

# Tag Salad

*XML, JSON, Web Data, and Big Data*

**Menu**

---

## EMC Documentum Platform REST Services Tutorial

*Posted on June 4, 2013 by tagsalad*

EMC® Documentum® Platform REST Services will soon be released, and I thought it might be useful to write some simple code to show people how to get started. The following code is based on the soon-to-be-released GA version of the product, not the technology preview that some of you may have used. It uses the *requests* library for all HTTP requests, and the JSON representation of resources. An XML representation of resources is also available. Because Documentum Platform REST Services is a REST API, it is language independent. I will post examples in other languages later. This tutorial is based on data taken from businesses registered in San Francisco, which is stored in a cabinet named “San Francisco”.

## Basic Navigation

First let's look at the code needed to navigate structures in the REST API. Documentum allows rich hierarchies – the service contains a set of repositories, a repository contains a set of cabinets or folders, folders nest, either folders or cabinets can contain documents or sysobjects, a document contains metadata, and can contain a primary representation and multiple renditions, etc. The REST API allows clients to traverse these structures using navigation or queries. If the function *get\_link()* returns a link based on a link relation, the function *get\_repository()* gets a repository with a given name, and the function *get\_atom\_entries()* returns Atom entries from a collection based on their properties, we can navigate from the service entry point to the *tagsalad* repository, from there to the *San Francisco* cabinet, and from there to the documents in the cabinet using the following code, which is explained below:

```
import requests
import json

homeResource = "http://example.com/documentum-rest/serv
drel="http://identifiers.emc.com/linkrel/"
repository = "tagsalad"
cabinet = "San Francisco"
credentials=('tagsalad', 'password')

# Get home resource
home = requests.get(homeResource)
home.raise_for_status()
```

```
# Navigate down to documents for San Francisco cabinet
repositories_uri = home.json()['resources'][drel+'repos
repository = get_repository(repositories_uri, reposit
cabinets_uri = get_link(repository, drel+'cabinets')
sanfran = get_atom_entries(cabinets_uri, 'title="San Fr
documents = get_link(sanfran, drel+'documents')
```

The rest of this section explains the above code in some detail, shows the JSON representation of resources, and discusses some REST design principles. If you are skimming this, or you are familiar with REST, you may skip to the following section.

Still here? Let's look at the URIs and JSON representations used in the above code, then look at the `get_repositories()`, `get_atom_entries()` and `get_link()` functions that it uses.

## The Service Entry Point

Documentum Platform REST Services is a hypermedia-driven API, with a single entry point, which is associated with the home resource. A client retrieves the home resource, located at "`http://example.com/documentum-rest/services`" in the above code, and finds other resources by navigating links or doing queries.

The home resource uses the format defined in [Home Documents for HTTP APIs](#). Here is the JSON structure that is returned by `home.json()`:

```
{
  "resources": {
    "http://identifiers.emc.com/linkrel/repositories": {
      "hints": {
        "allow": [
          "GET"
        ],
        "representations": [
          "application/xml",
          "application/json",
          "application/atom+xml",
          "application/vnd.emc.documentum+json"
        ]
      },
      "href": "http://example.com/documentum-rest/reposit
    },
    "about": {
      "hints": {
        "allow": [
          "GET"
        ],
        "representations": [
          "application/xml",
          "application/json",
          "application/vnd.emc.documentum+xml",
          "application/vnd.emc.documentum+json"
        ]
      }
    }
  }
}
```

```

    },
    "href": "http://example.com/documentum-rest/product-
  }
}
}

```

Now a quick word about link relations. In a hypermedia-driven REST API, the entry point for a REST service must contain links, identified by link relations, that allow a client to navigate to all resources exposed by the service. The link relation is not a physical location, it is a name, encoded as a URI, that identifies the purpose of a link; it is associated with an href that contains the physical address of the link.

Suppose the client has retrieved this home document shown above, and wants to find the URI for the list of repositories. The "resources" entry contains the set of resources, the "http://identifiers.emc.com/linkrel/repositories" link relation identifies a URI for an Atom feed containing repository entries, and the "href" entry contains the URI for this Atom feed, which is "http://example.com/documentum-rest/repositories".

In Python, if *home* contains the result of a GET request that retrieved the home document, the following expression returns this URI:

```
home.json()['resources']['http://identifiers.emc.com/li
```

## Atom Feeds and EDAA Feeds

Collections are represented using Atom feeds in the XML representation. EDAA is a JSON representation of an Atom feed, and functions essentially the same way. Here is the EDAA feed associated with "http://example.com/documentum-rest/repositories".

```

{
  "id": "http://10.37.10.115:8080/dctm-rest/repositor
  "title": "Repositories",
  "updated": "2013-06-05T15:13:47.969-07:00",
  "author":
  [
    {
      "name": "EMC Documentum"
    }
  ],
  "total": 2,
  "links":
  [
    {
      "rel": "self",
      "href": "http://www.example.com/documentum-
    }
  ],
  "entries":
  [
    {

```

```

    "id": "http://www.example.com/documentum-re
    "title": "tagsalad",
    "summary": "Tag Salad",
    "content":
    {
        "content-type": "application/vnd.emc.do
        "src": "http://www.example.com/document
    },
    "links":
    [
        {
            "rel": "edit",
            "href": "http://www.example.com/doc
        }
    ]
},
{
    "id": "http://www.example.com/documentum-re
    "title": "acme",
    "summary": "ACME Company, Arizona Desert Re
    "content":
    {
        "content-type": "application/vnd.emc.do
        "src": "http://www.example.com/document
    },
    "links":
    [
        {
            "rel": "edit",
            "href": "http://www.example.com/doc
        }
    ]
}
]
}

```

Each entry represents one repository. We want the repository with the title ‘tagsalad’. In the above feed, each entry contains only a small subset of the content of a repository resource, we want to retrieve the entire resource, found at the URI in the *id* entry, which is "http://www.example.com/documentum-rest/repositories/tagsalad".

To retrieve an entry by title, we will use the following code, which is explained below.

```

def get_repository(repositories_uri, name):
    """ Get the repository with the specified name """
    for repository in get_atom_entries(repositories_uri
        if repository['name'] == name:
            return repository
    return None

def get_atom_entries(feed_uri, filter=None, default=None):
    """Get matching entries from an Atom feed."""
    if filter:
        params = {'inline': 'true', 'filter': filter }

```

```

else:
    params = {'inline': 'true' }
    response = requests.get(feed_uri, params=params, au
    response.raise_for_status()
    return [ e['content'] for e in response.json()['ent

repository = get_repository(repositories_uri, reposit

```

The first parameter for the `get_repository()` function is the URI of an Atom feed. The function does a GET on this URI, returning the feed. (Actually, Atom is defined using an XML representation, in JSON we use EDAA feeds, which are similar to Atom feeds but use a JSON representation.) The `get_repository` function calls `get_atom_entries()` to return a list of the repositories contained in the feed. It then searches for a repository with the specified name and returns it.

The `get_atom_entries()` function returns a list of entries contained in a feed. It sets the *inline* parameter to *true* so the resulting entries contain the entire resource, not just the summary used when *inline* is set to *false*. A function that returns hundreds or thousands of items and lets a user select one might prefer *inline=false*, so that all available choices can quickly be listed without the overhead of fetching all properties for each object.

The return expression contains the following list comprehension:

```
[ e['content'] for e in response.json()['entries'] ]
```

The list comprehension creates a list of all entries in the feed.

For collections within a given repository, filters can be used to specify conditions that are used to select results, using a subset of XPath. For instance, a repository contains cabinets, so we can find the “San Francisco” cabinet using the following code:

```
cabinet = get_atom_entries(cabinets_uri, 'title="San Fr
```

## Documentum Resources and Link Relations

Home documents, Atom feeds, and EDAA feeds are generic resources. Documentum Platform REST Services has a number of resources that it defines, each has a JSON representation and an XML representation. The JSON representation for these resources is defined in the media type `application/vnd.emc.documentum+json`. The XML representation is defined in the media type `application/vnd.emc.documentum+xml`.

In the last section, we retrieved a repository resource, which illustrates how link relations appear in Documentum resources. A repository contains a number of links that can be used to find many kinds of information associated with a repository, including the users, the current user, user groups, cabinets, formats, network locations, relations, relation types, checked out objects, and types. It can also be used to do DQL queries.

Here is the JSON representation of a repository:

```

{
  "id": 23,
  "name": "tagsalad",
  "description": "Tag Salad",
  "servers":
  [
    {
      "name": "tagsalad",
      "host": "rest_api_cs70_1",
      "version": "7.0.0000.0125 Win64.SQLServer",
      "docbroker": "rest_api_cs70_1"
    }
  ],
  "links":
  [
    {
      "rel": "self",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/",
      "href": "http://example.com/documentum-rest"
    }
  ]
}

```

```

    "href": "http://example.com/documentum-rest
  },
  {
    "rel": "http://identifiers.emc.com/linkrel/
    "hreftemplate": "http://example.com/documen
  }
]
}

```

Each link contains a "rel", which identifies the purpose of the link, and an "href", which provides the location of the resource. For instance, the following entries indicate the location of the cabinets feed, identified by the "http://identifiers.emc.com/linkrel/cabinets" link relation.

```

{
  "rel": "http://identifiers.emc.com/linkrel/cabinets"
  "href": "http://example.com/documentum-rest/reposito
}

```

If *repository* contains the repository shown in this section, the following code returns the URI for the cabinets feed:

```

def get_link(e, linkrel, default=None):
    return [ l['href'] for l in e['links'] if l['rel']

cabinets_uri = get_link(repository, 'http://identifiers

```

The *get\_link()* function returns the link associated with a link relation in a resource. If the link relation is not present, an error is raised. In Python this can be done in a single line, which uses a list comprehension to create a list that contains all link relations that match the property, then returns the first entry (there will never be more than one entry matching a given link relation).

## From Home Resource to Documents

Now you should be able to understand the code at the beginning of the navigation section.

Three kinds of resources are used. The first resource is a home resource. Here is the code that retrieves the home resource and finds the repositories URI in it:

```

homeResource = "http://example.com/documentum-rest/serv
drel="http://identifiers.emc.com/linkrel/"

home = requests.get(homeResource)
home.raise_for_status()
repositories_uri = home.json()['resources'][drel+'repos

```

The second kind of resource is an EDAA feed, which is the JSON equivalent of an Atom feed. The following code retrieves a repository from an EDAA feed using the title of the corresponding entry:

```
repository = get_repository(repositories_uri, repositior
cabinets_uri = get_link(repository, drel+'cabinets')
```

The following code retrieves a cabinet from the cabinets feed, then retrieves the URI of the documents feed from it:

```
sanfran = get_atom_entries(cabinets_uri, 'title="San Fr
documents = get_link(sanfran, drel+'documents')
```

## Reading Entries

Now that we have found a collection of documents, let's read each document and print its title and a few other properties.

Let's start simple. If the number of documents is small enough to fit on one page (by default, 100 entries, but you can set this with a URI parameter), and you only print properties that are present in the Atom feed when *inline=false*, this is very easy. If *documents* contains the URI of the documents feed, the following code does the trick.

```
r = requests.get(documents, auth=credentials)
for e in r.json()['entries']:
    p = e['content']['properties']
    print ( p['title'])
```

Now let's add two complications. First, let's print a property that is only available when *inline=true*. Second, let's assume that results may span many pages, following the *next* link relation as long as it is present to make sure that we print all results. This can be done with the following code:

```
documents = get_link(sanfran, drel+'documents')

# Read all documents, paging as needed
while True:
    response = requests.get(documents, params='inline=t
    response.raise_for_status()
    for e in response.json()['entries']:
        p = e['content']['properties']
        print ( p['object_name'], ' ', p['title'])
    try:
        documents = get_link(response.json(), 'next')
    except:
        break
```

The first part of the while loop is identical to our previous example. After printing the items on the given page, it looks for a *next* link relation that contains the URI for the next page. If it does not find a *next* link relation, it knows that it has read all pages in the collection.

## Filters, Sorting, and Paging



When searching for the San Francisco cabinet, we used a simpler filter, which has a language based on a subset of XPath. Here is an example of a slightly more complex filter:

```
contains(object_name, "COFFEE") and r_modify_date >= da
```

Conditions can be combined using and or or, parenthesis can be used to group conditions, not() can be used to negate conditions, and comparisons can be made using =, <, >, <=, >=, or !=. The functions date(), starts-with(), and contains are supported.

A filter always returns an entire object or document, but a view can be used to specify a set of properties that should be returned. Sort order can also be specified, as can the number of items on a page.

The following code shows how these URI parameters can be combined in the parameter list.

```
params = {
    'inline' : True,
    'sort' : 'object_name',
    'view' : 'object_name,title',
    'filter' : 'contains(object_name, "COFFEE") and r_m
    'items-per-page' : 50
}

response = requests.get(documents, params=params, auth=
response.raise_for_status()
for e in response.json()['entries']:
    print (prettyprint(e))
```

## Writing Entries

Entries can be written to an Atom feed using POST, using the content-type 'application/vnd.emc.documentum+json' if the body contains a JSON representation, or 'application/vnd.emc.documentum+xml' if the body contains an XML representation. At least the object name and the object type must be specified. In a real application, we might create an object type that allows us to represent the fields of a given kind of document. To keep it simple, we will abuse the title to contain the address of a business.

```
body = json.dumps(
{
    "properties" : {
        "object_name" : "Earthquake McGoons",
        "r_object_type" : "dm_document",
        "title" : "50 California Street, 94111"
    }
})
```

```
headers = { 'content-type' : 'application/vnd.emc.docum
```

```
response = requests.post( documents, data=body, headers
```

If the POST succeeds, the status code is 201 CREATED and *response* contains the newly created document resource.

```
>>> response = requests.post( documents, data=body, hea
>>> response.status_code
201
>>> response.raise_for_status()
>>> response.reason
'Created'
>>> response.json()
{'name': 'document', 'properties': {    !!! SNIP !!!
```

## Updating Entries

An update is done using a POST to the URI of the object that is to be updated, using the content-type 'application/vnd.emc.documentum+json' if the body contains a JSON representation, or 'application/vnd.emc.documentum+xml' if the body contains an XML representation. If the POST is successful, it returns a representation of the updated resource. Suppose *response.json()* contains a document. The following code changes a property in the document and updates it in the repository.

```
document = response.json()
document['properties']['object_name'] = 'Kilroy was Her
headers = { 'content-type' : 'application/vnd.emc.docum
uri = get_link(document, 'edit')
response = requests.post(uri, json.dumps(document), hea
```

Note that the above code uses the *edit* link relation, an IANA standard link relation that allows a resource to be read, updated, or deleted.

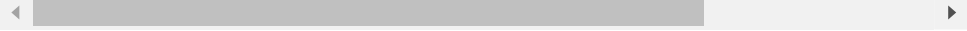
If the POST succeeds, the status code is 200 OK and *response* contains the updated document resource.

```
>>> response.status_code
200
>>> response.reason
'OK'
>>> response.raise_for_status()
>>> response.json()
{'name': 'document', 'properties': {    !!! SNIP !!!
```

## Deleting Entries

Deleting an entry is done using a DELETE to the URI of the object that is to be updated.

```
response = requests.delete(get_link(document, 'edit'),
```



The response contains the HTTP status code, with no JSON content.

```
>>> response.status_code
204
>>> response.reason
'No Content'
>>> response.raise_for_status()
>>>
```

Advertisements

Share this:

Twitter

Facebook

*Tagged Documentum, Python, REST*

---

*Previous Post*

## Introducing Tag Salad

---

## 2 thoughts on “EMC Documentum Platform REST Services Tutorial”



**EMC Documentum REST Services 7 | dm\_misc: Miscellaneous Documentum Information**

*July 14, 2013 Reply*

[...] and Documentum XML for non-collection resources). Here is the developer's guide, and a tutorial. REST Services 7.0 is compatible with Content Server 7.0, 6.7 SP2, and 6.7 (no 6.7 [...]

**Documentum releases REST resources | Architecture, programming and APIs**

*July 16, 2013 Reply*

[...] Also, please checkout Jonathan's blog on using our REST here [...]