA2283E`. The client code can detect that error and then shows the password change dialog for the end user to change the password. The following API shows how to change passwords: Note the `maxauth` header shows 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)"
}
```

24.2. Forgot password

The following API is used to reset passwords when users have forgotten their 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 API sends an email with the reset password. Note unlike the change password API, you do not have to provide the maxauth header since the user does not remember the password.

25. Supporting file import (CSV/XML) and import preview using REST API

From Maximo Asset Management 7.6.0.9, you can upload data from flat files or XML files to Maximo Asset Management by using RESTful API `action=importfile` by using the application import capability.

25.1. Preparing object structures

To import XML files, you do not need to configure the object structure. To import flat files, similar with the application import, you need to configure object structure to support the flat file structure. This implies there shouldn’t be any alias conflict for that specific object structure. To verify that, find the object structure, such as MXAPIMETER, in the Object Structure application. Make sure the Support Flat Structure check box is selected and no alias conflict is detected by the system (the Alias Conflict check box is clear). If there are any conflicting fields, select the Add/Modify Alias action and add a new alias to the conflicted field, starting with the first child object. The rest of this section focuses on flat files but XML files is the same with a different header value.

25.2. 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.

25.3. Prepare CSV

The CSV format is the same as a typical application import in Maximo Asset Management. When you import files from CSV, the attribute name of each field must match with attribute alias in Maximo Asset Management.

For example, the CSV file content can be as follows

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

25.4. Preview

Before processing the data into the database, you should validate if there any errors occured by using the preview functionality, which is supported in the import file API. For example:

```
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 you make the POST request to server, the system returns the preview response, which includes the validation information and warning messages and shown in the following error message example:

```json
{
    "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.
}
```

You can determine the problem from warning messages and then fix it. In the sample error responses, you can tell that the issue is caused by missing the security setup. After granting the sigoptions and reprocess the call, the successful preview looks like following example:

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

25.5. File import

After previewing, you can import the file into Maximo Asset Management 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
```

The response is as follows:

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

Other file types are supported, 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

26. Supporting file export

File exporting can be done by using query parameter _format=csv. You can download data from any object structure, for example `MXAPIASSET`, to a flat file. Similar, with File Import, before exporting data, supporting flat file, no alias conflict and security set up are required for the target object structure.

26.1. File export

`_format=csv` is always used with other query parameters together for different tasks.

The following parameters are the most common types:

`oslc.select :` Controls 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:` Controls how many records that are downloaded from server.

`oslc.orderBy:` Defines the orderby column in csv file.

`oslc.where:` Filters the data.

Most querying or filtering capabilities are available for exporting.

The following example is a REST call:

```
GET /oslc/os/mxapiasset?
lean=1&oslc.select=assetnum,siteid,description,location&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`.

27. Creating users using the REST API

Creating users can be done by using the `MXPERUSER` (or `MXPERAPIUSER`) object structure, which creates the user and person simultaneously. The following REST call is shown:

```
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 must 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 Asset Management is the owner of the authentication. If authentication is handled by the application server, the passwordinput and passwordcheck attributes (or any other password management details like emailpswd, generatepswd, password hint question, force expiration etc) is not required. Passwords are not managed by Maximo Asset Management when the mxe.useAppServerSecurity property is set to 1.

28. Creating a Multi-tenant using the REST api

You can create an object structure (say MXAPITENANTREG) with the tenantdbuserid attribute restricted and include the newusergroup and the docroot non-persistent attributes.

The following example 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 POST creates the tenant and the tenant admin with the login ID of myabc123. Also, an email is sent to the primary email address with the generated password. You must make sure that the smtp host is setup for this. If the application server security is enabled, then this smtp setup may not be needed as the password management is done outside of Maximo Asset Managemnt. Note an MT tenant cannot be created without the tenant admin. Also, note that this API needs to be invoked in the context of MT landlord.

28.1. Handling interactive logic by 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, you need to make the REST API request interactive. By default, all rest requests are not interactive. Therefore, the server side logic chooses the default option and executes the default logic. This may not be desirable in all cases. To allow users choose the options, interactive requests are provided in this API. The following example shows how to make that request:

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

This marks that request as interactive and now executes the interactive logic on the server side. However, you need to somehow set the desired user input for the interactive logic. To do that you need to set the request header `yncuserinput` where the value can be a `;` separated list of name value pairs - each name corresponds to the `YesNo` key - for example in the `FldWoAssetnum` class one of the interactions is shown:

```
MXApplicationYesNoCancelException.getUserInput("woassetlocation_change",...
```

The request header would look like:

```
yncuserinput: woassetlocation_change:<value>
```

where the value is one of

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

An example shown below:

```
yncuserinput: woassetlocation_change:8
```

If you have YNC nested - like one YNC leads to the other - you can solve all of them by providing the values in sequence - such as:

```
yncuserinput: woassetlocation_change:8;<someotherkey>:<someothervalue>
```

A sample YNC response from the rest api would look like below


```
{
  "yncuserinputoptions": {
    "no": "16",
    "yes": "8",
    "close": "1"
  },
  "Error": {
    "errorattrname": "assetnum",
    "extendedError": {
      "moreInfo": {
        "href": "http:\/\/localhost:7001\/maximo\/oslc\/error\/messages\/BMXAA4722E"
      }
    },
    "errorobjpath": "workorder",
    "correlationid": null,
    "yncerror": true,
    "reasonCode": "BMXAA4722E",
    "message": "BMXAA4722E - The specified asset is not in the current location. Do you want to update the
location with this  asset's location - BPM3100? If you do not want to apply changes to the location or asset,
click Close.",
    "yncuserinputid": "woassetlocation_change",
    "statusCode": "400"
  }
}
```


Note that its an error json (as it originated because the server threw an exception - of YNC kind). Also note that the "message" contains the question to the user. Also the "yncuserinputid" contains the id of the YNC interaction. This will be used in the header `yncuserinput` as discussed before.

29. Virtual (Non-persistent) Mbo support in REST API

A lot of Maximo business processes have been developed using virtual (non-persistent) mbos. These mbos have subtle differences in the life cycle callbacks/events compared to their persistent equivalents. MIF/REST did not full support for those callbacks like `setup()` and `execute()` until recently. Starting 761 this support is available. Using this, we can now leverage a lot Maximo functionality using apis/mif that was previously not easily accesible. Below is an example of downtime reporting using object structure MXAPIASSET. Note that the virtual mbo DOWNTIMEREPORT has been added to the MXAPIASSET as a child object to ASSET.

```
POST /oslc/os/mxapiasset?lean=1
x-method-override: SYNC
patchtype: MERGE

{
  "assetnum":"13150",
  "siteid":"BEDFORD",
  "downtimereport":[
    {
      "isdowntimereport":"1",
      "startdate": "2019-07-17T22:57:56-04:00",
      "enddate":"2019-07-17T22:58:59-04:00",
      "code":"BRKDWN"
    }
    ]
}
```

The above example also demonstrates the usage of the SYNC header, which is used when we want to avoid providing the individual asset url (with the rest id) and instead just want to use the collection url. The same could have been achieved using the asset url with the rest id and a x-method-override value as PATCH.

30. Handling duplicate requests in REST API

Sometimes the REST call committed successfully on the server, but the communication channel broke and a `500` error occurs. You may think that the server rolled back the transaction and resend the request. This in some cases can result in erroneous or duplicate data. To avoid this situation, 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` where the API framework validates for duplication. If no matches are found, the transaction is good to go. If a match is found, the request is rejected with a HTTP `409` Conflict error.

Note that the transactionid header value is user 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 five minutes, controlled by the escalation `OSLCTXN`. However, this can be modified as per the installation need.

Note that this feature is primarily useful when you 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.

31. Interfacing with BIRT reports using REST apis

Starting 7609 REST apis support interfacing with BIRT reports. We provide 2 basic apis:

- Get the list of available reports for a given user and a given application.
- Generate a report for a given set of mbos.

Below we show a sample of the "get list of reports" api for the MXAPIWODETAIL object structure with a sample response.

```
GET /oslc/os/mxapiwodetail?action=listreports

[
  {
     "reportname": "woprint.rptdesign",
     "description": "Work Order Details",
     "genreport": "http://localhost:7001/maximo/oslc/os/MXAPIWODETAIL?
action=genreport&reportname=woprint.rptdesign"
  },
  {
     "reportname": "wotrack.rptdesign",
     "description": "Work Order List",
     "genreport": "http://localhost:7001/maximo/oslc/os/MXAPIWODETAIL?
action=genreport&reportname=wotrack.rptdesign"
  }
]
```

Note that for the object structure MXAPIWODETAIL, there is an associated auth app to which the reports need to get associated with. The `genreport` property contains the url for generating individual reports.

```
GET /oslc/os/MXAPIWODETAIL?action=genreport&reportname=woprint.rptdesign
```

This will generate the report in the deafult format (PDF). You can specify other supported formats using query parameter `reportformat`. We can filter the list of mbos using the `oslc.where` query parameter. Additionally we can also add attachments to the generated report setting the `attachment` query parameter to 1 (default value is 0 which implies no attachments).

It also supports report parameters - as rest api query parameters, with the same name.

The generated pdf is returned as a downloadable file as response to this GET request.

32. Interfacing with REST apis using API keys

Starting 7609 REST apis support the concept of API keys. Apikeys are tokens that can be used to make rest api calls without needing to provide the user credentials. Apikeys are created for a given Maximo user and the api calls made using that apikey effectively works in the context of the said user. This implies all the user's permissions are respected in that api call. REST apis invoked using api keys use "silent" login. This implies that there is no MAXSESSION entries created. By default, there would not be any server-side web sessions. This implies that clients using rest apis with api keys, do not need to call the /logout api as there is no persistent server-side session. Api keys can be created and revoked as needed. There is only one api key per user allowed at this point. Each of these api keys may have an expiry (in minutes). The expiry time is specified at the time of creation. An expiry time of -1 implies the key will never expire. The only way to invalidate the key would be revoke it manually. Below we list the rest apis that can be used to create/revoke the api keys.

To create the api keys, login as the user for which you want to create the key and use the rest api shown below.

```
POST /oslc/apitoken/create
{
     “expiration”:-1
}
```

This will respond back with a json containing the apikey value.

To revoke that api key,

```
POST /oslc/apitoken/revoke
```

As you may have noticed, the create and revoke apis do not take in any user name as the user name is implicit from the logged in user. In case, the adminitrator needs to do this for a given user, we can use the Object structure based apis using the MXAPIAPIKEY Object structure.

```
POST /oslc/os/mxapiapikey

{
     "expiration":-1,
         "userid":"WILSON"
}
```

Note the userid maps to the MAXUSER.USERID attribute. The administrator can also get a list of the apikeys using

```
GET /oslc/os/mxapiapikey
```

Administrator can also delete the api keys using the DELETE REST call

```
DELETE /oslc/os/mxapiapikey/{id}
```

Note this Object structure was added only in 7611. For prior releases, customers can easily add a different object structure for this to do the same.

To make an api call using the api key, we need to add the apikey value as a query parameter or a request header to the rest api call. Below is an example for fetching assets

```
GET /oslc/os/mxapiasset?apikey=<the apikey value>&lean=1
```

While this model works out great for maximo authentication enabled deployments, it does not work out for app server authentication-based Maximo deployments. The reason being, in the latter model, the /oslc servlet is security constrained and hence the rest api calls would not be able to reach to the application code without the user credentials.

To overcome that in the upcoming release, 7611, we have added a route /api in the maximouiweb modules web.xml which maps to the same oslc servlet. The mapping is shown below

```
<servlet-mapping>
        <servlet-name>OSLCServlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

We also need to make sure this mapping should not get protected using security-constraints. This way we can make sure the application server security would not interfere with the apikey way of authenticating the rest api calls. This /api mapping is not present in the maximouiweb modules web.xml file for 7609 or 7610 releases. If the need is to leverage the apikeys in app-server authentication enabled Maximo, we recommend adding this mapping to the maximouiweb modules web.xml.

With this mapping the rest api calls would look like:

```
GET /api/os/mxapiasset?apikey=<the apikey value>&lean=1
x-public-uri:http://host:port/maximo/api
```

Note that without the x-public-uri request header, the response json will have href urls pointing to the /oslc context and not to the /api context. Note that this call works for app server security, as the /api context is not protected using the security-constraints. The rest api framework makes sure that the rest api call does not go through unless the apikey is provided.

This model with apikeys (along with the /api route) provides a way to make headless client calls to the rest apis when Maximo is configured with interactive authentication schemes like SAML/OIDC which otherwise would need some browser-based interaction to authenticate.

33. Performance Tips

When talking about api performance, we are mostly looking at reducing the number of calls that we are making from client (browser based power app/integration clients) to Maximo Server and Maximo Server to Maximo database. Below we discuss the common programming mistakes that are made during app or integration development regarding the usage of the apis.

33.1. Duplicate calls

Analyze using the network tab the requests that are being made from your application. You may see a bunch of duplicate requests which maybe the by-product of erroneous js coding/event handling.

33.2. Look for pageSize

For example - do not give a big pagesize when you are going to show only a few. General rule of thumb - make it 1:2 - so if you are planning to show only 10 - give a pageSize of 20.

33.3. Look out for attributes in oslc.select clause

33.3.1. Dot notation attributes

For example say we are fetching assets and the request looks like oslc.select=assetnum,description,location.description,location.location,site.description,siteid - you would perform much worse than if the query was oslc.select=assetnum,description,siteid,location - why? This is because for a pageSize of N, the former is going to fire 2N+1 sqls to the database as opposed to the latter - which will be always 1 sql. The former select will fire a sql to location table anf site table for every asset it selects. So if you were getting 100 assets - you will get 1+100 (for locations)+ 100 (for site) = 201 sqls. Also pay attention that location is a native attribute of the asset table - so using `location.location` will just make it go and make 100 calls for nothing. You could have just done `location` instead. In this case however it does not make a differnce as you are already fetching `location.description` - so you would have taken that hit anyways.

33.3.2. oslc.select=*

This is a common mistake we do. We are selecting all attributes from an OS or Mbo - when we may need only a few.

33.3.3. Domain description

This is a common requirement - we get domain descriptions for status and other domain bound attributes. For this, the rest api framework maintains a ML cache to store the descriptions for different domains. Whenever the framework detects the domain bound attributes - it automatically adds the description (from the cache) to the response json. Unfortunately we are seeing a lot of the rest api calls include a select clause that refers to the domain by a relationship - thus bypassing the cache completely → resulting in yet another sql.

33.4. Fetching count

There is a simple api call to fetch count `<rest collection url to the resource>?count=1` . When we fetch count, we donot need to fetch anything else with it, ie we should not be using oslc.select or collectioncount=1 as they would result it mbos getting initialized which will result in unnecessary sqls. If we need to fetch count for a set of child objects along with data from the parent, we can use the oslc.select=<parent attributes>, <relation name>._dbcount, where relation_name is the maximo relation name to the related object. if we just need to fetch count of a child object with no data from parent, use the GET call like /os/<os name>/{id}/<relation to child object >?count=1

33.5. Fetching data for other tabs while in one tab

In an multi-tabbed application, there is no need to fetch data for tabs that are not visible yet. Just-in-time fetching helps improve the app loading performance.

33.6. Lookout for properties header for POST requests

This is similar to oslc.select in GET cases. We should look out for things that we do not need - for example domain descriptions, imagelib references etc which are automatically populated. We donot need to explicitly specify them. Specifying them are costlier. Also avoid doing properties=*.

33.7. Sorting on non-indexed attribute

One of the common mistakes we see is rest queries use of oslc.orderBy clause sorting on non-indexed attributes like say `description` .Sorting on attributes that is not indexed will have a performance cost on the query. While developing the apps, consider not sorting by default and just use the order that the database provides. Let the end user drive the sorting.

33.8. ignorecollectionref=1 query parameter

This is another area for optimization. We can set this query parameter to 1 and reduce the size of the json payload response in cases when we are requesting data from a big OS like MXAPIWODETAIL. This will remove the collection ref to child collections from the response json. We should leverage it in our list page collections.

33.9. Evaluating/filtering data at the server side

Look for evaluating/filtering data at the server side as opposed to client side and only transfer required data to the client, effectively filter at the server as opposed to getting all data and then filtering on the client side.

33.10. Optimize selecting related mbos

This use case deals with situations where we are getting information from a related mbo on a collection of selected mbos. For example we try to get information on locations and sites for a given set of Assets. Now if from a collection of 100 Assets, we have 20 distinct Locations and 5 distinct sites – we will stil end up firing 100 + 100 = 200 sql queries for each asset loading the site and location individually. To optimize that we need to make sure we have entries in the apilinkedobject table for the relations in use from Asset→Location and Asset→Site. A sample Asset→Location is shown below. This tells the system that Asset is related to location using the “matchexpr” which helps the system to maintain a transient cache (of locations) while creating the json from the collection of Assets. Parent| Relation| Matchexpr ASSET |LOCATION | location,siteid

This will help reduce the number of queries to just the needed 20+5 based on unique locations and sites.

33.11. Aggressive fetching of data vs fetching data as needed

Avoid fetching all children objects in an object structure at once. Consider fetching those as needed basis ie if the requested by the end user. For example to fetch PO such that we can show it in a UI table/List – we maynot need to fetch the POLINES and Terms. They can be fetched after the PO’s are populated and as the user wants to drill down into

the individual PO's. This rest api provides various ways to enumerate a child relationship and those can be leveraged for this.

34. Troubleshooting Performance

This section discusses how to debug and generate the SQL for your rest calls. One of the ways we can troubleshoot performance of an api call is by checking the amount of sql its generating. The simplest way to do that would be to enable thread logging for the context "oslc" and for the user you intend to use for testing. This can be done using the "Custom Logging" action from the logging app → Thread Logging. Note you need to enable the logger type – in this case SQL. Note if you want to track other loggers feel free to set them up as well here for the "oslc" context. Once you start making the rest api calls – you should see a file getting generated under your application server working directory that will have a naming convention like "OSLC_XXXXX.log" and that will contain all the sqls and other logs that this api call generated

Last updated 2019-07-18 20:47:13 UTC