opentext™

REST API Developer's Guide

Product:	OpenText™ InfoArchive Compliance
Version:	23.3
Task/Topic	Customization, Development
Audience:	Developers, Administrators
Platform:	All
Document ID:	100047
Updated:	June 5, 2023

Contents

Introduction

This guide is intended to provide an overview of various aspects of the OpenText™ InfoArchive (IA) platform, focusing on REST API. It touches on several other topics that should help you understand the overall architecture, dependencies and how the system works in general. This guide is also complementary to InfoArchive REST Reference API documentation that covers every aspect of the REST API.
InfoArchive Reference REST API Documentation covers broad aspects of REST API resources. While this guide focuses on a subset of those resources, it also explains how the resources work and provides examples that should help you get started.
REST is a uniform interface which defines the interface between client and server. There are four guiding principles of a uniform interface:

- Resource based
- Manipulation of Resources Through Representations
- Self-descriptive messages
- Hypermedia as the engine of Application State (HATEOS)

For more information, it is recommended to view a tutorial on REST
https://www.restapitutorial.com/lessons/whatisrest.html#.

Any REST client can be used to access InfoArchive REST API. You are free to use your browser's plugins, command line tools and standalone applications or write your own REST clients in any of the preferred languages (Java, #Net, JavaScript, Objective-C, etc.). This guide uses the **curl** tool, which can be downloaded here: https://curl.haxx.se. It is a generic command line tool used for data transfer that can be used successfully as a REST client. Depending on the operating system you use, you may already have it installed or you may need to install it as an add-on to your existing operating system. If you do so, you should be able to run all the examples included in this guide.

REST API Full Documentation

This document represents a supplemental guide detailing some high-level concepts, authentication/authorization and several samples. For full documentation, including all the REST resources, please consult our published documentation on developer's site: https://developer.opentext.com/ce/products/infoarchive

Prerequisites

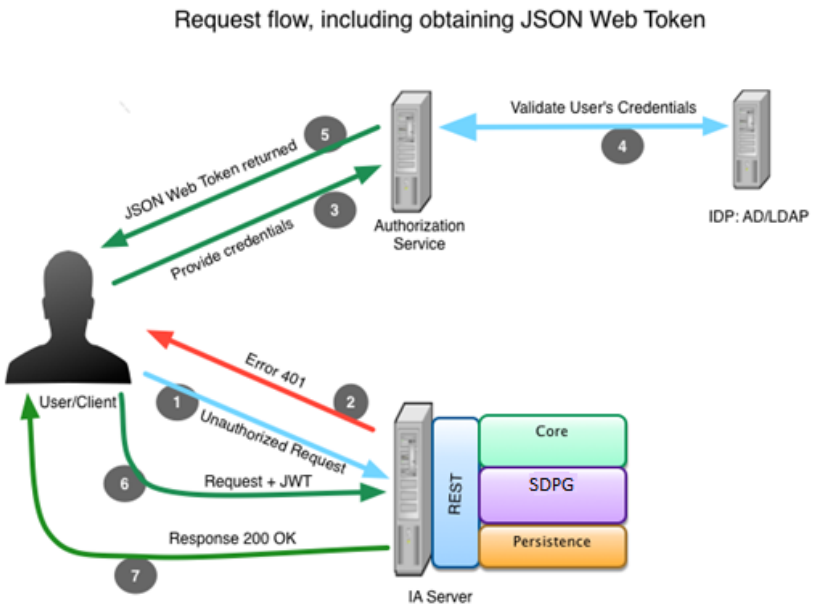
The bare minimum is to have a running InfoArchive environment. If you do not have access to the running InfoArchive environment, you will need to download the distribution and install it. An InfoArchive production environment does not require a lot of system resources so a test InfoArchive environment can be easily deployed on most desktops/laptops with minimum configuration (InfoArchive can even run on a Raspberry Pi).

Note about running the examples: You may copy/paste all the provided examples, but the JSON Web Token cannot be copied/pasted for two reasons:

- *JSON Web Tokens do expire. JSON Web token is explained later in this guide. So, you will need to get a fresh token if you intend to run provided examples in your own environment.*
- *JSON Web Tokens included in most examples have been shortened from their original length to keep examples shorter and clearer.*

Negotiation to Obtain JWT

The following diagram illustrates the process:



Step 1 shows the client issuing an unauthorized request (for example, Authorization header not set). Since this request does not includes the JWT, the REST Layer will respond with a 401 Error (**Step 2**), forcing the client to obtain authentication/authorization token (JWT).

Currently, IA does not provide redirection capabilities, so the client needs to be aware of where the Authorization Service resides and how to negotiate for a JSON Web Token, which is illustrated in **Step 3** and discussed in detail in next section.

> **NOTE:** The Gateway component (part of InfoArchive Web Application aka IAWA) of InfoArchive provides the Authentication and Authorization services. The Gateway component can also dispatch REST API calls to InfoArchive Server (aka IAS). It is encouraged to route the REST API calls through the Gateway so that the IA Servers do not need to be exposed

In this guide, most of the curl examples will not route through the gateway.

Step 4 shows internal verification of credentials within the IDP layer.

In **Step 5,** the client receives a valid JWT and then proceeds to include the token on a request (**Step 6)**.

Once the internal calls are completed (on IA Server Layers), a successful response is returned to the client (**Step 7)**. Subsequent calls' flows are much simpler, if the JWT is still valid and included with each request. For the duration of the JWT validity, no further interactions with the Authorization Service are necessary. The client can issue as many requests as will fit into the JWT time-to-live window. Since each window (JWT time-to-live) will eventually expire, it is a good idea to have a strategy on dealing with expired or potentially expired access JWT. The different strategies as well as intricacies involved in obtaining and refreshing JWT are discussed in more detail in next section.

Obtaining JWT for the First Time

To obtain a JSON Web Token, the client needs to obtain it from the Authorization Service. By default, there is an Authorization Service deployed within the InfoArchive Web Application. It can be accessed, as follows:

- Issue a POST request to Authorization Service. For example: http://localhost:8080/oauth/token
- Set the "**Authorization" request header** to the Base64 encoded value of **clientId + : + clientSecret**. Those values are configured in the corresponding **client** section of the **application-CLIENTS.yml** file located in the folder **[IA distribution root]/infoarchive/config/webapp**]. For example, if the **clientId** value is **infoarchive.custom,** and its value is **mysecret,** you would have to Base64 encode following string: **infoarchive.custom:mysecret**

> **NOTE:** If you are using OTDS SSO, in this mode OTDS is acting as a OAuth2 server. The OTDS is responsible for OAuth2 clients. To define new OTDS Clients, you must use OTDS Admin Web Application.

Set the Content-Type header value to **application/x-www-form-urlencoded**

For the Body of this POST request, set the following values (it is a form):

i. **username**=[name of end user] (for example, myuser@company.com)
ii. **password**=[user's password]
iii. **client_id**=[The value of this field should be the value of **clientId** from relevant section of yml file]
iv. **scope**=[one or more supported scopes from the client's section in the yml file]. If providing more than one scope, they should be separated by white space
v. **grant_type**=[one of the supported **authorizedGrantType** from the corresponding client's section of the application.yml file]

- If the POST method is successful, the server will return a 200 (OK) Response code, and the body will contain several values: **access_token, token_type, refresh_token, expires_in, scope, jti** – all in JSON format. The following is an example:

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cC...",
    "token_type": "bearer",
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5...",
    "expires_in": 2147483646,
    "scope": "administration compliance search",
    "jti": "82e6a274-d8fd-4067-99f0-dcd8231b35c1"
}
```

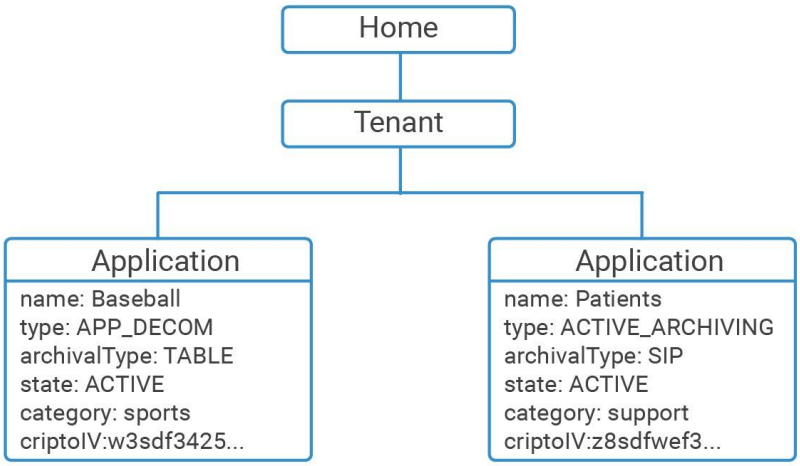

The following is a practical example performing a POST request using popular the **curl** tool:

```
curl -X POST -H 'Authorization: Basic aW5mb2FyY2hpdmUuY3VzdG9tOnNlY3JldA==' -d
'grant_type=password&username=connie@iacustomer.com&password=password&scope=search&client_id=info
archive.custom' localhost:8080/oauth/token
```

The response should appear like the following:


```
{
        "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjM2MzUyODE5NTMsInVzZXJfbmFtZSI6ImNvbm5pZUBpY
WN1c3RvbWVyLmNvbSIsImF1dGhvcml0aWVzIjpbIkdST1VQX0RFVkVMT1BFUiIsIlJPTEVfREVWRUxPUEVSIl0sI
mp0aSI6IjRiNTcyZjk3LTc2MmMtNDIwNC05ZjZjLWMwMzRjYzMwOGUzNCIsImNsaWVudF9pZCI6ImluZm9hcmN
oaXZlLmN1c3RvbSIsInNjb3BlIjpbImFkbWluaXN0cmF0aW9uIiwiY29tcGxpYW5jZSIsInNlYXJjaCJdfQ.YhnezAzS1n
LznNh2iCSdyRXkV94Ah3k51yIsZYTLFQ0",
        "token_type": "bearer",
        "refresh_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX25hbWUiOiJjb25uaWVAaWFjdXN0b21lci5jb20iLCJzY29wZ
SI6WyJhZG1pbmlzdHJhdGlvbiIsImNvbXBsaWFuY2UiLCJzZWFyY2giXSwiYXRpIjoiNGI1NzJmOTctNzYyYy00MjA
0LTlmNmMtYzAzNGNjMzA4ZTM0IiwiZXhwIjozNjM1MjgxOTUzLCJhdXRob3JpdGllcyI6WyJHUk9VUF9ERVZFT
E9QRVIiLCJST0xFX0RFVkVMT1BFUiJdLCJqdGkiOiJmYzhjNGFmOS1kYTU2LTRmMjMtODA5NC00NmQwNzcw
MTdjMTEiLCJjbGllbnRfaWQiOiJpbmZvYXJjaGl2ZS5jdXN0b20ifQ.ipQb8qcgzEGD7uT9dnOlXXzv5-
1wY3idN3kQzmUqsfU",
        "expires_in": 2147483646,
        "scope": "administration compliance search",
        "jti": "4b572f97-762c-4204-9f6c-c034cc308e34"
}
```

Application



The Application resource represents collections of applications available to a specific Tenant. InfoArchive supports two types of applications: Table Archiving and SIP Archiving applications. The **archiveType** attribute displays the value accordingly: **TABLE** or **SIP.** For more information about table and SIP archiving, refer to the InfoArchive Configuration and Administration guides.

Retrieving Application Resource

To retrieve a list of Application Resources, leverage the http://identifiers.emc.com/applications href value that resulted from the http://localhost:8765/systemdata/tenants call.

The following is an example using curl:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json'
localhost:8765/systemdata/tenants/41ceb11f-2943-4a4d-9fb4-bb198509942e/applications
```

If successful, then the IA Server will return a Payload Collection with one or more configured IA Applications.

Creating an Application Resource

Creating an application

In the previous section, a new Tenant resource was created that included all the links, including the link relations that point to a collection of applications (http://identifiers.emc.com/applications). To create a new Application resource, a POST Method must be executed to that collection.

To create an Application resource, at a minimum, the following attributes are required: **name** and **archiveType** (both are mandatory). We will also pass the **type** attribute and that should allow for the new Application resource to be created:

```
curl -X POST -H 'Content-Type: application/hal+json' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json' -d '{"name":"myFirstApplicaiton", "archiveType":"TABLE", "type":"APP_DECOMM"}' localhost:8765/systemdata/tenants/1177febe-d514-41ac-a3b9-f3b7357241eb/applications
```

After a successful call, the IA server will return a 201 Created response that includes the newly created Application Resource.

Declarative Configuration

Declarative configuration allows one to configure multiple objects in a single REST call. These can be SIP applications, table applications, or even both. The syntax and semantics of the declarative configuration is explained in the *Configuration Guide*.

Use the http://identifiers.emc.com/configuration link relation to discover URLs for importing or exporting declarative configurations.

The home resource supports importing a declarative configuration via PUT. This call takes the YAML as content:

```
curl -X PUT -H 'Content-Type: text/yaml' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json' –T configuration.yml localhost:8765/systemdata/configuration
```

Alternatively, you can import a ZIP file that contains the configuration YAML file plus any external files that are referenced by the YAML:

```
curl -X POST -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json' -H 'Content-Type: multipart/form-data' -F 'file=@configuration.zip' localhost:8765/systemdata/configuration
```

Exporting to a declarative configuration is supported on multiple levels: system, tenant, application, and holding. For instance, to export all configuration objects of an application to a single YAML file:

```
curl -X GET -H 'Accept: text/yaml' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' localhost:8765/systemdata/applications/5287aff2-3fd1-416d-875d-9e9ead71a471/configuration?exportReferenceDetails=create_or_update
```

For more information on the configure property, see the section on declarative configuration in the *Configuration Guide*.

You can also export the declarative configuration to a ZIP file. This ZIP will then contain the YAML file as before, but various larger pieces of content will be stored in separate files. This makes it easier to edit those files with dedicated editors, e.g., an HTML editor for XForms. Use the `Accept` header to specify the desired media type:

```
curl -X GET -H 'Accept: application/zip' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' localhost:8765/systemdata/5287aff2-3fd1-416d-875d-9e9ead71a471/configuration?exportReferenceDetails
```

Ingestion

As far as InfoArchive supports two kinds of application for archiving: Table based and SIP based, then there are two different mechanism of data ingestion to each type of applications. Both ingestion ways are covered in the chapter.

When planning about data ingestion from the REST perspective, it is recommended to start working at the application level via "**Home Resource**" -> "**Tenant**" -> "**Application**".

SIP Ingestion

Batch Ingestion

Batch ingestion is performed in two steps: Receive SIP step and Ingest SIP step.

Every step is performed by separate REST calls.

At first, we need to find the link relation http://identifiers.emc.com/aips rooted from the single SIP application and get an URI for the collection of AIPs by extracting **href**'s value for the link relation.

Based on this URI, we perform a GET call to retrieve the collection of AIP object:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json'
localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips
```

When there is no SIP data ingested into the application, the response is as follows:

```
{
 "_links": {
  "self": {
   "href": "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips"
  },
  "http://identifiers.emc.com/receive": {
   "href": "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips"
  },
  "http://identifiers.emc.com/ingest-direct": {
   "href": "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips?ingestDirect=true"
  }
 },
 "page": {
  "size": 10,
  "totalElements": 0,
  "totalPages": 0,
  "number": 0
 }
}
```

It contains only the page information and the number of relation links to perform different actions: "receive" and "ingest-direct".

Receive Step

We need to take an href's value (URI) for the http://identifiers.emc.com/receive link relation rooted from the AIP's collection resource. It is used for the reception step.

The reception is performed by a POST request to the above link, but it is required to set up several important parameters to set up form-data:

- -F “format=sip_zip” – it is a mandatory parameter that sets up the SIP format to be received. In the below example, it is “sip_zip”. Receiver nod should support provided format.
- -F “sip=@PhoneCallsSample-2001.zip” - a parameter that contains a path to the received SIP package. NOTE, that in the below example the SIP package is in the same folder as curl executed.

```
curl -X POST -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H "Accept: application/json" -H "Accept: multipart/form-data" -F "sip=@PhoneCallsSample-2001.zip" -F "format=sip_zip" localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips
```

After the SIP package is received, the response contains a newly created AIP object. The response is very large because the AIP object has many fields. The below example of response payload is truncated to show the relation links on the whole page.

```
{
 "id" : "6728b300-0383-497b-9faa-e205b8433aff",
 "name" : "PhoneCalls-CC-1000000-1",
 "application" : "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424",
 …
 "state" : "Waiting ingestion",
 …
 "_links" : {
  "self" : {
    "href" : "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips/6728b300-0383-497b-9faa-e205b8433aff"
  },
  "http://identifiers.emc.com/ingest" : {
    "href" : "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips/6728b300-0383-497b-9faa-e205b8433aff/ingest"
  }
 }}
```

The “state” attribute and its value “Waiting ingestion” confirms the newly created AIP is ready for ingestion via the http://identifiers.emc.com/ingest link relation, included below.

Ingest Step

From that reception response, it is worth taking the URI for the http://identifiers.emc.com/ingest link relation. It is rooted from the single AIP resource. The URI is used to perform the ingestion by issuing a POST request without any payload.

```
curl -X POST -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..'  -H 'Accept: application/hal+json' localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips/6728b300-0383-497b-9faa-e205b8433aff/ingest
```

After a successful response, the payload contains the same AIP object, but with updated attributes. To be more explicit, we are showing on the below listing only the links relation for the AIP.

The POST request accepts a parameter named **allowBackgroundRequest**. If this parameter is set to true (default value is false) and if the structured database is not available, the ingestion will failback to an asynchronous ingestion. In this case the result will be an AIP in “Pending” State, which will contain a link http://identifiers.emc.com/order-item to the corresponding asynchronous ingestion order item.

Direct Ingestion

When ingesting a large number of SIPs via a batch, it is first necessary to receive all the SIP packages and then ingest them into the application, as described in the previous sections. InfoArchive’ s unitary archiving

allows for simultaneous reception\ingestion of a single SIP via an exposed REST resource, resulting in a reduction of steps.

For performing the direct ingest, leverage the href's value (URI) for the http://identifiers.emc.com/ingest-direct link relation rooted from the AIP's collection resource.

Please note the parameter "**?ingestDirect=true"**.

```
"http://identifiers.emc.com/ingest-direct": {
  "href": "http://localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips?ingestDirect=true"
}
```

The direct ingest is performed by POST request to the above link, but it is required to set up the same parameters that were discussed for Batch Ingestion: -F "format=sip_zip" and -F "sip=@PhoneCallsSample-2001.zip", see below:

```
curl -X POST -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H "Accept: application/json" -H "Accept: multipart/form-data" -F "sip=@PhoneCallsSample-2001.zip" -F "format=sip_zip" localhost:8765/systemdata/applications/7a7b9d20-e31f-4d5c-8bc4-d74de313a424/aips?ingestDirect=true
```

In case of a successful response, you may notice that in the response payload the AIP state is "Completed".

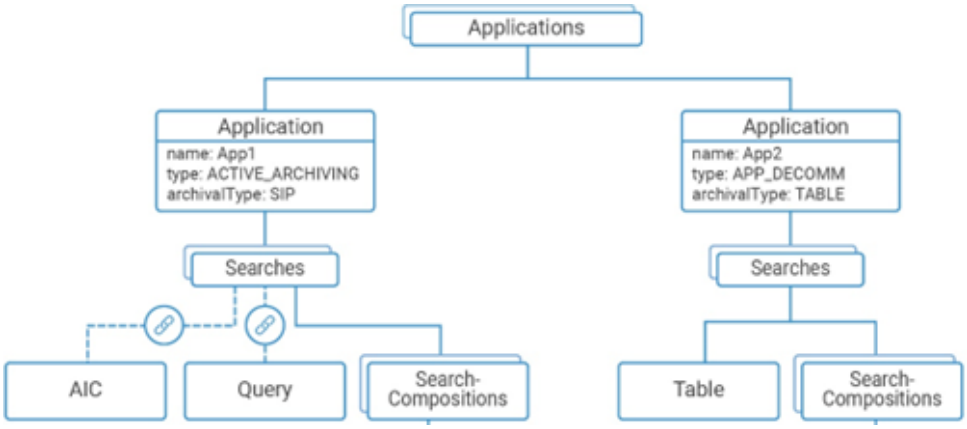
Search

Overview

Search is one of the primary and frequently used resources in InfoArchive. In the resource hierarchy, searches are located under *Home Resource->Tenants->Applications->Searches*)

The search resource is a composite resource that consists of several, independent resources, logically creating one higher level resource. This approach allows for some flexibility.

See the following hierarchy representation, showing similarities and differences, for Searches depending on what application they are associated with. You will notice that there are some differences whether a search is created for application of **archivalType**: TABLE versus application of **archivalType**: SIP – see below:



Each search needs to be associated with a Data Set and, depending on which application a given search is rooted with, that Data Set could be:

- Searches rooted under SIP Application
  - A Data Set is a combination of AIC and Query resources. It is worth pointing out that a search resource does not contain these resources itself, but instead, has links to such. For example, a SIP application has link relations to a collection of AICs and to a collection of Queries, where the objects are persisted and a whole collection can be explored from the root application resource. Moreover, not all combinations of AIC and Query are possible. AIC is bound to a certain number of configured Queries. In the next chapters, it is explained how to find correct AIC and Query pair for the search.
- Searches rooted under Table Application
  - Data Set could be either a Database or Schema/Table. If your search can be narrow down to a single table query, it is best to set both: Schema and Table references when creating search. Setting Table reference is crucial if you want to allow users ability to put holds/retention on the search results (if search has no Table reference set, users will not be able to put search results on holds for example).
  - For cross-table searches only Schema reference should be set as then search is run on the Schema.
  - For cross-schema searches only Database reference should be set.

Determine what searches are available

Effectively, we need to do a GET on the search resources that are available from an application.

To help facilitate this procedure, install the sample Audit application, which has some default searches available.

From the applications link, find the searches link.

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json'
"http://localhost:8765/systemdata/applications/800c33e2-c88e-4bf9-9d9e-d8e7e72b7edd/searches"
```

Search object can be either Primary or Nested. Primary searches can refer to Nested searches (Nested search would typically expend on the results returned to Primary search.

By default, collection of searches returns all searches, regardless of whether they are Primary or Nested.

Search also can be in any of the two states: DRAFT or PUBLISHED. By default, collection of searches returns all searches, regardless of the state.

Table Searches

Step #1
This example leverages the Searches associated with the TABLE Type, “Baseball” application, which needs to be found in the collection. Once found, take note of the link relation “http://identifiers.emc.com/databases and “http://identifiers.emc.com/searches”. We will need both **href** values URIs to be use later in Step #2 and Step #6, respectively.

See example below:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json'
localhost:8765/systemdata/tenants/2f9fd3f2-5334-4f61-8538-9e5595cca4e7/applications
```

Finding the Reference to a Search’s Data Set

Step #2

Each Search needs to be associated with a data set. Baseball is an application of “archiveType TABLE”, so searches associated with the Baseball application will need to have an association with a Schema or both: Schema and a Table.

To find the reference to either one, now we will use our reference to databases (http://identifiers.emc.com/databases) resource found in Step #1:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..'  -H 'Accept: application/hal+json'
localhost:8765/systemdata/applications/e7691d42-b562-4178-9958-e18114ff13b6/databases
```

Step #3

Step #2 returns a collection of databases for the Baseball application, and we’ll process it to find database of interest to us and then examine its **_links** node to find the link relation to schemas (http://identifiers.emc.com/schemas). When we do, we issue a GET on it:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json'
localhost:8765/systemdata/databases/f8fd9488-c3ff-456e-9d64-119a127fb040/schemas
```

Step #4

Step #3 returns a collection of schemas. Find the schema of interest in the array and, for that schema, find the **self**-link relation and its value, and save the value to be used in Step #6 tables (http://identifiers.emc.com/tables).

Step #5

Issue a GET on the value of the **table's** link relation:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json'
localhost:8765/systemdata/schemas/e8309487-0821-4d19-90c3-cbc10d4c2445/tables
```

Now iterate over the collection of tables and find the table of interest, comparing name attributes of each item in the array until we find the one, we want. Let us say we will be looking for table **SALARIES**. When found, store the value of its reference URI (value of **href** attribute on **self**-link relation), since it is needed in **Step #4**.

Getting it All Together

Step #6

Now that we know the reference to our collection of searches (Step #1), we also know the reference to both the Schema (Step #4) and Table (Step #5) we are ready to create our first search.

We will need to do a POST to our collection of searches, and we will pass the search's attributes in a JSON payload.

To create a search for a TABLE archiving application, we need to set the following attributes: name, description (optional), schema and table.

```
curl -X POST -H 'Content-Type: application/json' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept:
application/hal+json'  -d '{"name":"mySampleSearch","description":"my
description","schema":"http://localhost:8765/systemdata/schemas/379762dd-5f49-4964-8c7d-
46fb3bb3a0dd","table":"http://localhost:8765/systemdata/tables/c87656f2-c50c-48dd-86ea-96fff83ecba4"}'
localhost:8765/systemdata/applications/e7691d42-b562-4178-9958-e18114ff13b6/searches
```

SIP Searches

Let us say we are looking for Searches associated with the PhoneCalls application, which is a SIP application. To find the application, we must use the same response, which is received by issuing a GET on the relation link to the collection of applications. This time, we process every item in the response array until we meet application with *name == PhoneCalls.*

Finding the Reference to a Search's Data Set

As we are considering a search for a SIP application type, then its data set should be associated with AIC and Query references. Both resources are mandatory for the search.

Let us try to find a reference for the required single AIC object first. To retrieve an AICs collection, we must issue a GET to the URI for AICs (the URI was found in the previous step).

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..'  -H 'Accept: application/hal+json'
localhost:8765/systemdata/applications/14b5ffbe-e91b-456c-8dc9-5134f7585887/aics
```

The response for that request contains an array of AICs. Assume that the array contains a single element, which is named “PhoneCalls-aic”. What is required for now is to find the **self**-link relation for that item and save its value for a while in notepad or any other desired text editor.

Now it is time to find a reference for a required single Query object.

NOTE: There are two roots from where a collection of Queries can be retrieved: application root and AIC root. Both roots have the same link relation http://identifiers.emc.com/queries, which contains the **href’s** value to the URI for the collection of Queries. It worth pointing out that all Queries that exist in the system can be found from application root.

For defining the search, we need to retrieve and only select the Queries that are bounded with a certain AIC. Otherwise, you may select an inappropriate Query object for that given AIC, which will lead to from search configuration.

That is why we need to perform a GET on the URI from the selected AIC root and find a relation for the collection of queries to take href’s value for bounded queries for that AIC.

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..'  -H 'Accept: application/hal+json'
localhost:8765/systemdata/aics/683b95c0-71f2-45f4-a031-01747ab93f1c/queries
```

Creating a search composition resource

Search-composition is created by issuing a POST on the collection of search-compositions, which are rooted at Search.
In response of newly the created search in a previous step, we will find the link relation search-compositions (http://identifiers.emc.com/search-compositions). We will send a POST to that location. To create a search-composition, a JSON payload is straightforward, as at minimum requires name attribute only.

```
curl -X POST -H 'Content-Type: application/json' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept:
application/hal+json' -d '{"name":"mySearchComposition"}' localhost:8765/systemdata/searches/4d2e0d9a-cff4-
4fda-8d17-e8a7472322a8/search-compositions
```

If the name is unique, that POST should be successful, and we should get back a 201 Created response code and the payload should contain the newly created search-composition resource.

Executing a Synchronous Search

Payload should be enclosed in <data></data> XML nodes. So, you will see that in the example below, in which we set search the input value to **birthYear** 2013-01-01.

```
<data><birthYear>2013-01-01</birthYear><birthMonth>July</birthMonth></data>
```

As you can see, we also set the time out to 300 seconds, which is excessive.

Let us try to execute this now.

```
curl -X POST -H 'Content-Type: application/xml' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json' -d '<data><birthYear>2013-01-01</birthYear></data>' localhost:8765/systemdata/search-compositions/eac888b4-325f-4595-9e5f-bb532b8be504?timeOut=300000
```

Executing an Asynchronous Search

If you recall from the section on synchronous search, we need to **execute** a link relation to search in synchronous mode. There is another link relation, **execute-async**, for running searches in the background. We use it in a similar fashion. Here is an example:

```
curl -X POST -H 'Content-Type: application/xml' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..' -H 'Accept: application/hal+json' -d '<data><birthYear>2013-01-01</birthYear></data>' localhost:8765/systemdata/search-compositions/da1cf6cd-bab3-49f3-b870-25c555336cc8/async?name=myBackgroundSearch
```

Jobs

Job definitions indicate which jobs can be done. Job Instances represent a run of the job. Job instances can be scoped to the system, tenant, or application.

When accessing via rest, the resources are accessed directly of the home resource.

Viewing job definitions

Job definitions can be viewed from following the job-definitions link relation off the home resource:

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9....' -H 'Accept: application/hal+json' -H 'Content-Type: application/hal+json' http://localhost:8765/systemdata/job-definitions
```

View job instances

If the job has been run, there will be job instance created for each run.

```
curl -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9....' -H 'Accept: application/hal+json' -H 'Content-Type: application/hal+json' http://localhost:8765/systemdata/job-definitions/439d7eeb-f662-4d4e-b1c8-f8a3d0c09e91/job-instances
```

Running or scheduling a job

To run or schedule a job, a post is done to the job instances on the job definition you are running. The expectation is that the job definition is not suspended.

Here is an example if you want the job to run now:

```
curl -X POST -H 'Content-Type: application/hal+json' -H 'Authorization: Bearer ...' -H 'Accept: application/hal+json' -d '{now:true}' http://localhost:8765/systemdata/job-definitions/bc37dc50-e956-45db-a4bb-ec118c3a84bf/job-instances
```

The response should be a 201 (CREATED)

About OpenText

OpenText enables the digital world, creating a better way for organizations to work with information, on premises or in the cloud. For more information about OpenText (NASDAQ: OTEX, TSX: OTC) visit opentext.com.

Connect with us:

OpenText CEO Mark Barrenechea's blog

Twitter | LinkedIn