EMC® Documentum® Platform REST Services

Version 7.2

Development Guide

EMC Corporation
Corporate Headquarters
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Documentation Feedback

Your opinion matters. We want to hear from you regarding our product documentation. If you have feedback about how we can make our documentation better or easier to use, please send us your feedback directly at ECD.Documentation.Feedback@emc.com

Table of Contents

List of Figures

List of Tables

Preface

This document is a guide to using EMC Documentum REST Services to interact with Documentum repositories.

Intended Audience

This document is intended for developers and architects who are building Documentum REST Services.

Related Documentation

The following documentation provides additional information:

- *EMC Documentum Platform REST Services Resource Reference Guide*
- *EMC Documentum Platform REST Services Release Notes*

Conventions

The following conventions are used in this document:

Font Type	Meaning
italic	Book titles, emphasis, or placeholder variables for which you supply particular values
`monospace`	Commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter

Terminologies

Documentum REST MVC	Built on top of Spring MVC, Documentum REST MVC is a framework that facilitates the custom resource development in Documentum REST Services. For more information, see Documentum REST MVC.
Core resources	Any resources that are shipped with EMC Documentum Platform REST Services out of the box. Appendix C provides a complete list of Core resources. Additionally, the *EMC Documentum Platform REST Services Resource Reference Guide* introduces each Core resource in detail.

Acronyms and Abbreviations

REST	Representational state transfer
XML	eXtensible Markup Language
JSON	JavaScript Object Notation
HATEOS	Hypermedia as the Engine of Application State

Revision History

The following changes have been made to this document.

Revision history

Revision Date	Description
February 2016	Added note to section 5.2.3.2.1 Register and Map the Service Principal Name.
July 2015	Added JBoss Enterprise Application Platform to the certified application server list.
February 2015	Initial publication.

Chapter 1

Overview

EMC Documentum Platform REST Services, which is also referred to as Documentum REST Services, is a set of RESTful web service interfaces that interact with the Documentum platform.

Being developed in a purely RESTful style, Documentum REST Services is hypertext-driven, server-side stateless, and content negotiable. This provides you with high efficiency and simplicity in programming, and it also makes all services easy to consume. These advantages make Documentum REST Services the optimal choice for Web 2.0 applications and mobile applications to interact with Documentum repositories.

Documentum REST Services models objects in Documentum repositories as resources and identifies resources by Uniform Resource Identifiers (URIs). It defines specific media types to represent resources and drives application state transfers by using link relations. It uses a limited number of HTTP standard methods (GET, PUT, POST, and DELETE) to manipulate resources over the HTTP protocol.

Documentum REST Services supports two formats for resource representation:

- JSONJavaScript Object Notation (JSON) is a lightweight data interchange format based on a subset of the JavaScript Programming Language standard. Documentum REST Services uses JSON as the primary format for representing resources.
- XMLXML is the dominant data format in traditional SOAP based Web Services, yet as well as being widely used in RESTful Web Services. Documentum REST Services supports two kinds of XML formats: XML-based Atom (See RFC4287) and Documentum XML. Atom is an XML-based document format that describes collections of related information known as "feeds". Documentum REST Services represents collection-based resources in Atom feeds, and represents non-collection based resources in Documentum XML documents.

Understanding RESTful Programming

Documentum REST Services delivers a deployable Java web archive (WAR) file that runs in a web container of Java EE application server (refer to the release notes for system requirements). It exposes the interface as network-accessible resources identified by URIs. Documentum REST Services is programming language independent. Therefore, you can consume the services by using any language that has an HTTP client library, such as Java, .NET, Python, Ruby, and so on. You can take full freedom and ownership to develop the REST client to consume REST Services if you follow hypertext-driven principles. Furthermore, Documentum REST Services ships an SDK. The SDK contains development libraries, toolkits, and code samples enabling you to extend Documentum REST Services by composing, customizing, and creating new RESTful resources in an easy and efficient way.

Relations with Other Documentum Platform APIs

Documentum REST Services relies on the DFC library to communicate with Documentum Content Server, and thus the communication between the REST server and Content Server is conducted over Netwise RPC. Documentum REST Services is a lightweight alternative to the existing Documentum Platform Web APIs, such as WDK and DFS. However, it is not intended to provide the equivalent functionalities in this release. You can leverage the simplicity of RESTful services to achieve high productivity in software development.

Chapter 2

Deploy Documentum REST Services

Documentum REST Services has been certified on the following application servers:

- Apache Tomcat
- VMware vFabric tc Server
- Oracle WebLogic Server
- IBM Websphere
- JBoss Enterprise Application Platform

For detailed information about how to deploy Documentum REST Services, see the *EMC Documentum REST Services Release Notes*.

Chapter 3

General REST Definitions

Common Definition - HTTP Headers

Documentum REST Services supports the following common HTTP headers:

HTTP Header Name	Description	In Request or Response?	Value Range

Authorization	Authorization header for authentication	Request	HTTP basic authentication header with the credential part encoded, for example:
			`Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`
			Or, Kerberos authentication header with the credential part encoded, for example:
			`Authorization: Negotiate YIIZG1hZG1pbjpwYXNzd29yZ...`

Accept	Acceptable media type for the response	Request	See Supported MIME Types, page 23
Content-Type	MIME type of the request body or response body	Request /Response	See Supported MIME Types, page 23 The REST server ignores the `charset` parameter in the Content-Type header. Therefore, the error `E_INPUT_MESSAGE_NOT_READABLE` occurs when you send a POST operation that contains non-utf8 characters in the request body.
Content -Length	Size of the entity-body, in decimal number of OCTETs, sent to the recipient	Request /Response	Non-negative number
Location	URI of the newly-created resource	Response	URI

HTTP Header Name	Description	In Request or Response?	Value Range
Set-Cookie	Sets an HTTP cookie Response	Response	Used by client token
WWW -Authenticate	Indicates the authentication scheme that should be used to access the requested entity	Response	Used by HTTP basic authentication and HTTP Kerberos negotiate authentication Set-Cookie.
ETag	ETag value generated by the REST server	Response	String value
If-None-Match	Checks whether the resource is changed	Request	ETag value in the previous server's response.
Last-Modified	Last modified time of the resource	Response	HTTP-Date
If-Modified -Since	Checks whether the resource is changed since last modified time	Request	Last-Modified value in the previous server's response.

Common Definition - Query Parameters

Documentum REST Services supports the following common query parameters:

Note: For a specific resource, it may not support all the following query parameters. For detailed information about what query parameters a resource supports, see the Query Parameters section of that resource.

Query Parameter Name	**Description**	**Data type**	**Value Range**	**Default**
inline	Determines whether or not to show content (the object instance) in an atom entry or EDAA entry for a collection.	boolean	• `true` - return the object instance and embed the object instance into the entry's `content` element. • `false` - do not return object instance.	false **Note:** Although the default value is `false`, we still recommend that you try setting this parameter to `true` in your development environment to get the entire content of resources and explore the complete data structure. In your production environment, you can set this parameter to `false` for better performance.
items-per-page	Specifies the number of items to be rendered on one page.	integer	Any integer no less than 1	100
page	Specifies the page number of the page to return. If you set `items-per-page` to 200, and `page` to 2, the operation returns items 201 to 400.	integer	Any integer no less than 1	1
view	Specifies the object properties to retrieve. This parameter works only when `inline` is set to `true`.	string	See Property View, page 49.	:default

Query Parameter Name	Description	Data type	Value Range	Default
include -total	Determines whether or not to return the total number of objects. For paged feeds, objects in all pages are counted.	boolean	• `true` - return the total number of objects. • `false` - do not return the total number of objects.	false
sort	Specifies a set of the sort specifications in a returned collection.	string	This parameter consists of multiple sort specifications, separated by comma (`,`). Each sort specification consists of a property (any non-repeating property) by which to sort the results and its sort order (`DESC` or `ASC`), separated by the whitespace character ( ). The sort order is optional. if not specified, the default sort order is `ASC`. If any property with an invalid name is specified, an error is thrown. **Example**: `sort=r_modify_date desc,object_id asc,title`	NA
links	Determines whether or not to return link relations in the object representation This parameter works only when `inline` is set to `true`.	boolean	• `true` - return link relations. • `false` - do not return link relations.	true

Query Parameter Name	Description	Data type	Value Range	Default
recursive	Determines whether or not to return all indirect children recursively when a request tries to get the children of an object.	boolean	• `true` - return all indirect children recursively when a request tries to get the children of an object. • `false` - Only return direct children when a request tries to get the children of an object.	false
filter	Filter expression	string	Filter Expression, page 41	NA
q	Specifies full text search criterion when sending a GET request to the Search resource or certain collection resources	string	String that follows the simple search language syntax	NA

HTTP Status Codes

Documentum REST Services supports the following HTTP status codes:

Status Code	Description
200 OK	The request has succeeded. The information returned with the response is dependent on the method used in the request, for example: GET an entity corresponding to the requested resource is sent in the response; PUT an entity describing or containing the modified resource.
201 Created	The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field. The entity format is specified by the media type given in the Content-Type header field.
204 No Content	The server has fulfilled the request but does not need to return an entity-body.
304 Not Modified	The server responds with this status code if the client has performed a conditional GET request and access is allowed, but the document has not been modified.
400 Bad Request	The request could not be understood by the server due to malformed syntax, missing or invalid information (such as a validation error on an input field, or a missing required value). The client should not repeat the request without modifications.
401 Unauthorized	The authentication credentials included with this request are missing or invalid. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.
403 Forbidden	The server has recognized your credentials, but you do not possess the authorization to perform this request, and the request should not be repeated.
404 Not Found	The request specifies a URI of a resource that does not exist.
405 Method Not Allowed	The HTTP verb specified in the request (DELETE, GET, HEAD, POST, PUT) is not allowed for the resource identified by the request URI.
406 Not Acceptable	The resource identified by this request URI is not capable of generating a representation corresponding to one of the media types in the Accept header of the request.
409 Conflict	The request could not be completed, because it would cause a conflict in the current state of the resources supported by the server (for example, an attempt to create a new resource with a unique identifier already assigned to some existing resource).
415 Unsupported Media Type	The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.
500 Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.

Supported MIME Types

Adhering to RFC 2616, section 14 for content negotiation, Documentum REST Services supports XML and JSON representations with the following MIME types:

MIME Type	Usage
application/atom+xml	XML representation for feeds (collections)
application/vnd.emc.documentum+xml	XML representation for a single object
application/vnd.emc.documentum+json (default MIME type)	JSON representation
application/json	JSON representation for compatible viewing
application/xml	XML representation for compatible viewing

Note:

- If the client does not specify the Accept header for the expecting response, the REST server uses default MIME type `application/vnd.emc.documentum+json`.
- If the client does not specify the Content-Type header for a PUT or POST request, the REST server rejects the request with the HTTP status code 415.
- In some scenarios, the MIME types that Documentum REST Services supports are not limited to the types in this table. For more information about other supported types, see Other Types, page 23.

Other Types

Media Type for Home Document

The Home Document resource uses `application/home+json` and `application/home+xml` as its media types.

MIME Type for Content

For the content import or export operations, the REST server accepts any MIME type registered to the `dm_format` table in a repository. For example, if the REST client imports a new PDF rendition for a document object, the Content-Type in the request body can be `application/pdf`.

Multipart Type

Documentum REST Services supports multipart type in contentful SysObject importing and the `multipart/related` type for batch operations with content. For more information about multipart type, see RFC 2616, section 3.7.2 on www.ietf.org.

URL Extension

In Documentum REST Services, you can use the URL extensions `.xml` and `.json` to negotiate the content type of the response. See the following examples:

- `/repositories.xml` returns a collection of repositories in an ATOM XML feed representation.
- `/repositories/acme01.xml` returns the repository acme01 in an XML representation.
- `/repositories.json` returns a collection of repositories in a JSON representation.
- `/repositories/acme01.json` returns the repository acme01 in a JSON representation.

When you use a URL extension in a GET operation, the URL extension will take precedence over the Accept header. The content type of the return varies with the URL extension as follows:

URL Extension	Content Type of the Return
.json	application/json
.xml	application/xml
other extension	The Accept header is used for content negotiation. For more information about the Accept header and content negotiation, see RFC2616, section 14.1:http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.1

URL extensions in POST and PUT operations are ignored. The REST server reads the media type from the Content-Type header.

Content Type of an Entry in a Feed

If the `inline` parameter is set to `false` when an operation tries to retrieve a feed, the operation returns the content type for an entry. The entry's content type is decided by the content type of the feed that contains the entry. For details, see the following table.

Feed Content Type	Entry Content Type
application/atom+xml	application/vnd.emc.documentum+xml

Feed Content Type	Entry Content Type
application/xml	application/xml
application/vnd.emc.documentum+json	application/vnd.emc.documentum+json
application/json	application/json

URI and URL

A Uniform Resource Identifier (URI) is a compact sequence of characters that can identify an abstract or physical resource. In Documentum REST Services, URIs are the only resource identifiers.

In Documentum REST Services, a resource is located by a uniform resource locator (URL). The URL is designed in a pattern that is only known to and interpretable by the REST server. REST clients cannot parse or concatenate the URL for a resource by using the object ID or name. All REST clients can follow the link relations on a resource representation to locate the related resources.

URLs in the resource representations are in the form of an absolute path. If the REST server is deployed behind a reverse proxy server or a load balancer, you must configure the proxy rules to replace the backend hostname with the front hostname. For example, the following reverse proxy configuration in the Apache HTTP server maps `http://internal-node1.acme.com:8080/dctm-rest` to `http://reverse-proxy-server:80`.

```
ProxyPass / http://internal-node1.acme.com:8080/dctm-rest/
ProxyPassReverse / http://internal-node1.acme.com:8080/dctm-rest/

<Location/>
AddOutputFilterByType SUBSTITUTE application/xml
Substitute "s|internal-node1.acme.com:8080/dctm-rest| reverse-proxy-server |i"
</Location>
<Location/>
AddOutputFilterByType SUBSTITUTE application/json
Substitute "s|internal-node1.acme.com:8080/dctm-rest| reverse-proxy-server |i"
</Location>
<Location/>
AddOutputFilterByType SUBSTITUTE application/atom+xml
Substitute "s|internal-node1.acme.com:8080/dctm-rest| reverse-proxy-server |i"
</Location>

<Location/>
AddOutputFilterByType SUBSTITUTE application/vnd.emc.documentum+xml
Substitute "s|internal-node1.acme.com:8080/dctm-rest| reverse-proxy-server |i"
</Location>

<Location/>
AddOutputFilterByType SUBSTITUTE application/vnd.emc.documentum+json
Substitute "s|internal-node1.acme.com:8080/dctm-rest| reverse-proxy-server |i"
</Location>

<Location/>
AddOutputFilterByType SUBSTITUTE text/html
Substitute "s|internal-node1.acme.com:8080/dctm-rest| reverse-proxy-server |i"
</Location>
```

For more information about URI encoding in Documentum REST Services, see the following RFC document:

http://tools.ietf.org/html/rfc3986

HTTP Methods

Documentum REST Services supports the following HTTP methods:

- GET Use this method to retrieve a representation of a resource.
- POST Use this method to create new resources, or update existing resources.
- PUT Use this method to update existing resources.
- DELETE Use this method to delete a resource.

Note: Documentum REST Services does not support the OPTIONS method. Do not use this method to request a list of available operations on a resource. The *EMC Documentum Platform REST Services Resource Reference Guide* provides detailed information about the operations you can perform on each resource.

Web Client Caching

Caching is one of the most useful features built on the top of the HTTP's uniform interface. HTTP clients can take advantage of caching to reduce the perceived latency to users, increase reliability, reduce bandwidth usage and cost, and reduce server load. Documentum REST Services supports web client caching on the content media resource using an HTTP ETag token.

Representation

Documentum REST Services supports two representation formats, JSON and XML.

JSON is the primary format for resource representation. Collection-based resources are presented as EDAA, which is a JSON representation of an Atom feed. For more information about EDAA, see the *Platform REST Services EMC Data Access API*.

JSON supports basic data types. Therefore, Documentum properties are mapped to JSON data types as shown in the following table:

Table 1. Documentum Property to JSON Data Type

Documentum Property Data Type	JSON Data Type
Boolean	Boolean
Integer	Number
Double	Number

Documentum Property Data Type	JSON Data Type
String	String
ID	String
Time	String
Repeating	Array

XML is the other format for resource representation in Documentum REST Services. Like the JSON format, collection-based resources are presented as feeds, defined by Atom. For more information about Atom, see RFC42870.

http://tools.ietf.org/html/rfc4287

Collection Resource

In both formats, items (object instances) in the collection are represented as entries containing metadata and links in the feed. By default, the detail of an object instance is not presented in the entry body. Instead, a content `src` link points to the single instance resource. Alternatively, the object instance can be embedded in the entry by enabling `inline`.

The metadata of a feed consists of the following elements/properties:

Table 2. Metadata of a Feed

Feed Metadata	Description
id	URI of the collection resource without the file extension
title	Feed Title
updated	Last update time of the feed
author	Feed author
self link	URI of the collection resource with the file extension

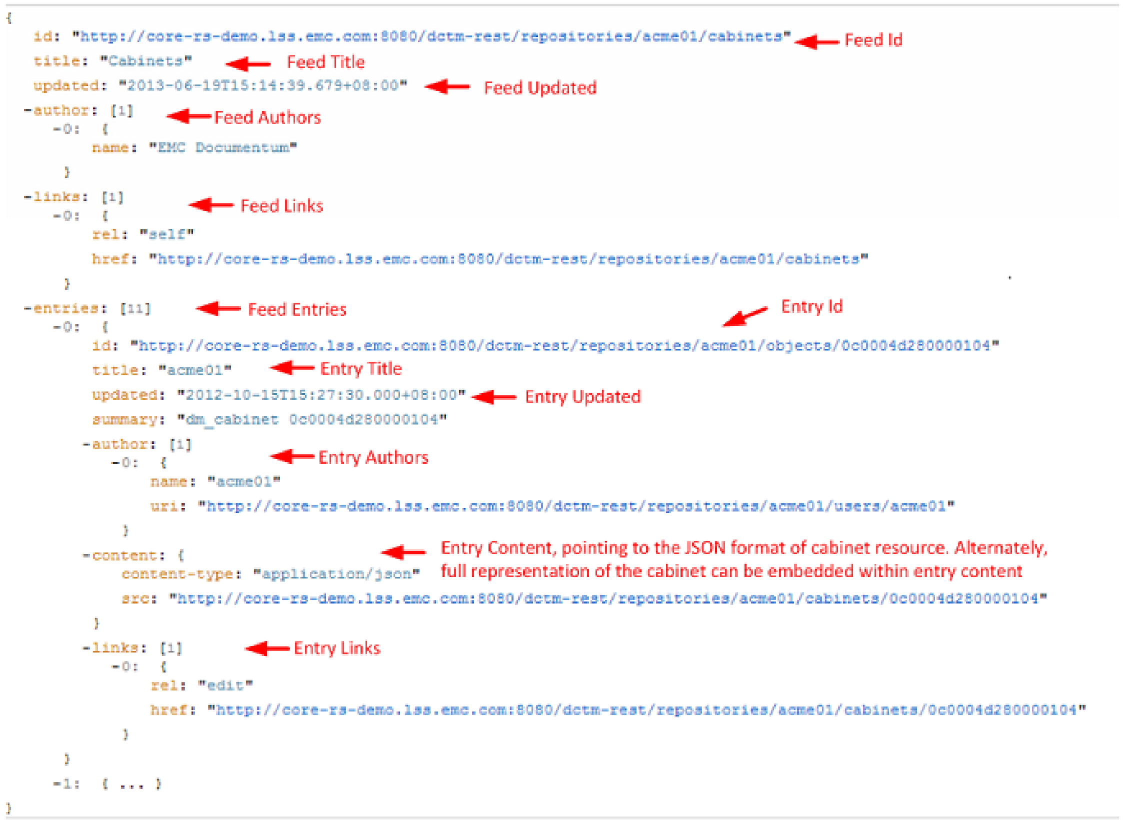
The metadata of an entry consists of the following elements/properties:

Table 3. Metadata of an Entry

Feed Metadata	Description
id	URI of the single resource without the file extension
title	Entry title
updated	Last update time of the entry
author	Entry author
summary	Entry description

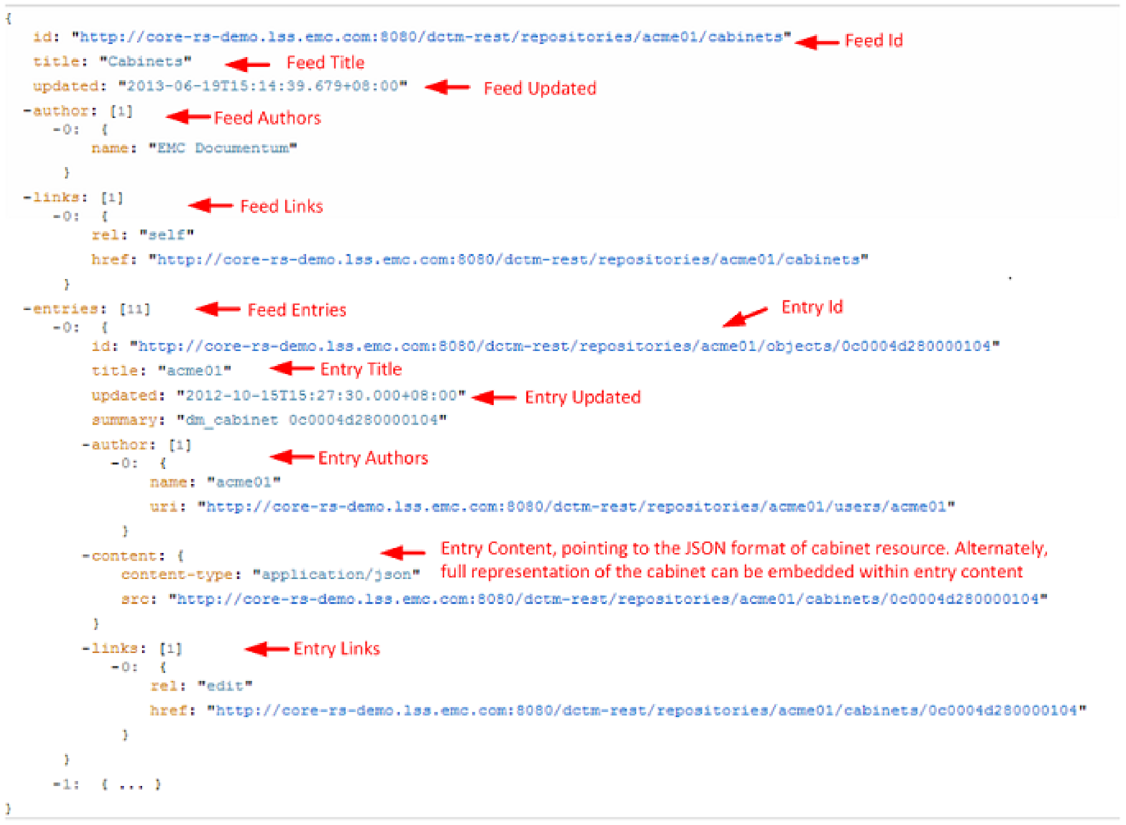
Feed Metadata	Description
content	URI of the single resource with the file extension, or a full representation of the resource. See Embedded Entry, page 30. If you set `inline` to `false`, `content-type` in JSON and `type` in XML indicate the MIME type of the resource.
edit link	URI of the single resource with the file extension

The following sample illustrates a feed (EDAA feed) of the Cabinets Resource in JSON. Note that only a URI of the single resource is presented in this sample as `inline` is not enabled.


Feed Id
Feed Title
Feed Updated
Feed Authors
Feed Links
Feed Entries
Entry Id
Entry Title
Entry Updated
Entry Authors
Entry Content, pointing to the JSON format of cabinet resource. Alternately, full representation of the cabinet can be embedded within entry content
Entry Links

```
{
   id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets"
   title: "Cabinets"
   updated: "2013-06-19T15:14:39.679+08:00"
  -author: [1]
     -0:  {
           name: "EMC Documentum"
         }
  -links: [1]
     -0:  {
           rel: "self"
           href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets"
         }
  -entries: [11]
     -0:  {
           id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/objects/0c0004d280000104"
           title: "acme01"
           updated: "2012-10-15T15:27:30.000+08:00"
           summary: "dm_cabinet 0c0004d280000104"
          -author: [1]
             -0:  {
                   name: "acme01"
                   uri: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/users/acme01"
                 }
          -content: {
              content-type: "application/json"
              src: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
           }
          -links: [1]
             -0:  {
                   rel: "edit"
                   href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
                 }
         }
     -1:  { ... }
}
```

The following sample illustrates a feed of the Cabinets Resource in XML. Note that only a URI of the single resource is presented in this sample as `inline` is not enabled.


Feed Id
Feed Title
Feed Updated
Feed Authors
Feed Links
Feed Entries
Entry Id
Entry Title
Entry Updated
Entry Authors
Entry Content, pointing to the JSON format of cabinet resource. Alternately, full representation of the cabinet can be embedded within entry content
Entry Links

```
{
   id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets"
   title: "Cabinets"
   updated: "2013-06-19T15:14:39.679+08:00"
  -author: [1]
     -0:  {
           name: "EMC Documentum"
        }
  -links: [1]
     -0:  {
           rel: "self"
           href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets"
        }
  -entries: [11]
     -0:  {
           id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/objects/0c0004d280000104"
           title: "acme01"
           updated: "2012-10-15T15:27:30.000+08:00"
           summary: "dm_cabinet 0c0004d280000104"
          -author: [1]
             -0:  {
                   name: "acme01"
                   uri: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/users/acme01"
                }
          -content: {
              content-type: "application/json"
              src: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
           }
          -links: [1]
             -0:  {
                   rel: "edit"
                   href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
                }
        }
     -1:  { ... }
}
```

Embedded Entry

For collection resources, when the `inline` parameter is set to true, a full representation of the object instance is embedded in the content element of the entry.

The following sample illustrates an entry that contains a full representation of cabinet in JSON.

```
"entries": [
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme01/
      objects/0c0004d280000104",
      "title": "acme01",
      "updated": "2012-10-15T15:27:30.000+08:00",
      "author": [
        {
          "name": "acme01",
          "uri": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
          users/61636d653031"
    }
      ],
      "content": {
        "name": "cabinet",
        "type": "dm_cabinet",
        "definition": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/
        acme01/types/dm_cabinet",
        "properties": {
          "r_object_id": "0c0004d280000104",
          "object_name": "acme01",
          "title": "Super User Cabinet",
          "subject": "",
          "resolution_label": "",
          "owner_name": "acme01",
          "owner_permit": 7,
          "group_name": "docu",
          "group_permit": 5,
          "world_permit": 3,
          "log_entry": "",
          "acl_domain": "acme01",
          "acl_name": "dm_450004d280000100",
          "language_code": "",
          "r_object_type": "dm_cabinet",
          "r_creation_date": "2012-10-15T15:27:30.000+08:00",
          "r_modify_date": "2012-10-15T15:27:30.000+08:00",
          "a_content_type": "",
          "authors": null,
          "r_lock_owner": "",
          "i_antecedent_id": "0000000000000000",
          "i_chronicle_id": "0c0004d280000104",
          "i_folder_id": null,
          "i_cabinet_id": "0c0004d280000104"
    },
        "links": [
          {
            "rel": "self",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            cabinets/0c0004d280000104.json"
      },
          {
            "rel": "edit",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            cabinets/0c0004d280000104.json"
      },
          {
            "rel": "http://identifiers.emc.com/documentum/linkrel/delete",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            cabinets/0c0004d280000104.json"
      },
          {
            "rel": "http://identifiers.emc.com/documentum/linkrel/folders",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            folders/0c0004d280000104/folders.json"
      },
          {
            "rel": "http://identifiers.emc.com/documentum/linkrel/documents",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            folders/0c0004d280000104/documents.json"
      },
          {
            "rel": "http://identifiers.emc.com/documentum/linkrel/objects",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            folders/0c0004d280000104/objects.json"
      },
          {
            "rel": "http://identifiers.emc.com/documentum/linkrel/child-links",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            folders/0c0004d280000104/child-links.json"
      },
          {
            "rel": "http://identifiers.emc.com/documentum/linkrel/relations",
            "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
            relations.json?related-object-id=0c0004d280000104&related-object-role=any"
      }
        ]
  },
      "links": [
        {
          "rel": "edit",
          "href": "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
          cabinets/0c0004d280000104.json"
    }
      ],
      "thumbnail": {
        "url": "http://localhost:8081/thumbsrv/getThumbnail?object_type=dm_cabinet
        &format=&is_vdm=false&repository=1234"
  }
},
….
```

The following sample illustrates an entry that contains a full representation of cabinet in XML.

```
<entry>
  <id>
  http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/objects/
    0c0004d280000104
  </id>
  <title>acme01</title>
  <updated>2012-10-15T15:27:30.000+08:00</updated>
  <author>
   <name>acme01</name>
   <uri>
   http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/users/
       61636d653031
   </uri>
  </author>
  <content>
   <dm:cabinet xsi:type="dm:dm_cabinet" definition="http://core-rs-demo.lss.emc.co
      m:8080/dctm-rest/repositories/acme01/types/dm_cabinet">
   <dm:properties xsi:type="dm:dm_cabinet-properties">
    <dm:r_object_id>0c0004d280000104</dm:r_object_id>
    <dm:object_name>acme01</dm:object_name>
    <dm:title>Super User Cabinet</dm:title>
    <dm:subject/>
    <dm:resolution_label/>
    <dm:owner_name>acme01</dm:owner_name>
    <dm:owner_permit>7</dm:owner_permit>
    <dm:group_name>docu</dm:group_name>
    <dm:group_permit>5</dm:group_permit>
    <dm:world_permit>3</dm:world_permit>
    <dm:log_entry/>
    <dm:acl_domain>acme01</dm:acl_domain>
    <dm:acl_name>dm_450004d280000100</dm:acl_name>
    <dm:language_code/>
    <dm:r_object_type>dm_cabinet</dm:r_object_type>
    <dm:r_creation_date>2012-10-15T15:27:30.000+08:00</dm:r_creation_date>
    <dm:r_modify_date>2012-10-15T15:27:30.000+08:00</dm:r_modify_date>
    <dm:a_content_type/>
    <dm:authors xsi:nil="true"/>
    <dm:r_lock_owner/>
    <dm:i_antecedent_id>0000000000000000</dm:i_antecedent_id>
    <dm:i_chronicle_id>0c0004d280000104</dm:i_chronicle_id>
    <dm:i_folder_id xsi:nil="true"/>
    <dm:i_cabinet_id>0c0004d280000104</dm:i_cabinet_id>
   </dm:properties>
   <dm:links>
    <dm:link rel="self" href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/
         repositories/acme01/cabinets/0c0004d280000104" />
    <dm:link rel="edit" href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/
         repositories/acme01/cabinets/0c0004d280000104" />
    <dm:link rel="http://identifiers.emc.com/documentum/linkrel/delete"
```

```
         href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
         cabinets/0c0004d280000104" />
    <dm:link rel="http://identifiers.emc.com/documentum/linkrel/folders"
         href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
         folders/0c0004d280000104/folders" />
    <dm:link rel="http://identifiers.emc.com/documentum/linkrel/documents"
         href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/
         acme01/folders/0c0004d280000104/documents" />
    <dm:link rel="http://identifiers.emc.com/documentum/linkrel/objects"
         href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/
         acme01/folders/0c0004d280000104/objects" />
    <dm:link rel="http://identifiers.emc.com/documentum/linkrel/child-links"
         href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/
         acme01/folders/0c0004d280000104/child-links" />
    <dm:link rel="http://identifiers.emc.com/documentum/linkrel/relations"
         href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/
         relations?related-object-id=0c0004d280000104&related-object-role=any" />
   </dm:links>
  </dm:cabinet>
  </content>
  <link rel="edit" href= "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/
   acme01/cabinets/0c0004d280000104" />
   <media:thumbnail url="http://localhost:8081/thumbsrv/getThumbnail?object_type=
   dm_cabinet&format=&is_vdm=false&repository=1234"/>
 </entry>
...
```

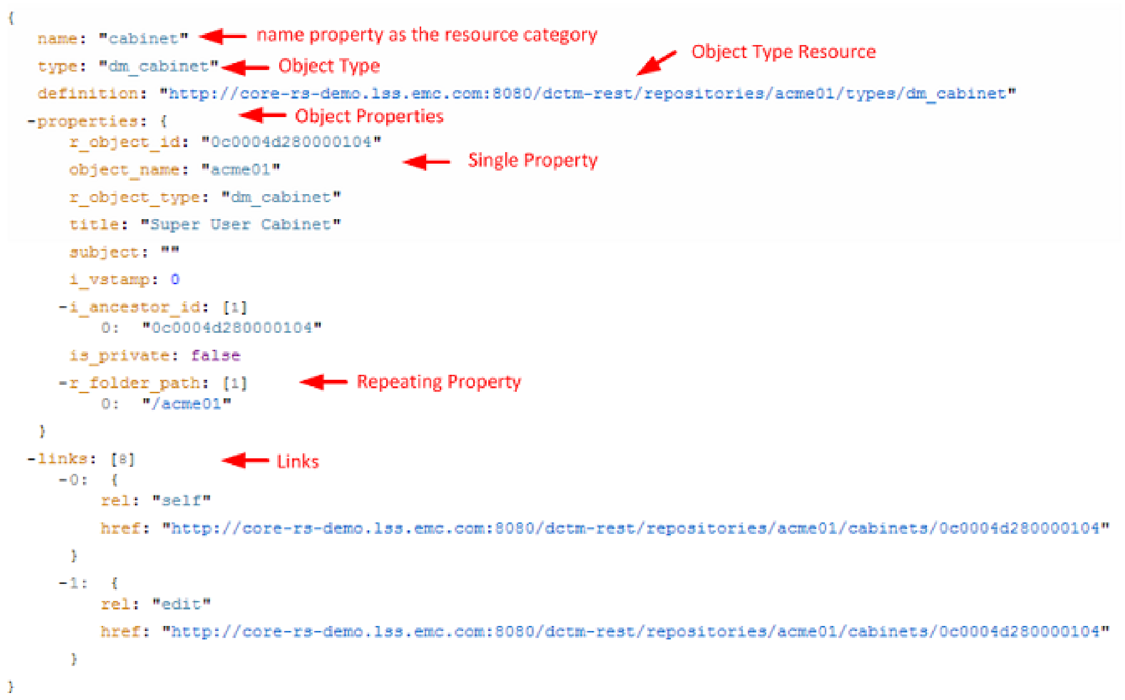
Single Resource

The representation of a single resource consists of the following elements/properties:

Table 4. Elements/Properties in a Single Resource

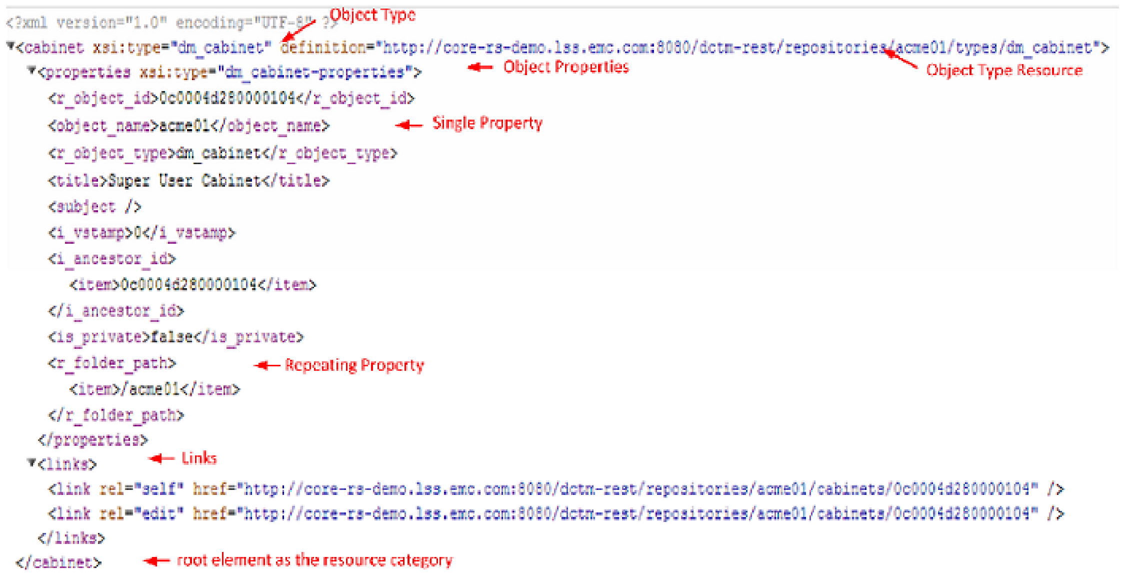
Element in XML	Property in JSON	Description
root	name	Category of the resource. For more information about categories, see Appendix B, Category for Single Resources.
xsi:type	type	Type name
definition	definition	URI pointing to the DML representation of the type
properties	properties	Properties of the resource
links	links	Link relations of the resource

The following sample illustrates the Cabinet resource in JSON.



```
{
   name: "cabinet"
   type: "dm_cabinet"
   definition: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/types/dm_cabinet"
  -properties: {
      r_object_id: "0c0004d280000104"
      object_name: "acme01"
      r_object_type: "dm_cabinet"
      title: "Super User Cabinet"
      subject: ""
      i_vstamp: 0
     -i_ancestor_id: [1]
         0:  "0c0004d280000104"
      is_private: false
     -r_folder_path: [1]
         0:  "/acme01"
   }
  -links: [8]
     -0:  {
         rel: "self"
         href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
      }
     -1:  {
         rel: "edit"
         href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
      }
}
```

The following sample illustrates the Cabinet resource in XML.



```
<?xml version="1.0" encoding="UTF-8" ?>
▼<cabinet xsi:type="dm_cabinet" definition="http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/types/dm_cabinet">
  ▼<properties xsi:type="dm_cabinet-properties">
     <r_object_id>0c0004d280000104</r_object_id>
     <object_name>acme01</object_name>
     <r_object_type>dm_cabinet</r_object_type>
     <title>Super User Cabinet</title>
     <subject />
     <i_vstamp>0</i_vstamp>
     <i_ancestor_id>
       <item>0c0004d280000104</item>
     </i_ancestor_id>
     <is_private>false</is_private>
     <r_folder_path>
       <item>/acme01</item>
     </r_folder_path>
   </properties>
  ▼<links>
     <link rel="self" href="http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104" />
     <link rel="edit" href="http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104" />
   </links>
 </cabinet>
```

Not all resources have the same structure of the representation. For example, the Repository resource representation does not have the `type` or `properties` keys.

Multi-part Request Representation

Some operations may use multi-part requests. This section provides examples of multi-part requests in JSON and XML.

Example 3-1. JSON representation in multi-part request

```
POST http://localhost/rest-api-web/repositories/myrepo/
folders/0c00000c80000105/documents
Content-Type: multipart/form-data; boundary=314159265358979

  --314159265358979
  Content-Disposition: form-data; name=metadata
  Content-Type: application/vnd.emc.documentum+json

  {"properties": {"object_name": "rest-api-test"}}
  --314159265358979
  Content-Disposition: form-data; name=binary1
  Content-Type: text/plain

  This is primary content
  --314159265358979--
```

Example 3-2. XML representation in multi-part request

```
POST http://localhost/rest-api-web/repositories/myrepo/
objects/0c00000c80000105/objects
Content-Type: multipart/form-data; boundary=314159265358979

--314159265358979
Content-Disposition: form-data; name=metadata
Content-Type: application/vnd.emc.documentum+xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<dm:dm_document xmlns:dm="http://identifiers.emc.com/documentum">
        <properties>
          <dm:object_name>hello</dm:object_name>
        <properties>
</dm:dm_document>
--314159265358979
Content-Disposition: form-data; name=binary
Content-Type: text/plain

This is a sample
--314159265358979--
```

Error Representation

An error representation contains the following items:

- HTTP Status Code (mandatory) - identical to the status code in the header
- REST Error Code (mandatory) - REST application-specific code
- REST Error Message (mandatory) - descriptive message for the error code
- Root Causes (optional) - a set of error code/message mappings of the under layer

Documentum REST Services supports the JSON and XML formats in error responses.

Example 3-3. XML Error Representation

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<error>
 <status type="integer">xxx</status>
 <code>xxx</code>
 <message>xxx</message>
 <details>xxx</details>
</error>
```

Example 3-4. Sample 1. Get an object with an invalid ID

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<error>
 <status type="integer">404</status>
 <code>E_RESOURCE_NOT_FOUND</code>
 <message>The resource is not found.</message>
 <details>(DM_API_E_EXIST) Document/object specified by 0b00000c80000dda
 does not exist;
 Cannot fetch a sysobject - Invalid object ID : 0b00000c80000dda;</details>
</error>
```

Example 3-5. Sample 2. Create a folder with an empty input message

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<error>
 <status type="integer">400</status>
 <code>E_VALIDATION_ATTR_MISSING</code>
 <message>Input message contains invalid items.</message>
 <details>Properties 'r_object_type'  should not be null or empty;
 Properties 'object_name'  should not be null or empty;</details>
```

```
</error>
```

Example 3-6. JSON Error Representation

```
{
 "status":xxx,
 "code":"xxx",
 "message":"xxx",
 "details:"xxx"
}
```

Example 3-7. Sample 1. Get an object with an invalid ID

```
{
 "status":404,
 "code":"E_RESOURCE_NOT_FOUND",
 "message":"The resource is not found.",
 "details":"(DM_API_E_EXIST) Document/object specified by 0b00000c80000dda
 does not exist;
 Cannot fetch a sysobject - Invalid object ID : 0b00000c80000dda;"
}
```

Example 3-8. Sample 2. Create folder with empty input message

```
{
 "status":400,
 "code":"E_VALIDATION_ATTR_MISSING",
 "message":"Input message contains invalid items.",
 "details":"Properties 'r_object_type'  should not be null or empty;
 Properties 'object_name' should not be null or empty;"
}
```

Transaction Support

Documentum REST Services provides basic transaction support for copy and delete operations. The transaction begins at the start of an operation and is committed at the end of the operation if it completes successfully. If any part of the operation fails, the entire operation is rolled back.

In a batch request, you can control the transaction behavior of the batch by using the transactional property.

Notes to PUT and POST Operation

When you perform a PUT operation or POST operation to update a resource, changes to the root element and links are ignored. If you try to update the namespace or read-only properties, an error is returned.

When performing a POST operation to create a resource, you can use the `type` property in JSON or the `xsi:type` property in XML to specify the object type of the resource. For Sysobject and its sub types, you can also use the `r_object_type` property to specify the object type. If the value of `type` (or `xsi:type`) and that of `r_object_type` are not consistent, the value of `type` (or `xsi:type`) takes precedence.

Runtime Property Configuration

Runtime property configuration files enable you to set preferences for how Documentum REST Services handles certain choices in the course of its execution, for example, setting authentication schemes and specifying the default page size. REST Services leverages two configuration files for runtime property setting. Both files are located in `dctm-rest.war\WEB-INF\classes`.

`rest-api-runtime.properties.template`
This file holds the default settings for all runtime properties. Do not modify the content in this file.

`rest-api-runtime.properties`
Out of the box, this file contains no settings, indicating that all properties use their default values. When you need to modify the value of a certain runtime property, add an entry in this file, specifying the name and value of the property. Settings in this file override the settings in `rest-api-runtime.properties.template`.

Batch Operations

Starting from release 7.2, you can leverage the newly added Batches collection resource to execute a series of RESTful Web service operations in one request. This resource also provides a number of batch options such as transactional, sequential, and so on. Furthermore, Documentum REST Services provides the Batch Capabilities resource for you to check the list of resources that are batchable at runtime.

For more information, see the Batches and Batch Capabilities section in the *EMC Documentum Platform REST Services Resource Reference Guide*.

Chapter 4

Resource Specific Features

Filter Expression

A filter expression, which is the value of the `filter` URI parameter, enables you to filter entries in a collection of results according to the specified criterion.

This section describes the building blocks of a filter expression, including:

- Literals, page 41, which describes the literal formats in filter expressions.
- Functions, page 44, which describes the functions in filter expressions.
- Logical Operators, page 47, which describes the logical operators supported by filter expressions.
- Comparison Operators, page 47, which describes the comparison operators supported by filter expressions.

Note: All operator names and function names are case sensitive.

Literals

Literals are values that are interpreted by the server exactly as they are entered. Filter expressions introduce three types of literals:

- Numeric Literal, page 41
- String Literal, page 42
- Boolean Literal, page 42
- Datetime Literal, page 43

Numeric Literal

Filter expressions support integer literals and floating point literals.

An integer literal specifies any whole number and is expressed in the following format:

```
[+ | -] n
```

where `n` is any number between 0 and 2,147,483,647.

A floating point literal specifies any number that contains a decimal point and is expressed in the following format:

- 5.347 (regular floating point literal)
- -4.12 (negative floating point literal)
- 21. (floating point literal with a blank fractional part)
- .66 (floating point literal with a blank integer part)

String Literal

String literals are strings of printable characters and are enclosed in a pair of single quotes or a pair of double quotes.

If a string literal is enclosed in a pair of single quotes, the double quote character (") can be included as a part of the literal without any change. For example:

```
'foo"bar'
```

Similarly, if a string literal is enclosed in a pair of double quotes, the single quote character (') can be included as a part of the literal without any change. For example:

```
"The company's third quarter results were very good."
```

However, to include a single quote character (') as a part of a literal that is enclosed in a pair of single quotes, you must include the single quote character twice. For example:

```
'The company''s third quarter results were very good.'
```

Similarly, to include a double quote character (") as a part of a literal that is enclosed in a pair of double quotes, you must include double quote character twice. For example:

```
"foo""bar"
```

The maximum length of a string literal is determined by the maximum allowed by the underlying RDBMS, but in no case will the maximum length exceed 1,999 bytes. If a property is defined as a string datatype, the maximum length of the string literal you can place in the property is defined by the property's defined length. If you attempt to place a longer value in the property, DFC throws an exception. You can change the behavior and allow DFC to truncate the character string value to fit the property by setting the dfc preference called `dfc.compatibility.truncate_long_values` in `dfc.properties`.

Boolean Literal

Boolean literals specify constant values for the true and false values used in filter expressions. There are two Boolean literal values: `true` and `false`.

Datetime Literal

A datetime literal represents a date or a combined date and time representation, which is enclosed in a pair of single quotes or a pair of double quotes, using one of the following syntax:

- `date ("YYYY-MM-DD")`
- `date ("YYYY-MM-DDThh:mm:ss.[sss][TZD]")`

Where:

- YYYY = four-digit year
- MM = two-digit month (01=January, etc.)
- DD = two-digit day of month (01 through 31)
- hh = two digits of hour (00 through 23) (am/pm NOT allowed)
- mm = two-digit of minute (00 through 59)
- ss = two-digit of second (00 through 59)
- sss = three-digit millisecond. (000 through 999, an optional field which is ignored when being processed. This is because Content Server does not store the millisecond field in a datetime property.)
- The optional field TZD represents the time zone designator:
  - If this field is set to the special UTC designator ("Z") or unspecified, the time is expressed in UTC (Coordinated Universal Time).
  - The time is expressed in local time, together with a time zone offset in hours and minutes, which represents the difference between the local time and UTC.

Note:

- If a date without time is entered, the time `00:00:00` is assumed.
- In SQL Server, both `date ("1970-01-01T00:00:00.000+0000")` and `date ("1753-01-01T00:00:00.000+0000")` are taken as a null datetime.
- In Oracle, `date ("0001-01-01T00:00:00.000+0000")` is taken as a null datetime.

Example 4-1. Examples for datetime literals

- `date ("2013-01-01")`

  Because the time field is not specified. The time `00:00:00` is assumed. This literal equals to `date ("2013-01-01T00:00:00Z").`
- `date ('2007-07-16T19:20:30.45')`

  Because the TZD field is not specified, this datetime literal is expressed in UTC. This literal equals to `date ('2007-07-16T19:20:30.45Z').`
- `date ("2007-07-16T19:20:30.45Z")`
- `date ("2007-07-16T19:20:30.45+08:00")`
- `date ('2007-07-16T19:20:30.45-03:00')`

Functions

Functions are operations on values. Filter expressions introduce the following functions:

starts-with

The `starts-with` function checks the starting string of a property.

This function is a Boolean term that returns `True` if the specified property starts with a specified literal string, and `False` otherwise.

Syntax:

```
starts-with(<property-name>, "<string-literal>")
```

Arguments:

- `property-name`: Specifies the property whose starting string this function tests.
- `string-literal`: String literal with which the starting literal of the specified property is compared.

Example 4-2. Example for starts-with

```
starts-with(object_name, "foo")
```

By using this filter expression, the request only returns objects whose `object_name` starts with `foo`.

contains

The `contains` function checks whether or not a property contains a specified literal string.

This function is a Boolean term that returns `True` if the property contains the specified string, and `False` otherwise.

Syntax:

```
contains(<property-name>, "<string-literal>")
```

Arguments:

- `property-name`: Specifies the property that this function tests.
- `string-literal`: String literal with which the specified property is compared.

Example 4-3. Example for contains

```
contains(object_name, 'hello')
```

By using this filter expression, the request only returns objects whose `object_name` contains `hello`.

between

The `between` function checks whether or not the value of a property lies between a specified range.

This function is a Boolean term that returns `True` if the property lies between the specified range, and `False` otherwise.

Syntax:

```
between(<property-name>, from, to)
```

Arguments:

- `property-name`: Specifies the property that this function tests.
- `from`: Lower value in the range this function evaluates. This argument can be one of the following types:
  - Datetime
  - Numeric
- `to`: Higher value in the range this function evaluates. This argument can be one of the following types:
  - Datetime
  - Numeric

Note: All arguments in this function must be of the same type.

Example 4-4. Examples for between

```
between(r_link_cnt, 1, 5)
```

By using this filter expression, the request returns objects whose `r_link_cnt` is no less than 1 and no greater than 5.

```
between(r_modify_date, date("2013-01-16"), date("2013-01-18")
```

By using this filter expression, the request only returns objects whose `r_modify_date` is no earlier than 2013-01-16 and no later than 2013-01-18.

type

The `type` function checks whether or not an object is an instance of a specified type or its subtype.

This function is a Boolean term that returns `True` if the object is an instance of the specified type or its subtype, and `False` otherwise.

Syntax:

```
type(<type-name>)
```

Argument:

- `type-name`: Specifies the type that this function uses as a filter.

Example 4-5. Example for type

```
type(dm_folder)
```

By using this filter expression, the request only returns objects of the `dm_folder` type and objects of any subtype of `dm_folder`.

nilled

The `nilled` function checks the nullity of a property.

This function is a Boolean term that returns `True` if the value of the specified property is null, and `False` otherwise.

Syntax:

```
nilled(<property-name>)
```

Argument:

- `property-name`: Specifies the property that this function tests.

Example 4-6. Example for type

```
nilled(object_name)
```

By using this filter expression, the request only returns objects whose `object_name` is null.

Logical Operators

Logical operators apply to Boolean terms and return Boolean values. The following logical operators are supported in filter expressions.

- `not`
- `and`
- `or`

These operators follow the standard logical semantics. They are listed in order of precedence, with the highest-precedence one at the top.

Comparison Operators

Comparison operators compare one expression to another. Filter expressions support two sets of comparison operators: value comparison operators and general comparison operators. These two sets of operators function the same except for note [1] in the following table.

Table 5. Comparison Operators

Value comparison operator	General comparison operator	Description	Example

eq [1]	=	Equal	`object_name = "REST"`
			`object_name eq REST"`

ne	!=	Not equal	`object_name != "REST"`
			`object_name ne "REST"`

lt	<	Less than	`r_modify_date < "2013-01-16"`
			`r_modify_date lt "2013-01-16"`

le	<=	Less than or equal to	`r_full_content _size <= 2000`
			`r_full_content _size le 2000`

gt	>	Greater than	`r_full_content _size > 2000`
			`r_full_content _size gt 2000`

Value comparison operator	General comparison operator	Description	Example

ge	>=	Greater than or equal to	`r_modify_date >= "2013-01-16"`
			
			`r_modify_date ge "2013-01-16"`

[1] This operator cannot be used to compare a property with a list of values. For more information about how to compare the value of a property with a list of values, see Comparison with Multiple Values, page 48.			

Comparison with Multiple Values

The = comparison operator allows you to compare the value of a property with a list of values. When a filter expression compares the value of a property with multiple literals by using the = comparison operator, the expression returns True if the value equals to any of the literals, and False otherwise.

Syntax

```
<property-name> = ("literal1", " literal2", "literal3" … )
```

Example 4-7. Example for Comparison with Multiple Values

```
object_name = ("foo", "bar")
```

Note:

- The list of literals must be enclosed in a pair of brackets.
- The `eq` comparison operator does not support this function.

Comparison with Repeating Properties

The following syntax enables you to check whether or not any item in a repeating property matches the comparison.

```
<repeating-property-name> /item <comparison-operator> <literal-or-value
-list>
```

The expression returns True if any value of the repeating property matches the comparison, and False otherwise.

Example 4-8. Example for Comparison with Repeating Properties

- `keywords/item = "hey"`
- `keywords/item = ("hello", "world")`

Filter Expression Examples

This section provides comprehensive examples that demonstrate uses of filter expressions.

Example 4-9. Example 1

```
type(dm_document) and between(r_modify_date, date("2013-01-16"),
date("2013-01-18"))
```

With this expression, the request only returns instances of `dm_document` that were modified between 2013-01-16 and 2013-01-18.

Example 4-10. Example 2

```
not(starts-with(object_name, "foo"))
```

With this expression, the request filters out resources whose `object_name` starts with `foo`.

Example 4-11. Example 3

```
(contains(object_name, 'hello') or contains(object_name, 'hey')) and
r_modify_date le date("2013-01-01")
```

With this expression, the request returns resources whose `object_name` contains `hello` or `hey`. Additionally, resources that were modified later than 2013-01-01 are filtered out.

Example 4-12. Example 4

```
author != "Jack" and keywords/item = ("hello", "world")
```

With this expression, the request returns resources that have the keyword `hello` or `world`. Additionally, resources whose author is Jack are filtered out.

Property View

You can use the property view to specify the object properties to retrieve in an operation. This can be done by creating a view expression in the `view` query parameter. A view expression can either be a predefined view expression or a custom view expression that contains a list of property names.

Predefined View Expression

Documentum REST Services supports the following predefined view expressions:

View expression	Description
:all	Returns all properties of the requested object to the client. If you use this view expression, the performance may degrade when a request tries to retrieve a collection resource.
:default	Returns the default set of properties of the requested object to the client. This is the default value if there is no value specified for the `view` query parameter.

Custom View Expression

A custom view expression contains a list of property names of the properties that you want to retrieve separated by commas (,).

A property name must observe the following rules:

- The maximum allowed length is 27 characters.
- The first character must be a letter. The remaining characters can be letters, digits, or underscores.
- The name cannot contain spaces or punctuation.
- The name cannot be any of the words reserved by the underlying RDBMS.

Note: Property names are not case sensitive.

If a property name violating any of the rules above is specified in the expression, the view expression is considered as invalid and rejected. In this case, the server returns the `HTTP 400 Bad Request` status.

Except for the rules above, you cannot specify a custom property in a custom view expression when trying to get a collection resource. Otherwise, the `HTTP 400 Bad Request` status is returned. For example, GET requests to a URL that resembles the following are regarded as bad requests:

```
/repositories/repository/folders/folderID/documents?inline=true
&view=r_object_id,custom_property_name
```

To retrieve custom properties in a collection resource, set the view expression to `:all`.

View Expression Syntax

The syntax of a view expression is defined with the following EBNF:

```
view-expression = predefined-view | properties-list ;
predefined-view = leading-separator, predefined-view-name ;
leading-separator = ":" ;
predefined-view-name = "default" | "all" | "none" ;
properties-list = property-name, { "," property-name } ;
property-name = character, 26 * [ character ] ;
character = letter | digit | underscore ;
```

```
letter = "a" .. "z" | "A" .. "Z" ;
digit = "0" .. "9" ;
underscore = "_" ;
```

Example 4-13. Predefined view example

```
view=:all
```

This example sets the `view` parameter to the predefined view expression `:all`, meaning that all properties of the requested object are returned.

Example 4-14. Custom view example

```
view=object_name,r_object_type
```

This example sets the `view` parameter to the custom view expression `object_name,r_object_type`, meaning that only the `object_name` and `r_object_type` properties of the requested object are returned.

NULL in REST

Documentum REST Services sets the value of a single property to the corresponding DFC default value when the property is set to NULL. For details, see the following table.

Documentum Datatype	JSON/XML Datatype	DFC Default Value
DM_BOOLEAN	Boolean	false
DM_INTEGER	Number	0
DM_DOUBLE	Number	0.0
DM_STRING	String	""
DM_ID	String	"0,000,000,000,000,000"
DM_TIME	String	"nulldate"

For repeating properties, setting NULL for a property or leaving a property empty removes all values from the repeating property.

NULL Value Representation

In a response that is returned from the REST server, NULL or empty values are represented as follows:

Example 4-15. NULL value for Datetime in XML

```
<dm:property_name xsi:nil="true"/>
```

Example 4-16. NULL value for Datetime in JSON

```
"property_name":null
```

Example 4-17. NULL value for String in XML

```
<dm:property_name/>
```

Example 4-18. NULL value for String in JSON

```
"property_name":""
```

Example 4-19. NULL value for repeating properties in XML

```
<dm:xxx xsi:nil="true"/>
```

Example 4-20. NULL value for repeating properties in JSON

```
"property_name":null
```

Note: The examples above show how the REST server handles NULL values in a response. REST clients are free to use any valid format to represent NULL values in a request. For example, you can use this pattern to represent a NULL value in XML:

```
<dm:property_name><dm:property_name/>
```

For atom feeds, if a property is set to NULL or empty, the property is not displayed in the response.

Whitespace in XML

Documentum REST Services supports whitespace characters. You can use the CDATA section or the `xml:space="preserve"` property to preserve whitespace characters. For details, see the following examples:

Example 4-21. Preserve whitespace by using the CDATA section

```
<dm:dm_document xmlns:dm="http://identifiers.emc.com/documentum">
  <dm:properties>
    <dm:object_name><![CDATA[    ]]></dm:object_name>
  </dm:properties>
</dm:dm_document>
```

Example 4-22. Preserve whitespace by using the xml:space property

```
<dm:dm_document xmlns:dm="http://identifiers.emc.com/documentum">
  <dm:properties>
    <dm:object_name xml:space="preserve"> </dm:object_name>
  </dm:properties>
</dm:dm_document>
```

In a response, the REST server only uses the `xml:space="preserve"` property to preserve whitespace characters.

Note: Even if all whitespace characters are processed by Documentum REST Services correctly, some of them can be ignored. This is a limitation in Content Server.

Thumbnail Link

Thumbnails on atom feed entries help mobile clients preview the documents within a collection resource. Thumbnail links are available to the following collection resources:

- Cabinets
- Folder Child Documents
- Folder Child Objects
- Folder Child Folders
- Checked Out Objects

For more information about these resources, see the *EMC Documentum Platform REST Services Resource Reference Guide*.

To enable thumbnail links, the following conditions must be met:

- The Thumbnail server must be installed.
- The `thumbnail` query parameter is set to `true`.

When retrieving these collection resources, you can use the `thumbnail` query parameter to determine whether or not to return the thumbnail link for each entry.

Variable	Description	Data type	Default value
thumbnail	Specifies whether or not to return the thumbnail link for each entry. • `true` - Return the thumbnail link for each entry in the collection resource. • `false` - Do not return the thumbnail link for each entry in the collection resource.	boolean	Value of the `rest.entry.thumbnail.default` parameter in the `rest-api-runtime.properties` file, which defaults to `false`.

Example 4-23. Thumbnail link in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
      xmlns:dm="http://identifiers.emc.com/documentum"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <id>/repositories/acme01/folders/0c0004d280000d1f/documents</id>
    <title>Folder child objects</title>
    <updated>2013-05-08T23:53:09.035-0700</updated>
    <author>
        <name>EMC Documentum</name>
    </author>
    <link rel="self"
    href="/repositories/acme01/folders/0c0004d280000d1f/documents.xml"/>
    <entry>
        <id>/repositories/acme01/objects/090004d280005123</id>
        <title>testOverview.ppt</title>
        <updated>2013-05-07T01:16:36.000-0700</updated>
        <author>
            <name>dmadmin</name>
            <uri>/repositories/acme01/users/646d61646d696e</uri>
        </author>
        <content type="application/xml"
        src="/repositories/acme01/documents/090004d280005123.xml"/>
        <link rel="edit"
        href="/repositories/acme01/documents/090004d280005123.xml"/>
        <link rel="icon" href="/thumbsrv/getThumbnail?path=000004d2\80
        \00\00\5f.jpg&store=thumbnail_store_01"/>
    </entry>
    <entry>...</entry>
</feed>
```

Example 4-24. Thumbnail link in JSON

```
{
    "id":"/repositories/acme01/folders/0c0004d280000d1f/documents",
    "title":"Folder child objects",
    "updated":"2013-05-08T23:54:25.165-0700",
    "author":[
        {
            "name":"EMC Documentum"
    }
    ],
    "links":[
        {
            "rel":"self",
            "href":"/repositories/acme01/folders/0c0004d280000d1f/documents.json"
    }
    ],
    "entries":[
    {
        "id":"/repositories/acme01/objects/090004d280005123",
        "title":"testOverview.ppt",
        "updated":"2013-05-07T01:16:36.000-0700",
        "author":[
            {
                "name":"dmadmin",
                "uri":"/repositories/acme01/users/646d61646d696e"
        }
        ],
        "content":{
            "content-type":"application/json",
            "src":"/repositories/acme01/documents/090004d280005123.json"
    },
        "links":[
            {
                "rel":"edit",
                "href":"/repositories/acme01/documents/090004d280005123.json"
        },
            {
                "rel":"icon",
                "href":"/thumbsrv/getThumbnail?path=000004d2\80\00\00\
                5f.jpg&store=thumbnail_store_01"
        }
        ]
}]
}
```

Feed Pagination

Paged feeds can be useful when the number of entries is very large or indeterminate. To save bandwidth, clients navigate through the feed and only access a subset of the feed's entries as necessary.

Documentum REST Services utilizes the following query parameters for feed pagination (For detailed information, see Common Definition - HTTP Headers, page 17):

- page
- items-per-page
- include-total

If all results can fit on one page, the response is not paged. In this case, the response does not contain the `page` and `items-per-page` fields even if they are specified in the request. Also, the pagination link relations (`first`, `last`, `previous` and `next`) are not available.

For a paged feed, the response must have at least one of the following link relations:

- firstThis link relation is always available.
- lastThis link relation is available when the size of the entire feed is known. This condition is met in the following scenarios:
  - The `items-per-page` query parameter specified is larger than the number of entries in the response for current page.
  - The client sets the `include-total` query parameter to calculate the number of entries of the feed.
  - The current page is the last page of the feed.
- previous: This link relation is available when the current page is not the first page.
- next: This link relation is available when the current page is not the last page or when the size of the feed is unknown.

For detailed information about these link relations, see Appendix Appendix A, Link Relations.

Full Text Query in Collection Resources

When you send a GET request to one of the following collection resources, you can use the full text query parameter `q` to filter entries in a result set according to specific criterion.

- Folder Child Documents
- Folder Child Objects

- Checked Out Objects
- All Versions

For more information about these resources, see the *EMC Documentum Platform REST Services Resource Reference Guide*.

The value of the full text query parameter q is a string that follows the simple search language syntax.

Note: Parentheses. which can be used in Search, are not supported in the parameter q when being appended to the collection resources shown here.

Simple Search Language

The simple search language is used in the Search resource and the full text query parameter to specify search criterion.

Search criterion in the simple search language must follow this syntax:

```
searchExpression:   orExpression
orExpression:       andExpression (<OR> andExpression)*
andExpression:      operandExpression (<AND> operandExpression)*
operandExpression:  ( parenthesis | implicitExpression )
parenthesis :       <PARENTHESIS_START> orExpression <PARENTHESIS_END>
implicitExpression: ( term )+
term:               ( positiveTerm | <NOT> positiveTerm)
positiveTerm:       ( <WORD> | <PHRASE> )
<DEFAULT> SKIP :
{
    <SPACE: ( [" ","\t","\n","\r"] )+ >
}
<DEFAULT> TOKEN :
{
    <AND: "and" >
|   <OR:  "or" >
|   <PARENTHESIS_START: "(" >
|   <PARENTHESIS_END: ")" >
|   <NOT: "not" >
|   <WORD: ( ~["(",")","\""," ","\t","\n","\r"] )+ >
|   <PHRASE: "\"" ( ~["\""] )+ "\"" >
}
```

term can be a word, or a phrase. It can also be a list of words or phrases, or both, separated by spaces and quoted appropriately. When used in the Search resource, *term* can also be enclosed in parentheses to escape boolean operators.

The full text simple language supports:

- Words
- Phrases
- Implicit AND
- Boolean Operators
- Wildcards

Words

A word is a set of characters with the following exceptions:

- (
-)
- \"
- \t
- \n
- \r

Multiple words enclosed in double quotes are treated as a phrase, such as `"foo bar"`. When the words are not enclosed, implicit and is applied.

Phrases

A phrase, which contains more than one word separated by spaces, is enclosed in double quotes. For example:`"foo bar"` returns objects that contain the phrase `foo bar`.

The single quote character (`'`) can be included as a part of a phrase without any change. For example: `"foo ' bar"`

Note that `\"` is not supported even when it is enclosed in double quotes. For example, `"foo \" bar"` is invalid.

Implicit AND

A blank space separating two terms is interpreted as the `and` boolean operator. For example: `foo bar`

This expression, which equals to `foo and bar`, returns objects that contain both `foo` and `bar`. To search for objects that contain either `foo` or `bar`, include `or` in the expression explicitly:`foo or bar`

Boolean Operators

The following boolean operators are supported in the simple search language.

- not
- and
- or

These operators follow the standard logical semantics. They are listed in order of precedence, with the highest- precedence one at the top.

In the Search resource, you can use both parentheses and double quotes to escape boolean operators. In the full text query parameter, you can only use double-quotes to escape boolean operators.

To include any of the boolean operators as a string in search criterion, enclose the operators in double quotes. And thus, the system treats them as phrases. For example: `"and" or "or"`.

Wildcards

The following wildcard characters are supported in the simple search language.

Wildcard	Description	Example
*	Matches any number of characters. You can use the asterisk (`*`) anywhere in a character string.	`wh*` finds `what`, `white`, and `why`, but not `awhile` or `watch`.
?	Matches a single alphabet in a specific position.	`b?ll` finds `ball`, `bell`, and `bill`, but not `buell`.

Facet Search

When xPlore is configured for the Content Server repositories, Documentum REST Services supports full-text search by facets, which enables you to explore a collection of search results categorized in specific dimensions (facets). By default, the following properties of a SysObject are facet-enabled:

- r_modifier
- keywords
- r_modify_date
- r_full_content_size
- a_application_type
- r_object_type
- owner_name

To enable facet on other properties, you must modify xPlore configuration and re-index.

When you perform a facet search, a facet section appears in the response feed of the search result as a sibling of entries.

```
<feed>
    ...
    <entry>
        ...
    </entry>
    ...
    ...
    <entry>
        ...
    </entry>
    <dm:facet>
        <dm:facet-id>facet_r_modify_date</dm:facet-id>
        <dm:facet-label>Modify Date</dm:facet-label>
            <dm:facet-value>
            <dm:facet-value-id>LAST_YEAR</dm:facet-value-id>
            <dm:facet-value-count>23</dm:facet-value-count>
            <dm:facet-id>facet_r_modify_date</dm:facet-id>
            <dm:facet-value-constraint>
            2013-01-01T00:00:00.000\+0000/2014-01-01T00:00:00.000\+0000
            </dm:facet-value-constraint>
            <link rel="search"
            href="http://localhost:8080/dctm-rest/repositories/acme01/search.xml?
            q=emc&facet=r_modify_date&facet-value-constraints
            =2013-01-01T00:00:00.000%5C%2B0000/2014-01-01T00:00:00.000%5C%2B0000"/>
        </dm:facet-value>
        <dm:facet-value>
            <dm:facet-value-id>LAST_MONTH</dm:facet-value-id>
            <dm:facet-value-count>3</dm:facet-value-count>
            <dm:facet-id>facet_r_modify_date</dm:facet-id>
            <dm:facet-value-constraint>
            2014-04-01T00:00:00.000\+0000/2014-05-01T00:00:00.000\+0000
            </dm:facet-value-constraint>
            <link rel="search"
            href="http://localhost:8080/dctm-rest/repositories/acme01/search.xml?
            q=emc&facet=r_modify_date&facet-value-constraints
            =2014-04-01T00:00:00.000%5C%2B0000/2014-05-01T00:00:00.000%5C%2B0000"/>
        </dm:facet-value>
    </dm:facet>
</feed>
```

The following table describes the properties in the `facet` section in detail:

Property	**Description**
facet-id	Indicates the property to be used as the facet. When being used in `facet-id`, a property is prefixed with `facet_`. In the example, the `r_modify_date` property is used as the facet.
facet-label	Label of the property used as the facet. Labels of properties are defined in DFC interfaces. For example, the label of `r_modify_date` is `Modify Date`.
facet-value	Each `facet-value` section contains data of the results belonging to one group. In the example, two `facet-value` sections appear, one for instances whose `r_modify_date` falls in last year's date range and the other for instances whose `r_modify_date` falls in last month's date range.
facet-value-id	Indicates the value of property that is used as the facet
facet-value-count	Indicates the number of results belonging to a certain group

Property	Description
facet-value-constraint	Indicates the property constraints expression.
search link	Points to corresponding results of a certain group. For example, you can navigate to instances whose `r_modify_date` falls in last year's date range by accessing this link: `http://localhost:8080/dctm-rest/repositories/acme01 /search.xml?  q=emc&facet=r_modify_date&facet -value-constraints=2013-01-01T00:00:00.000%5C %2B0000/2014-01-01T00:00:00.000%5C%2B0000`

If a repeating property is used as the facet, entries in the result set may also contain a facet block that is comprised of multiple facet value groups, each of which represents a combination of repeating values (*value_a+value_b*). This enables you to navigate to the results whose repeating property contains both *value_a* and *value_b* by accessing the corresponding search link.

Note: The plus sign (+) is encoded as `%2B` in the XML representation.

Consider the following scenario for example:

- You use the repeating property `keywords` as the facet.
- The result set contains three objects:
  - — Object1 contains the keywords `foobar` and `V1`.
  - — Object2 contains the keywords `foobar` and `V2`.
  - — Object3 contains the keywords `foobar`, `V1`, and `V2`.

In this scenario, when you click the search link `http://host:port/dctm-rest/repositories /documentum1/search?q=HellowWorld&facet=keywords&facet-value-constraints= foobar` to navigate to the entries whose `keywords` contains `foobar`, all entries in the feed contain a facet block that has multiple facet value groups. Each of these groups represents a value combination (`foobar+x`). For example, if you access the search link `http://host:port /dctm-rest/repositories/documentum1/search?q=HellowWorld&facet =keywords&facet-value-constraints= foobar%2BV1` in the first entry, Object1 and Object3 are returned as their keywords contain both `foobar` and `V1`. Similarly, the link `http://host:port /dctm-rest/repositories/documentum1/search?q=HellowWorld&facet =keywords&facet-value-constraints= V1%2BV2` returns two results, Object1 and Object 3.

In addition to the `AND` operator (+), you can also use the `OR` operator (`|`), which is encoded as `%7C` in URLs, in `facet-value-constraints`. Note that these two operators share the same precedence in `facet-value-constraints`, and the expression is evaluated according to the right to left associativity. For example, `a+b|c+d` is treated as `a+(b|(c+d))`.

Generating Link Relation in DQL Results

To facilitate content consumption, Documentum REST Services generates link relations, including thumbnail links, in DQL results. Whether to generate links and what links to generate is based on the criterion you specify in a DQL query request, such as the properties to select and types of resources to select from.

Generated links are presented at the entry level. By accessing these links, a client application is able to navigate to the corresponding resources. Typically, if the DQL query result item has a dedicated resource, the result entry contains an `edit`link relation pointing to that resource.

Example 4-25. XML Representation of Query Results with Links Generated

```
<entry>
  <id>...</id>
  <title>090007c2800001dc</title>
  <updated>...</updated>
  <content>...</content>
  <links>
    <!-- all object related links for this query result is added here -->
  <links>
</entry>
```

Example 4-26. JSON Representation of Query Results with Links Generated

```
{
  "id": "...",
  "title": "...",
  "updated": "...",
  "content": {...},
  "links": [## all object related links for this query result is added here ##]
}
```

Additional Information about Generating Links

When a client application submits a request that contains a DQL statement, the REST Server does not make any change to the DQL. The Query resource performs a simple parse to obtain the types needed for link generation from the DQL.

DQL Query Syntax:

```
SELECT [FOR base_permit_level][ALL|DISTINCT] value [AS name] {,value [AS name]}
FROM [PUBLIC] source_list
[WITHIN PARTITION (partition_id{,partition_id})
| IN DOCUMENT clause
| IN ASSEMBLY clause]
[SEARCH [FIRST|LAST]fulltext_search_condition
[IN FTINDEX index_name{,index_name}]
[WHERE qualification]
[GROUP BY value_list]
[HAVING qualification]
[UNION dql_subselect]
[ORDER BY value_list]
[ENABLE (hint_list)]
```

How and what links are generated is delegated to each resources (mostly the types you specify in the FROM clause, such as a document resource).

Note that not all DQL statements are eligible to generate links in query results as a DQL query may not return information needed to generate links. For example, properties such as `r_object_id`, or `user_name`, which can be used to identify resources, are not selected in the DQL statement. The following query does not generate any link because neither `object_name` nor `r_modify_date` is able to identify a resource.

```
select object_name, r_modify_date from dm_document
```

Additionally, if you query a type that pertains to no existing resource in Documentum REST Services, no link is generated. Similarly, if you query multiple types that are not relevant with one another, no link is generated. The following query does not generate any link:

```
select * from dm_document, dm_user
```

Thumbnail support

If the query result contains the attribute `thumbnail_url`, the system uses the value of this attribute to generate a link to the thumbnail.

Example 4-27. Query Results with Thumbnail Links Generated (XML Representation)

Sample Request:

```
GET http://localhost:8080/dctm-rest/repositories/REPO.xml?
dql=select r_object_id,object_name,r_object_type,thumbnail_url from dm_sysobject
```

Sample Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
    xmlns:dm="http://identifiers.emc.com/vocab/documentum"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <id>http://localhost:8080/dctm-rest/repositories/REPO</id>
    <title>query results</title>
    <updated>2014-05-20T15:09:18.473+08:00</updated>
    <author>
        <name>EMC Documentum</name>
    </author>
    <dm:page>1</dm:page>
    <dm:items-per-page>100</dm:items-per-page>
    <link
        href="http://localhost:8080/dctm-rest/repositories/REPO.xml?
        select+r_object_id,object_name,r_object_type,
        thumbnail_url+from+dm_sysobject" rel="self"/>
    <link
        href="http://localhost:8080/dctm-rest/repositories/REPO.xml?
        select+r_object_id,object_name,r_object_type,
        thumbnail_url+from+dm_sysobject&items-per-page=100&page=2" rel="next"/>
    <link
        href="http://localhost:8080/dctm-rest/repositories/REPO.xml?
        select+r_object_id,object_name,r_object_type,
        thumbnail_url+from+dm_sysobject&items-per-page=100&page=1" rel="first"/>
    <entry>
        <id>http://localhost:8080/dctm-rest/repositories/REPO.xml?
        select+r_object_id,object_name,r_object_type,
        thumbnail_url+from+dm_sysobject&index=0</id>
        <title>080020808000013c</title>
        <updated>2014-05-20T15:09:18.763+08:00</updated>
        <content>
            <dm:query-result
                definition=
                "http://localhost:8080/dctm-rest/repositories/REPO/types/
                dm_cryptographic_key"
                xsi:type="dm:dm_cryptographic_key">
                <dm:properties xsi:type="dm:dm_cryptographic_key-properties">
                    <dm:r_object_id>080020808000013c</dm:r_object_id>
                    <dm:object_name/>
                    <dm:r_object_type>dm_cryptographic_key</dm:r_object_type>
                    <dm:thumbnail_url>http://CS71P01:8081/thumbsrv/getThumbnail?
                    object_type=dm_cryptographic_key&format=&is_vdm=false&repository=8320
                    </dm:thumbnail_url>
                </dm:properties>
            </dm:query-result>
        </content>
        <link
            href="http://localhost:8080/dctm-rest/repositories/REPO/objects/
            080020808000013c.xml" rel="edit"/>
        <link
            href="http://CS71P01:8081/thumbsrv/getThumbnail?
            object_type=dm_cryptographic_key&format=&is_vdm=false&repository=8320"
            rel="icon"/>
    </entry>
</feed>
```

Example 4-28. Query Results with Thumbnail Links Generated (JSON Representation)

Sample Request:

```
http://10.32.168.193:8080/dctm-rest/repositories/documentum1?dql=
select r_object_id,object_name,r_object_type,thumbnail_url from dm_sysobject
```

Sample Response:

```
{
id: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1"
title: "DQL query results"
updated: "2014-07-04T15:44:00.896+08:00"
author:

 {
  name: "EMC Documentum"
 }

page: 1
items-per-page: 2
links:

 {
 rel: "self"
 href: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1?dql=
        select r_object_id,object_name,r_object_type,
         thumbnail_url from dm_sysobject"
 }

 {
 rel: "next"
 href: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1?dql=
        select r_object_id,object_name,r_object_type,
         thumbnail_url from dm_sysobject"
 }

 {
 rel: "first"
 href: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1?dql=
        select r_object_id,object_name,r_object_type,
         thumbnail_url from dm_sysobject"
 }

entries:

 {
 id: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1?dql=
      select r_object_id,object_name,r_object_type,
      thumbnail_url from dm_sysobject"
 title: "080f42418000013c"
 updated: "2014-07-04T15:44:00.896+08:00"
 content:
  {
  name: "query-result"
  type: "dm_cryptographic_key"
  definition: "http://10.32.168.193:8080/dctm-rest/repositories/
              documentum1/types/dm_cryptographic_key"
  properties:
   {
   r_object_id: "080f42418000013c"
   object_name: ""
   r_object_type: "dm_cryptographic_key"
   thumbnail_url: "http://lj-rest-cs:8081/thumbsrv/getThumbnail?object_type=
               dm_cryptographic_key&format=&is_vdm=false&repository=1000001"
   }

  links:

   }

 links:

  {
  rel: "edit"
  href: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1/objects/
      080f42418000013c"
 }

  {
  rel: "icon"
  href: "http://lj-rest-cs:8081/thumbsrv/getThumbnail?object_type=
          dm_cryptographic_key&format=&is_vdm=false&repository=1000001"
 }

 }

 {
 id: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1?
    dql=select r_object_id,object_name,r_object_type,
    thumbnail_url from dm_sysobject"
 title: "080f42418000013d"
 updated: "2014-07-04T15:44:00.896+08:00"
 content:
  {
  name: "query-result"
  type: "dm_public_key_certificate"
  definition: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1/
             types/dm_public_key_certificate"
  properties:
   {
   r_object_id: "080f42418000013d"
   object_name: ""
   r_object_type: "dm_public_key_certificate"
   thumbnail_url: "http://lj-rest-cs:8081/thumbsrv/getThumbnail?object_type=
               dm_public_key_certificate&format=&is_vdm=false&repository=1000001"
   }

  links:

  }

 links:
  {
 rel: "edit"
 href: "http://10.32.168.193:8080/dctm-rest/repositories/documentum1/objects/080f42418000013d"
 }

  {
 rel: "icon"
 href: "http://lj-rest-cs:8081/thumbsrv/getThumbnail?object_type=dm_public_key_certificate&
      format=&is_vdm=false&repository=1000001"
 }

 }

}
```

Chapter 5

Authentication

Documentum REST Services supports the following authentication schemes:

- HTTP basic authentication for inline users and LDAP users
- SPNEGO-based Kerberos authentication for users in Active Directory domains
- CAS Single Sign-on authentication for LDAP users
- RSA Access Manager Single Sign-on authentication for LDAP users
- CA SiteMinder Security Single Sign-on authentication for LDAP users

HTTP Basic Authentication

HTTP Basic authentication, which passes both usernames and passwords to the REST server, is typically implemented in the following scenarios:

Note: We strongly recommend that you use HTTPS when using HTTP Basic authentication, which requires a secure connection because the Basic authentication transmits the user name and password in BASE64 encoded plain text. HTTP Basic authentication is typically used in the following scenarios.

Scenario A (inline users):

- In your production environment, you do not have an identity provider (IdP).
- In your production environment, it is acceptable to store user credentials in Documentum repositories.

Scenario B (LDAP users):

- In your production environment, you have an IdP.
- In your production environment, it is acceptable to pass user credentials via Documentum systems (Content Server).
- In your production environment, it is acceptable to synchronize user information and group information to Documentum repositories.

When using HTTP Basic authentication, a request carries the username/password pair in the Authorization header with the following pattern:

```
Authorization: Basic BASE64{${username}:${password}}
```

Example 5-1. Username/password Pair for dmadmin/password

```
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
```

If the authentication fails, the REST server responds with the HTTP 401 Unauthorized status code.

Starting from this release, you can improve the efficiency of HTTP Basic authentication by using client tokens. That way, an authenticated REST client can access the REST server without a need to negotiate a new session ticket in a specified period of time.

The following diagram illustrates the workflow of HTTP Basic authentication with client tokens:

1. A client sends the request with HTTP Basic authentication credentials.

   ```
   GET /dctm-rest/repositories/REPO.xml HTTP/1.1
   Host: localhost:8080
   Connection: Keep-Alive
   User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
   Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
   ```

2. The Rest server tries to retrieve a session from Content Server with the credential the client provides. If the credential is valid, the Rest server creates a client token in the `DOCUMENTUM-CLIENT-TOKEN` cookie of the response and sends it back to the client.

   ```
   HTTP/1.1 200 OK
   Server: Apache-Coyote/1.1
   Set-Cookie: DOCUMENTUM-CLIENT-TOKEN= encrypted_client_token
   ```

   Otherwise, the Rest server rejects the request with an `HTTP 401` error.

3. When sending subsequent requests, the client includes the client token in the cookie

   ```
   GET /dctm-rest/repositories/REPO.xml HTTP/1.1
   Host: localhost:8080
   Connection: Keep-Alive
   User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
   Cookie: DOCUMENTUM-CLIENT-TOKEN= encrypted_client_token
   ```

4. The Rest server validates the client token. If the client token is valid, the server returns the response. If the client token expires or the token is invalid, the Rest server rejects the request with `HTTP 401`.

5. The client sends a request to log out:

   ```
   GET /dctm-rest/logout HTTP/1.1
   Host: localhost:8080
   Connection: Keep-Alive
   User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
   Cookie: DOCUMENTUM-CLIENT-TOKEN= encrypted_client_token
   ```

   See Explicit Logoff for more information.

6. The REST server reset the client token and returns `HTTP 204`:

   ```
   HTTP/1.1 204 No Content
   Server: Apache-Coyote/1.1
   Set-Cookie: DOCUMENTUM-CLIENT-TOKEN=""; Expires=Thu,
   01-Jan-1970 00:00:10 GMT; Path=/dctm-rest; HttpOnly
   Date: Thu, 06 Nov 2014 06:35:33 GMT
   ```

Note: When an authenticated client provides both HTTP Basic authentication credentials and a client token, the REST server ignores the client token, validating the credentials only. When the credentials are valid, a new client token is generated.

Enabling HTTP Basic Authentication

HTTP Basic is the default authentication scheme in Documentum REST Services, and thus it is enabled out of the box. If you want to enable client tokens in HTTP Basic Authentication, set the `rest.security.auth.mode` property to `ct-basic` in *`dctm-rest`*`\WEB-INF\classes\rest-api-runtime.properties`.

Note: *dctm-rest* represents the directory where you extract the `dctm-rest.war` file.

User Credential Mapping

In scenario A, user credentials are mapped as follows:

```
${username} = ${dm_user.user_login_name}
${password} = ${dm_user.user_password}
```

In scenario B, we recommend that you use the following mapping rule consistently for both Content Server domain-required and non-domain-required modes. This mapping is generic for any type of LDAP servers supported by Content Server.

```
${username} = ${dm_user.user_login_domain}\${dm_user.user_login_name}
${password} = ${LDAP user password}
```

If your Content Server repository is configured with the domain-required authentication model, it is possible to have multiple users with the same login name if each user is in a different domain. Therefore, under domain-required authentication model, it is required to put the domain name prefix in the `username` variable. If domain-required authentication model is not enabled (default setting), the domain name prefix is optional in the `username` variable.

Note:

- `dm_user.user_login_domain` is mapped from the Content Server LDAP Configuration name when LDAP users are synchronized to Content Server.
- In Content Server, you can create a user with a login name that contains the backslash sign ('\'), for example user `alpha\beta`, where alpha is not a domain name. To make such an authentication succeed, the REST client must insert an empty domain name to the user login name, for example, `\alpha\beta`. This is a known limitation in Content Server.

Known Limitation

According to RFC2617, section-2, user names for HTTP basic authentication cannot contain the colon character (':').

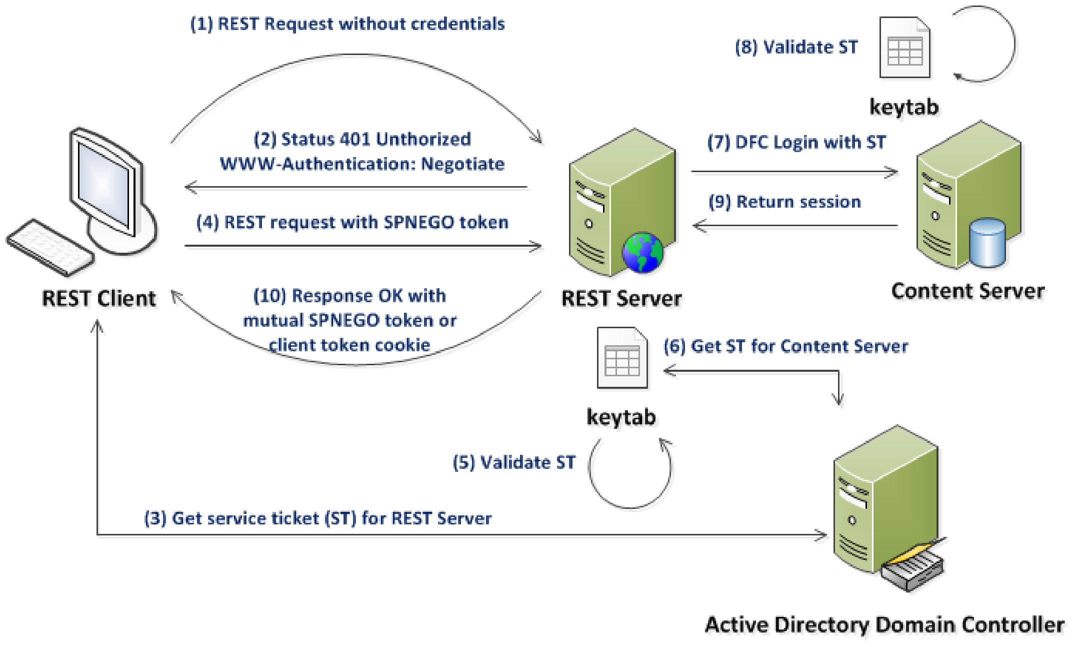
Authentication with SPNEGO-based Kerberos

Documentum REST Services implements SPNEGO-based Kerberos authentication to authenticate a request when Documentum REST Services is registered as a Kerberos service. We suggest that you use Kerberos authentication if your production environment meets the following conditions:

- You have set up your Active Directory domain environment.
- You want to integrate Kerberos SSO protocol with Content Server and other related products in the Documentum system.

Authentication Workflow

The following diagram illustrates the workflow of SPNEGO-based Kerberos Authentication in Documentum REST Services:


(1) REST Request without credentials
(8) Validate ST
keytab
(2) Status 401 Unthorized
WWW-Authentication: Negotiate
(7) DFC Login with ST
(9) Return session
(4) REST request with SPNEGO token
REST Client
(10) Response OK with
mutual SPNEGO token or
client token cookie
REST Server
Content Server
(6) Get ST for Content Server
keytab
(5) Validate ST
(3) Get service ticket (ST) for REST Server
Active Directory Domain Controller

Workflow of RESTful SPNEGO-base Kerberos Authentication

Note:

- The workflow may vary because the negotiation protocol allows mutual authentication that may take several rounds of request/response to complete the handshake.
- The client token cookie is optional. By default, it is disabled. For more information about how a client token works, see Client Token, page 116. When client tokens are used, SPNEGO-based Kerberos authentication supports single sign-out that invalidates client token cookies. When a client explicitly logs out, the session is terminated and the client has to negotiate a new session ticket.

Authentication negotiation between the REST client and the REST server

1. The REST client sends a resource request with no credentials to the REST server (Step 1 in the diagram).

   ```
   GET /${resource-url} HTTP/1.1
   ```

2. The REST server responds with the 401 status error and a Negotiate header (Step 2 in the diagram).

   ```
   HTTP/1.1 401 Unauthorized
   WWW-Authenticate: Negotiate
   ```

3. The REST client negotiates a SPNEGO-based Kerberos token and resends the resource request (Step 3 to Step 9 in the diagram).

   ```
   GET /${resource-url} HTTP/1.1
   Authorization: Negotiate YIIZG1hZG1pbjpwYXNzd29yZ.....
   ```

4. The REST server returns the requested resource. Optionally, a mutual token for the client to verify can be sent back to the client (Step 10 in the diagram).

   ```
   HTTP/1.1 200 OK
   WWW-Authenticate: Negotiate CXXCBApbjpwYXBSS90B.....
   ```

   Note: Note: Because Kerberos service ticket is for one-time use, a REST client cannot resubmit the same SPNEGO token for multiple times of authentication. If the client token cookie is not enabled, the REST client must negotiate new Kerberos service tickets for subsequent REST requests.

Multi Domain Support within a Forest

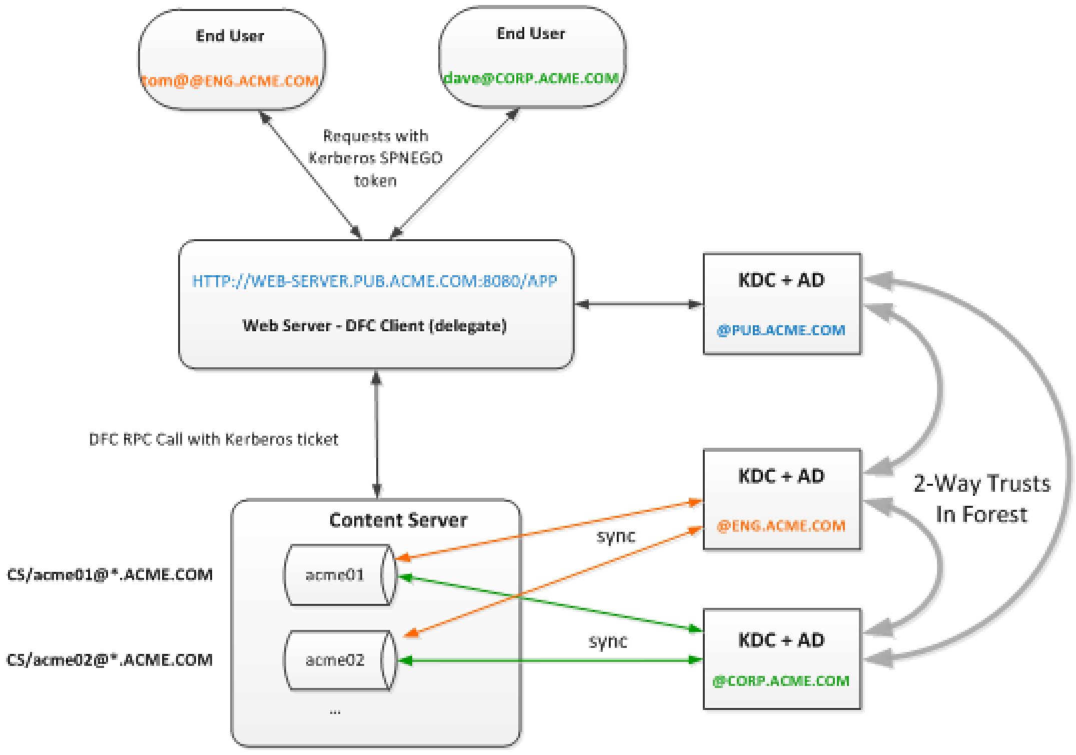
In Documentum REST Services, Kerberos authentication can be implemented across multiple domains if the following conditions are true:

- These domains are in the same forest.
- Two-way trusts are enabled across these domains.

A typical scenario where Kerberos authentication across multiple domains is used is shown as follows:

A domain user (A) tries to access a REST resource located on a domain other than the one where user A logs on.

Content Server supports Kerberos Multi Domain authentication within a forest since version 6.7SP2. The following diagram illustrates how multi-domain Kerberos authentication works across Documentum stacks.



Sample of Kerberos Mutli Domain Integration

In this diagram, the REST server is deployed in a different domain than the domain where users log in:

- The Documentum REST Services deployment is registered to domain `PUB.ACME.COM` with `SPN HTTP/WEB-SERVER.PUB.ACME.COM`.
- The logon users come from other domains `CORP.ACME.COM` and `ENG.ACME.COM`.
- Content Server repositories are registered to the sub-trusted domains under `ACME.COM`. This can be `PUB.ACME.COM`, `CORP.ACME.COM`, `ENG.ACME.COM`, or any other sub-trusted domain.
- Repositories must synchronize domain users from the Active Directory. In this diagram, each repository synchronizes AD users from multiple domains.
- Each repository can support multi-domain Kerberos authentication.

Setup SPNEGO-based Kerberos

To implement SPNEGO-based Kerberos authentication for Documentum REST Services, the following configurations are required:

- Content Server Configuration for SPNEGO-based Kerberos, page 73
- REST Server Configuration for SPNEGO-based Kerberos, page 73
- Recommendations on Web Browser Configuration for SPNEGO-based Kerberos, page 80

Note: Before configuring Content Server and the REST server for SPNEGO-based Kerberos, you must set up the domain controllers successfully in your Active Directory environment.

Content Server Configuration for SPNEGO-based Kerberos

To implement SPNEGO-based Kerberos authentication, the following configurations are required for Content Server:

- Register and map the service principal name
- Create the Kerberos configuration file
- Configure LDAP synchronization

For detailed information about Kerberos-related configurations in Content Server, see the *EMC Documentum Content Server Administration and Configuration Guide*.

REST Server Configuration for SPNEGO-based Kerberos

To implement SPNEGO-based Kerberos authentication, the following configurations are required for the REST server:

- Register and Map the Service Principal Name, page 73
- JAAS configuration, page 75
- Configuring Runtime Properties, page 78
- Start up script

You must restart the application server after you finish the configurations above.

Register and Map the Service Principal Name

First, you must determine a Service Principal Name (SPN) for the REST server. The SPN follows one of the following patterns:

- Pattern 1:

  ```
  <service-class>/<host>:<port>/<service-name>
  ```

- Pattern 2:

```
HTTP/<FULL-HOST-NAME>
```

Example: `HTTP/RESTSERVER.ACME.COM`

We suggest that you use pattern 2 because it is compatible with most web clients.

After you determine your SPN, follow these steps on the domain controller that is registered as the Key Distribution Center (KDC):

1. In the Active Domain Computers and Users Management console, create a service account for Content Server, for example, `webadmin`.
2. Run the following command to create the SPN:

   ```
   setspn -a <REST-SPN> <REST-SERVICE-ACCOUNT>
   ```

   Example:

   ```
   setspn -a HTTP/RESTSERVER.ACME.COM webadmin
   ```

3. Create a keytab file for the SPN and then map the SPN to the service account. To do this, run the following command:

   ```
   ktpass /pass <PASSWORD> -out <OUTPUT-KEYTAB-FILE-LOCATION>
   -princ <REST-SPN>@<REALM-NAME> -crypto <ENCRYPT-TYPE>
   +DumpSalt -ptype KRB5_NT_PRINCIPAL /mapOp set
   /mapUser <REST-SERVICE-ACCOUNT>
   ```

   For more information about the ktpass utility, see visit the following web site:

   http://technet.microsoft.com/en-us/library/cc776746(v=ws.10).aspx

   Example:

   ```
   ktpass /pass Password123 -out C:\shared\acme01.acme.keytab
   -princ HTTP/RESTSERVER.ACME.COM@ACME.COM -crypto ALL +DumpSalt
   -ptype KRB5_NT_PRINCIPAL /mapOp set /mapUser webadmin
   ```

4. Copy the keytab file to the application server where Documentum REST Services is deployed.
5. In the Active Domain Computers and Users Management console, select "Trust this user for delegation to any service (Kerberos only)" for the service account (for example, webadmin) in the Delegation tab.

JAAS configuration

The JAAS configuration file entry contains JAAS-specific settings such as the `<LoginContext>` name (which is also the name of the configuration entry), settings for the Kerberos login module, the REST server's SPN, and the location of the `*.keytab` file.

The location and format of the JAAS configuration settings might be different for each application server. The JAAS configuration file in most application servers is named `jaas.conf`.

`WebSphere`: In the `WebSphere` Application Server, the JAAS configuration file is named `wsjaas.conf`, and it's located in the following directory:

`<WAS_Installation_path>\AppServer\profiles\<APP_SERVER_NODE_NAME>\properties`.

In `JBoss 6.3` the `<WAS_Installation_path>\AppServer\profiles\<APP_SERVER_NODE_NAME>` directory is not available. In `JBoss 6.3` the deployment must go in the `jboss-eap-6.3\standalone\deployments` directory.

`JBoss 6.3`: In the `Jboss 6.3` Application Server, the JAAS configuration must be specified in the `standalone.xml` file, which is located in the following directory: `jboss-eap-6.3\standalone\configuration\`.

In your `web.xml` file, you must set the `jaas.config` entry so that it points to the above `standalone.xml` file.

`Jboss 6.3` uses JAAS configurations that are included in its `standlaone.xml` file. Any configuration settings that are made to any other files are ignored by the `Jboss 6.3` Application Server.

Element Id	Description
`<security-domain>`	Corresponds to the Documentum DFS web application's SPN. You replace separator characters with hyphen characters and omit the `@REALM` segment in the SPN. For example, the following Security domain is derived from the corresponding SPN: • security-domain: `HTTP-myhost-mydomain-com-8080` • SPN: `HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM` **Note:** Make sure that the SPN in the JAAS configuration matches the SPN defined in `dfs-runtime.properties`.

Element Id	Description

`<LoginModule>`	Specify the Kerberos login module to be used to perform user authentication.
	
	For Single Domain:
	
	• **Sun JDK**: `com.sun.security.auth .module .Krb5LoginModule`
	
	• **IBM JDK**: `com.ibm.security.auth .module .Krb5LoginModule`
	
	• **Quest Libraries**: `com.dstc.security .kerberos.jaas .KerberosLoginModule`
	
	For Multi-Domain:
	
	• `com.dstc.security.kerberos .jaas .KerberosLoginModule`
	
	Note: For Quest login modules, when you want to enable ticket cache, perform one of the following operations:
	
	• **Enable createTicketCache**:
	
	`useTicketCache=true`
	
	`createTicketCache=true`
	
	• **Enable createTicketCache and specify a cache path**:
	
	`useTicketCache=true`
	
	`createTicketCache=true`
	
	`ticketCache=<cache_path> 32`
	
	Otherwise, disable ticket cache by setting `useTicketCache` to `false`.

`<SPN>`	The Documentum DFS web application's SPN. For example, for SUN and IBM login modules: `HTTP/myhost .mydomain.com:8080@MYDOMAIN.MYCORP.COM` For QUEST login modules, the SPN does not contain the `@` character and the string after that. For example: `HTTP/myhost.mydomain.com:8080`
`<REALM>`	(Multi-domain support only) The realm name. For example: `@MYDOMAIN.MYCORP.COM`
`<dfsuser_keytab_path>>`	The path to the user account's `*.keytab` file on the Documentum DFS web application. For example: `C:\dfsuser.keytab`

Example 5-2. The `standalone.xml` File Entry

```
<security-domain name="HTTP-myhost-mydomain-com-8080" cache-type="default">
   <authentication>
      <login-module code="com.dstc.security.kerberos.jaas.KerberosLoginModule"
         flag="required">
          <module-option name="storeKey" value="true"/>
          <module-option name="useKeyTab" value="true"/>
          <module-option name="principal" value=" HTTP/myhost.mydomain.com:8080"/>
          <module-option name="keyTab" value="C:/kerberos/dfs.keytab"/>
          <module-option name="doNotPrompt" value="true"/>
          <module-option name="debug" value="true"/>
          <module-option name="useTicketCache" value="false"/>
          <!--<module-option name="refreshKrb5Config" value="true"/> -->
          <module-option name="noTGT" value="true"/>
          <module-option name="realm" value=" MYDOMAIN.MYCORP.COM "/>
      </login-module>
   </authentication>
</security-domain>
```

Example 5-3. JAAS Configuration referring to QUEST Libraries which Support both Single Domain and Multi Domain

```
{
 com.dstc.security.kerberos.jaas.KerberosLoginModule required
  debug=false
 principal=<SPN>
 realm="RESTKDC.IIG.EMC.COM"
 refreshKrb5Config=true
 noTGT=true
 useKeyTab=true
 storeKey=true
 doNotPrompt=true
 useTicketCache=false
 isInitiator=false
 keyTab=<REST_user_keytab_path>;
};
```

<loginContext>	Corresponds to the Documentum REST Services web application's SPN. You replace separator characters with hyphen characters and omit the `@REALM` segment in the SPN. For example, the following `LoginContext` is derived from the corresponding SPN:
	• `LoginContext`:
	`HTTP-myhost-mydomain-com-8080`
	• SPN:
	`HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM`
	Note: Make sure that the SPN in the JAAS configuration matches the SPN defined in `rest-api-runtime.properties`.

<LoginModule>	Specify the Kerberos login module to be used to perform user authentication. **Referring to QUEST Libraries for both Single domain and Multi-Domain**: `com.dstc.security.kerberos.jaas.KerberosLoginModule`

	Note: If you want to enable ticket cache, perform one of the following operations. Otherwise, disable ticket cache by setting `useTicketCache` to `false`.
	• **Enable createTicketCache**:
	`useTicketCache=truecreateTicketCache=true`
	• **Enable createTicketCache and specify a cache path**:
	`useTicketCache=true`
	`createTicketCache=trueticketCache=<cache_path>`

<SPN>	For QUEST login modules, the SPN does not contain the `@` character and the string after that. For example:
	`HTTP/myhost.mydomain.com:8080`
	If Documentum REST Services is deployed on WebLogic, append the realm name to the SPN. For example:
	`HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM`

<REALM>	(Multi-domain support only) The realm name. For example: `@MYDOMAIN.MYCORP.COM`
<REST_user_keytab _path>	The path to the user account's `*.keytab` file on the REST server. For example: `c:\RESTuser.keytab`

Configuring Runtime Properties

The `rest-api-runtime.properties` file contains the runtime properties for Documentum REST Services. You can configure the following authentication-related settings by using the corresponding properties in the file:

- Authentication mode
- Security configuration for client tokens, and expiration policies
- File path where `jaas.conf` is located (If you implement SPNEGO-based Kerberos authentication on Oracle Weblogic Server, do not specify the location of the jaas.conf file in the `rest-api-runtime.properties` file. For more information, see JAAS Configuration in Weblogic, page 79)
- Hostname or IP address of the DNS server for the domain
- Threshold (in bytes) at which Kerberos authentication cuts over from UDP to TCP

This file contains detailed instructions on how to set the runtime properties. Follow the instructions to complete the settings. The file is located in the following directory of the application server where REST web services is deployed:

```
<dctm-rest>\WEB-INF\classes
```

`<dctm-rest>` represents the directory where you extract the `dctm-rest.war` file.

JAAS Configuration in Weblogic

If Documentum REST Services is deployed on WebLogic, Kerberos authentication requires the following configurations:

- Besides setting the location of the `jaas.config` file in the `rest-api-runtime.properties` file, which is located in `dctm-rest.war\WEB-INF\classes\`, set its location by using a JVM argument of the `setDomainEnv` script:

  ```
  set JAVA_OPTIONS=%JAVA_OPTIONS% -Djava.security.auth.login.config=<jaas config file path>
  ```

- In the `rest-api-runtime.properties` file, append the realm name to the `rest.security.kerberos.spn` property:

  ```
  rest.security.kerberos.spn=HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM
  ```

Startup Script Modification

For Linux operating systems, if client tokens are used in your deployment, add the following option to the startup script of the application server where Documentum REST Services is deployed to achieve better performance:

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.egd=file:/dev/./urandom"
```

Recommendations on Web Browser Configuration for SPNEGO-based Kerberos

You may want to build a web browser application with access to Documentum REST Services using Kerberos authentication. Most web browsers support the HTTP Negotiate protocol. To enable HTTP Negotiate authentication for SPNEGO-based Kerberos SSO, you may need to change your browser configurations. The following instructions help you configure HTTP Negotiate authentication for SPNEGO-based Kerberos on some commonly used web browsers. To get the detailed support on configuring web browsers, refer to the web browser documentation.

Note: The instructions in this section are based on the following assumptions:

- You have created an Active Directory domain named `ACME.COM`.
- You have deployed Documentum REST Services on a Tomcat web server that is running on a computer with the host name `RESTSERVER.ACME.COM` in the domain.
- The home page for Documentum REST Services is: `http://restserver.acme.com:8080 /dctm-rest/`.
- You have created a Kerberos SPN `HTTP/RESTSERVER.ACME.COM` for Documentum REST Services.
- You log in to another domain computer with the host name `RESTCONSUMER.ACME.COM` with a domain user account `ACME\tuser / password`.
- You have finished Kerberos configurations for both the REST server and Content Server correctly.

Microsoft Internet Explorer

Minimal Requirements:

- Microsoft Internet Explorer 8.0 or later versions.
- The domain user `ACME\tuser` on the computer `RESTCONSUMER.ACME.COM` has been granted permissions to configure intranet zones. For example, you have already added this user to the Local Administrators group.

To enable SPNEGO-based Kerberos authentication for Documentum REST Services in Internet Explorer, follow these steps:

1. Open Internet Explorer.
2. Navigate to Tools -> Internet Options -> Security
3. Click Local intranet -> Sites -> Advanced. The Local Intranet dialog box opens.
4. In the Add this website to the zone field, enter the following site:

`http://*.acme.com`

And then, click Add -> Close -> OK.

5. On the Internet Options box, navigate to the Advanced tab.

   Scroll down to the Security settings, and then check Enable Integrated Windows Authentication.

   Click OK.

6. Restart Internet Explorer.

Mozilla Firefox

Minimal Requirements:

- Firefox 3.0 or later versions.

To enable SPNEGO-based Kerberos authentication for Documentum REST Services in Mozilla Firefox, follow these steps:

1. Open Firefox.
2. In the address bar, enter **`about:config`**.
3. In the Search bar, enter **`negotiate-auth`**. Preference names starting with `netowrk.negotiate-auth` will be returned.
4. Double click the preference `network.negotiate-auth.trusted-uris`, enter **`restserver.acme.com`**, and then click OK.
5. Double click the preference `network.negotiate-auth.delegation-uris`, enter **`restserver.acme.com`**, and then click OK.
6. Restart Firefox.

Google Chrome

Minimal Requirements:

- Chrome 10.0 or later versions

To enable SPNEGO-based Kerberos authentication for Documentum REST Services in Google Chrome, follow these steps:

1. Locate `Chrome.exe` on your system. Typically, the file is located in the following directory:

   `C:\Users\tuser\AppData\Local\Google\Chrome\Application\`

2. Create a `.bat` file, and then open it with a text editor.
3. Input the following text in the `.bat` file:

   ```
   <chrome-directory>\chrome.exe --auth-schemes="negotiate"
   --auth-server-whitelist="*acme.com"
   --auth-negotiate-delegate-whitelist="*acme.com"
   ```

Example 5-4. Bat sample

```
C:\Users\tuser\AppData\Local\Google\Chrome\Application\chrome.exe
    --auth-schemes="negotiate" --auth-server-whitelist="*acme.com"
    --auth-negotiate-delegate-whitelist="*acme.com"
```

4. Save the .bat file.

5. If the Chrome does not resolve the host `restserver.acme.com`, add a host/IP mapping to the host file.
6. Double click the .bat file you created to start Chrome.

Verify Browser Settings

To verify that SPNEGO-based Kerberos authentication for Documentum REST Services is correctly enabled in your web browser, enter the following URL in the address bar

`http://restserver.acme.com:8080/dctm-rest/repositories/acme/currentuser.xml`

If all settings are correct, the Current User resource, which contains the information of the login user `tuser`, is returned.

Troubleshooting and Diagnostics

The following checklist lists the settings that you must verify for the correct configuration of Kerberos authentication:

- Domain Controller
  — The service account that creates the SPN and maps the keytab must be configured as "Delegation Trust."

  — The encryption types for Kerberos service tickets must be supported across the client machines, the REST server machine, and the Content Server machine.

  — The system time across all domain machines must be synchronized.

  — For multi-domain deployment, the domains must be in the same forest, and two-Way trusts must be activated. You can verify this in the Active Directory Domains and Trusts console.
- Content Server
  — The Content Server repository SPN format must follow this pattern: `CS/<repository name>`.

  — The Content Server repository keytab must be loaded successfully. You can verify this by checking the Content Server repository log file under `%Documentum%/dba/logs`.

  — The DNS servers defined in the `krb5.ini` file must be reachable.
- REST Server
  — The REST Services SPN is suggested to follow this pattern: `HTTP/<HOSTNAME>`.

  — The REST Services keytab must be loaded successfully. You can verify this by running the following command:

```
%JAVA_HOME%/bin/klist -k -t <keytab-file-path>
```

— The DNS servers defined in the `security.rest.kerberos.nameservers` property must be reachable.

- REST Client
  — The client session must be within the domain. You can verify this by running the command **`klist tgt`** which checks the client TGT cache.

  — The service ticket to issue on the client must be forwardable.

Retrieve debug information

Kerberos-specific DEBUG information helps you identify the root cause of a Kerberos authentication failure.

To enable REST server Kerberos debugging, perform one or more of the following operations:

- Add a DEBUG level logger for class package `com.emc.documentum.rest` in the `log4j.properties` file located in `<dctm-rest>\WEB-INF\classes`:

```
log4j.logger.com.emc.documentum.rest=DEBUG, R
```

  The log file is saved in the location defined by the `log4j.appender.R.File` parameter.

- Enable JAAS debugging in the jaas.conf file:

```
<REST-SPN-LOGIN-MODULE>
{...
debug=true
...};
```

  The log file is shown on the console.

- Enable QUEST library debugging. To do this, add the following parameter to Web Server startup script (for example, catalina.bat in Tomcat):

```
-Didm.spnego.debug=true -Djcsi.kerberos.debug=true
```

  The log file is shown on the console.

To enable Content Server Kerberos debugging, follow these steps:

1. Stop docbase.
2. Edit the service parameter and append `-otrace_authentication`.
3. Start docbase.

The log files are saved in the following files:

- ${Documentum}\dba\logs\<docbase>.log
- ${Documentum}\dba\logs\dm_krb_<docbase>.log

Common Errors

Clients use NTLM authentication

When the Kerberos protocol is not enabled on client machines, the web browser may alternatively issue a NTLM token to respond the "HTTP 401 Negotiate" challenge. NTLM authentication is not supported by the REST server.

In this scenario, the following error message is returned:

```
E_NTLM_AUTH_NOT_SUPPORTED - The give token is a NTLM token, not Kerberos
token.  The NTLM authentication is not supported.
```

When this problem occurs, contact your administrator to validate your Kerberos deployment on the client machine.

Kerberos tickets are not forwarded

When the Kerberos TGT is not configured as forwardable, any ticket it produces (including the service ticket) is forwarded, and thus the Kerberos authentication fails.

In this scenario, the following error message is generated on the server:

```
E_NULL_DELEGATED_KERBEROS_CREDENTIAL - No credential delegated for the
Kerberos authentication with SPN: xxx and service ticket encrypted--<xxx>.
Pleas check whether the service account for the SPN has been configured as
delegation trust in KDC, or the service ticket issued on the client side
has been configured as forwardable.
```

When this problem occurs, contact your administrator to validate your Kerberos service ticket generation policy on client machines.

Delegation trust is not enabled for the service account

When the service account, the owner of the SPN, is not configured as delegation trust, the Kerberos authentication fails.

In this scenario, the following error message is generated on the server:

```
E_NULL_DELEGATED_KERBEROS_CREDENTIAL - No credential delegated for the
Kerberos authentication with SPN: xxx and service ticket encrypted--<xxx>.
Pleas check whether the service account for the SPN has been configured as
delegation trust in KDC, or the service ticket issued on the client side
has been configured as forwardable.
```

When this problem occurs, contact your administrator to validate service account setting on KDC servers.

CAS Authentication

Overview

Documentum REST Services supports Central Authentication Service (CAS) authentication to achieve a robust authentication and single sign-on (SSO) infrastructure for both browser and non-browser clients.

When using the CAS authentication scheme, you must install a CAS server (or clustered CAS servers) to provide the authentication service. Additionally, you must set up a directory service behind the CAS server and synchronize the directory users to a Content Server repository. Documentum REST Services does not provide CAS login or validation by itself. Instead, as a CAS client, REST Services forwards unauthorized Documentum REST clients to the CAS server to perform authentication and it validates the proof of principals on the CAS server by using its trust relationship with the CAS server. Upon a successful validation, the REST Services obtains a CAS proxy ticket on behalf of the clients to access the Content Server repository, where a CAS plug-in must be installed to provide the CAS proxy login capability.

Terminology

CAS-related terminologies are defined in the following table:

Table 6. Terminologies in CAS authentication

Term	Description
Client Token (CT)	Authentication token that the REST server provides for clients. This token contains an encrypted Documentum ticket and additional metadata used for token expiration. For more information about how a client token works, see Client Token, page 116.
Ticket Granting Ticket (TGT)	Ticket indicating that a client has successfully logged in to the CAS server
Service Ticket (ST)	Ticket that the CAS server sends to a service for identifying that service
Proxy Granting Ticket (PGT)	Ticket that the CAS server sends to a service with valid an ST for requesting Proxy Tickets
Proxy Ticket (PT)	Ticket that a proxy service uses to access a target service for multi-tier authentication

See Also

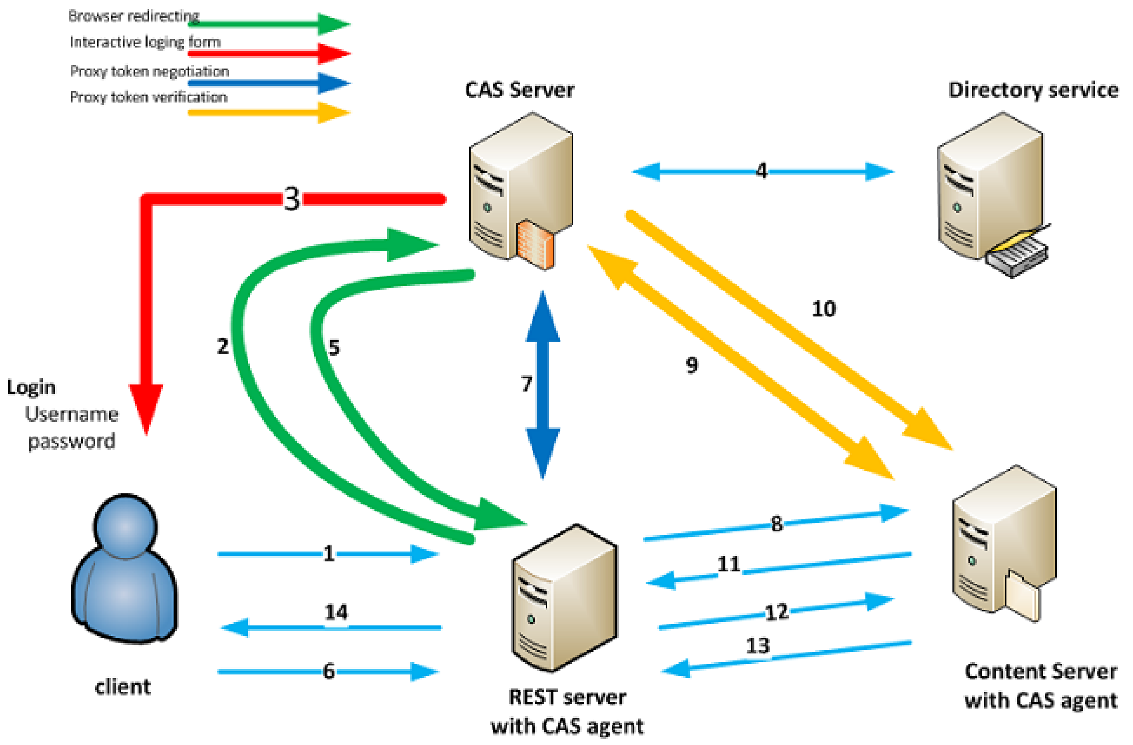
CAS is an open-source Java server component. You can find more information about CAS on the following web site:

www.jasig.org/cas

Authentication Workflow

CAS authentication supports browser and non-browser clients. This section elaborates on the authentication workflows for both scenarios.

The following diagram illustrates the workflow of CAS authentication for browser clients:



Authentication negotiation between a browser REST client and the REST server

1. A REST client (a web browser) sends a request to access a REST Services resource, for example, the client tries to visit the `acme01` repository with the following URL: `http://192.168.0.1:8080/dctm-rest/repositories/acme01`
2. The REST server sends back a `302` redirecting response, asking the client to authenticate itself on the CAS server:

```
Response Status Code: 302 Moved Temporarily
Location https://casserver:8443/cas/login?service=
http%3A%2F%2F192.168.0.1%3A8080%2Fdctm-rest%2Frepositories%2Facme01
```

   The browser client is automatically redirected to the location specified in the `Location` header and the CAS login page is displayed.
3. The browser client enters the username and password and submits the request to the CAS Server.
4. The CAS Server connects to a directory service to verify the user credential.
5. After the verification, the CAS Server returns a `302` redirecting response, providing the ST and TGT for the client.

```
Response Status Code: 302 Moved Temporarily
Location http://192.168.0.1:8080/dctm-rest/repositories/acme01
    ?ticket=ST-238-mHMdsK0A9sAhie2T1dep-cas01.example.org
```

```
Set-Cookie CASPRIVACY=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT; Path=/cas/
    CASTGC=TGT-165-zm6GUY4gFJP3AjtY4EQiXJ7M5lIGJDiIevY6AZTlkOhg2F9J2Bcas01.
example.org;Path=/cas/; Secure
```

The ST is located in the `ticket` URL parameter in the return message and it is for one-time use.

6. The client uses the ST obtained in step 5 to access the resource identified by the `Location` header of the response. The TGT is located in the Set-Cookie header. The client uses the TGT cookie to acquire subsequent STs, without a need to provide the username and password again.

7. The REST server negotiates with the CAS server to obtain the PT. For more information about this process, see Proxy Ticket Negotiation between REST and CAS, page 91.

8. The REST server sends the PT to Content Server.

9. The CAS plug-in on Content Server calls the CAS server to validate the PT.

```
https://casserver/cas/proxyValidate
?service=http://192.168.0.1:8080/dctm-rest/repositories/acme01
&ticket=ST-957-ZuucXqTZ1YcJw81T3dxf
```

10. The CAS server validates the PT. If the PT is valid, the CAS server responds Content Server with the proxy address.

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
    <cas:authenticationSuccess>
        <cas:user>testUser</cas:user>
        <cas:attribute name="dmCSLdapUserDN" value="CN=testUser,CN=Users,
        DC=ACME,DC=COM"/>
        <cas:proxies>
            <cas:proxy>https://192.168.0.1:8443/dctm-rest/cas/proxy/receptor
            </cas:proxy>
        </cas:proxies>
    </cas:authenticationSuccess>
</cas:serviceResponse>
```

11. Content Server creates a session for the client specified in the CAS response and generates a ticket that the client is able to use for subsequent calls.

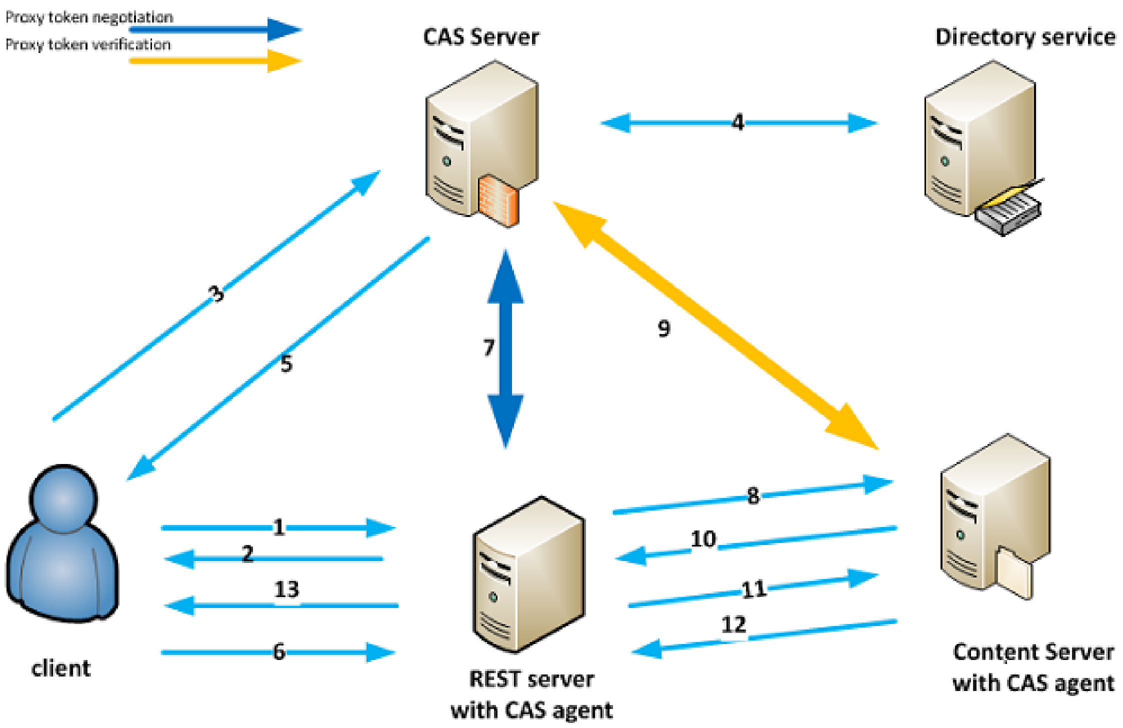
12. The REST server submits the actual call to Content Server by using the session created in step 11.

13. Content Server returns the operation results to the REST server.

14. The REST server returns the results to the client, including a DOCUMENTUM-CLIENT-TOKEN cookie (see Client Token, page 116) that the client can use for future calls.

```
Response Status Code: 200 OK
Content-Type    application/json;charset=UTF-8
Set-Cookie DOCUMENTUM-CLIENT-TOKEN="Ym9ibGVlOkRNX1RJQ0tFVD1UMEpL...==";
    Version=1; Path=/dctm-rest; HttpOnly
Response Body:
{ "id": 15,  "name": "acme01",.....}
```

The following diagram illustrates the workflow of CAS authentication for non-browser clients:



Authentication negotiation between a non-browser REST client and the REST server

1. A non-browser client sends a request to access a REST Services resource.
2. The REST server rejects the request because it does not carry any authentication proof. If the `DOCUMENTUM-NO-CAS-REDIRECT` header in the request is set to `true`, the REST server returns code `401` and puts the CAS RESTful ticket URL in the `Location` header.

```
=========== Request ===============
GET http://192.168.0.1:8080/emc-rest/repositories/acme01
DOCUMENTUM-NO-CAS-REDIRECT: TRUE
Host: 192.168.0.1:8080

=========== Response ===============
Status code: 401 Unauthorized
Location: https://casserver:8443/cas/v1/tickets
WWW-Authenticate : CAS realm="com.emc.documentum.rest"
```

   If the `DOCUMENTUM-NO-CAS-REDIRECT` header in the request is set to `false`, the REST server returns code `302`.

```
=========== Request ===============
GET http://192.168.0.1:8080/emc-rest/repositories/acme01
Host: 192.168.0.1:8080

=========== Response ===============
Status code: 302 Found
https://casserver:8443/cas/login?service=
    http%3A%2F%2F192.168.0.1%3A8080%2Femc-rest%2Frepositories%2Facme01
```

3. The client sends a POST request to the CAS server to obtain a TGT.

```
=========== Request ===============
   POST https://casserver:8443/cas/v1/tickets
```

```
    Host: casserver:8443
    Request Body: username={username}&password={password}
```

4. The CAS Server validates the client.
5. The CAS Server returns back a TGT.

```
=========== Response ===============
    Status code: 201 Created
    Response Body: https://casserver:8443/cas/v1/tickets/TGT-166-AHw7Sv5w
    FnVtWaUQZzxOTRc5YxiGnMJPEVWyai0mFeTccjwnWa-cas01.example.org
```

6. The client sends a POST request to the CAS server to obtain an ST for the resource by using the TGT.

   ```
   =========== Request ===============
   POST https://casserver:8443/cas/v1/tickets/
    TGT-166-AHw7Sv5wFnVtWaUQZzxOTRc5YxiGnMJPEVWyai0mFeTccjwnWa-cas01.example.org
   Host: casserver:8443
   Request Body:
   service=http%3A%2F%2F10.37.10.28%3A8080%2Femc-rest%2Frepositories%2Facme01

   =========== Response ===============
   Status code: 200 OK
   Response Body:
   ST-239-rYiYHoQJ2ZhJopdMuxjl-cas01.example.org
   ```

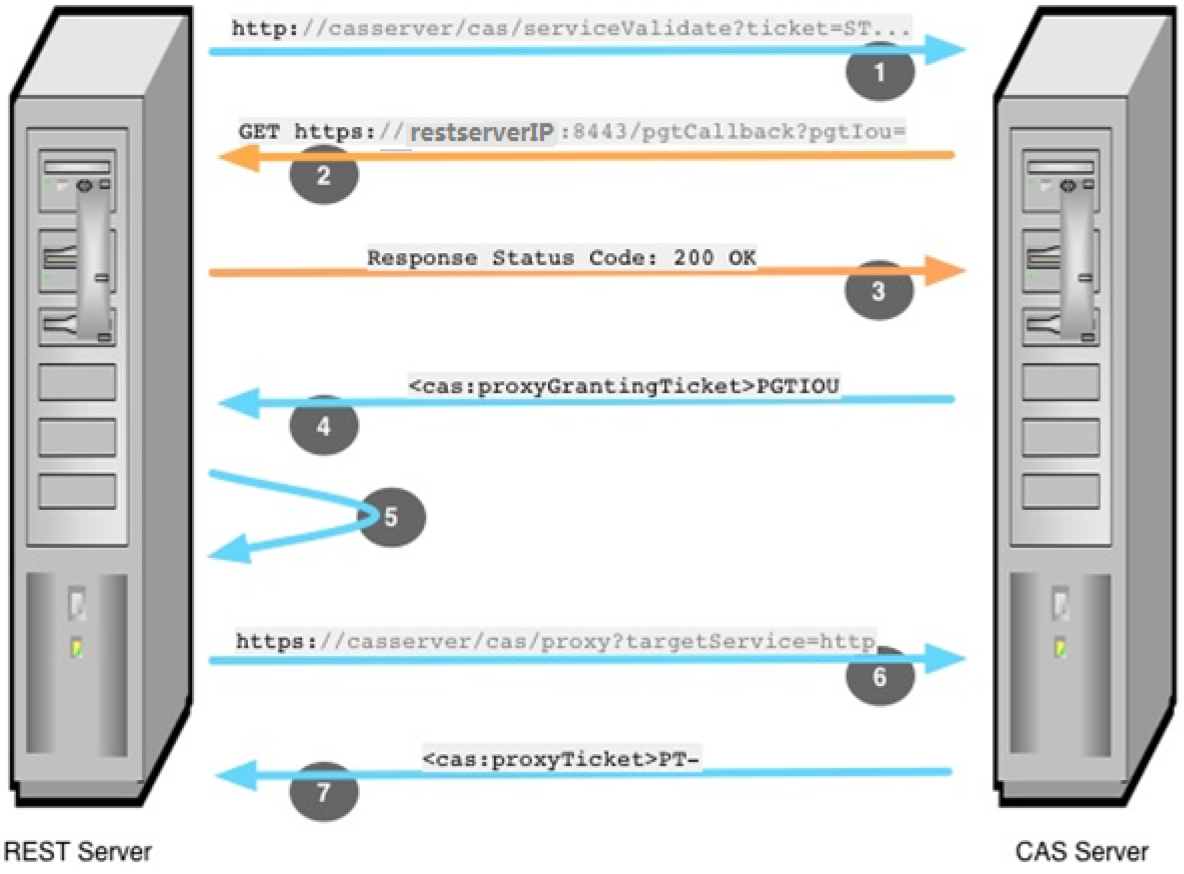
   Note: The value of `service` in the request body must to be URL encoded.

7. The client sends a POST request to the REST server to consume the resource, with the ST appended in the `ticket` parameter.

   ```
   =========== Request ===============
   GET http://192.168.0.1:8080/emc-rest/repositories/acme01
   ?ticket=ST-239-rYiYHoQJ2ZhJopdMuxjl-cas01.example.org
   Host: 192.168.0.1:8080
   ```

8. The rest of the process is the same with steps 7 through 14 in Authentication negotiation between a browser REST client and the REST server, page 87.

In CAS authentication, the REST server negotiates a CAS PT on behalf authenticated CAS clients for Content Server access. The following diagram illustrates the workflow of the PT negotiation between the REST server and the CAS server:


http://casserver/cas/serviceValidate?ticket=ST...
1
GET https:// restserverIP :8443/pgtCallback?pgtIou=
2
Response Status Code: 200 OK
3
<cas:proxyGrantingTicket>PGTIOU
4
5
https://casserver/cas/proxy?targetService=http
6
<cas:proxyTicket>PT-
7
REST Server
CAS Server

Proxy Ticket Negotiation between REST and CAS

1. To obtain the PGT IOU, the REST server calls the CAS server to validate the ST.

```
GET http://casserver/cas/serviceValidate
    ?ticket=ST-238-mHMdsK0A9sAhie2T1dep-cas01.example.org
    &service=http://192.168.0.1:8080/dctm-rest/repositories/acme01
    &pgtUrl=https://192.168.0.1:8443/cas/proxy/receptor
Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Cookie  DOCUMENTUM-CLIENT-TOKEN="Ym9ibGVlOkRNX1RJQ0tFVD1UMEpL...=="
```

2. The CAS server validates the ST and makes a callback to the address specified in the `pgtUrl` parameter of the REST request, passing the `pgtIou` (PGT IOU) and `pgtId` parameters.

   Note: `pgtUrl` must use HTTPS. The CAS server checks whether the SSL certificate of `pgtUrl` is valid and whether the name matches that of the requested service.

3. The callback service (`pgtUrl`) responds the CAS server with `HTTP 200`.

4. The CAS server responds to the initial request for the PGT IOU:

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
  <cas:authenticationSuccess>
```

```
      <cas:user>halbej</cas:user>
        <cas:proxyGrantingTicket>
          PGTIOU-85-8PFx8qipjkWYDbuBbNJ1roVu4yeb9WJIRdngg7fzl523Eti2td
        </cas:proxyGrantingTicket>
      </cas:authenticationSuccess>
    </cas:authenticationSuccess>
  </cas:serviceResponse>
```

5. The REST server retrieves the PGT (`pgtId`) from the callback service by providing the PGT IOU.
6. In order to obtain a PT, the REST server passes the PGT to the CAS server.

   ```
   GET https://casserver/cas/proxy
     ?targetService=http://192.168.0.1:8080/dctm-rest/repositories/acme01
   &pgt=PGT-330-CSdUc5fCBz3g8KDDiSgO5osXfLMj9sRDAI0xDLg7jPn8gZaDqS
   ```

7. The CAS server responds the REST server with a PT.

   ```
   <cas:serviceResponse>
     <cas:proxySuccess>
       <cas:proxyTicket>ST-957-ZuucXqTZ1YcJw81T3dxf</cas:proxyTicket>
     </cas:proxySuccess>
   </cas:serviceResponse>
   ```

For more information about CAS proxy, refer to the following article:

https://wiki.jasig.org/display/CAS/Proxy+CAS+Walkthrough

CAS Single Sign Out

CAS authentication supports single sign-out that invalidates client token cookies. When a client explicitly logs out, the session is terminated and the client has to negotiate a new session ticket.

The following workflow explains the single sign-out process in more detail:

1. A client sends a request to the REST server for logout by providing a CT.

   ```
   GET https://rest-server:8443/cas/logout HTTP/1.1
   Cookie : DOCUMENTUM-CLIENT-TOKEN= AYQEVn....DKrdst
   ```

2. The REST server validates the CT and resets it, and then redirects the REST client to the CAS server for logout.

   ```
   HTTP/1.1 302 302 Moved Temporarily
   Set-Cookie: DOCUMENTUM-CLIENT-TOKEN= ""; Expires=Thu, 01-Jan-1970 00:00:10 GMT;
   Path=/dctm-rest; HttpOnly; Secure;
   Location: https://cas-server:8443/cas/logout
   ```

3. The REST client resets the client side CASTGC cookie and access the CAS server for logout.

   ```
   GET https://cas-server/cas/logout HTTP/1.1
   Cookie : CASTGC= TGT-AYQEVn....DKrdst
   ```

4. The CAS server destroys the TGT from its memory entry and sends back `HTTP 200` with an empty CASTGC cookie.

   ```
   HTTP/1.1 200 OK
   Set-Cookie: CASTGC= ""; Expires=Thu, 01-Jan-1970 00:00:10 GMT; Path=/cas/
   ```

5. The REST client rests its client side TGT cookie, and the single sign-out finishes. Both TGT and CT are invalidated.

Configuration

CAS proxy authentication on Documentum REST Services requires configurations on the following servers:

- Content Server, page 94
- CAS Server, page 95
- REST Server, page 98
- Reverse Proxy Server, page 100

Content Server

To Enable CAS authentication for Documentum REST Services, the following configurations are required on Content Server:

1. Install the CAS authentication plug-in on Content Server. To do this, perform the following configurations:
   - Copy `dm_cas_auth.dll` to the authentication plug-in directory. Typically, the directory is `$DOCUMENTUM/dba/auth`.
   - Create a CAS plug-in configuration file (`dm_cas_auth.ini`) under `%DOCUMENTUM%\dba\auth`, and then add the following properties in the file:

     ```
     #This is a sample configuration file for the Documentum/CAS auth plugin

     [DM_CAS_AUTH_CONF]
     # Server host is the domain or host which is used in connecting to CAS
     # server.
     server_host=<host_ip>

     server_port=<port_number>

     #url path used in http requests sent to the CAS server to validate proxy ticket.
     url_path=/<cas_application_name>/proxyValidate

     # Target Service name for which the proxy ticket was generated.
     service_param=ContentServer

     # Specifies whether or not to set up the connection over https
     is_https=T
     ```
   - Restart Content Server.

   If the plug-in is loaded successfully, the log file (`$DOCUMENTUM/dba/log/<docbase>.log`) contains an entry starting with `DM_SESSION_I_AUTH_PLUGIN_LOADEDinfo`.
2. Synchronize LDAP users to Content Server repositories. For more information about how to synchronize LDAP users to Content Server repositories, see the section "Configuring LDAP synchronization for Kerberos users" in the *EMC Documentum Content Server Administration and Configuration Guide*.
3. If you have multiple repositories configured in your environment, set up trust relationships across repositories.

For more information about how to set up trust relationships across repositories, refer to the following sections:

- The “Managing the login ticket key” section in the *EMC Documentum Content Server Administration and Configuration Guide*
- The “Trusting and trusted repositories” section in the *EMC Documentum Content Server Fundamentals Guide*

CAS Server

To Enable CAS authentication for Documentum REST Services, the following configurations are required on the CAS server:

1. Download CAS Sever 3.5.2 from http://www.jasig.org/cas/download/cas.
2. Add the following dependencies to the corresponding `pom.xml` file under `\cas-server-uber-webapp` or `\cas-server-webapp`, depending on which CAS WAR you modify.

```
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-support-ldap</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-support-generic</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-integration-restlet</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-integration-ehcache</artifactId>
  <version>${project.version}</version>
</dependency>
```

3. To build a CAS WAR package, run the following maven command:

```
mvn clean install -DskipTests=true
```

4. Add the following servlet mapping into `\WEB-INF\web.xml` of the CAS WAR package you built in step 3:

```
<servlet>
    <servlet-name>restlet</servlet-name>
    <servlet-class>com.noelios.restlet.ext.spring.
    RestletFrameworkServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>restlet</servlet-name>
    <url-pattern>/v1/*</url-pattern>
</servlet-mapping>
```

For more information about CAS RESTful API, visit the following web site:https://wiki.jasig.org/display/CASUM/RESTful+API

5. Update the server name and host name in `WEB-INF/cas.properties`.

6. Register Content Server as a CAS proxy service by using one of the following methods.

   Method 1: Modify the in-memory service registry

   To do this, open `WEB-INF/deployerConfigContext.xml`, and add the following bean to `ServiceRegistryDao`:

```
<bean class="org.jasig.cas.services.RegexRegisteredService">
  <property name="id" value="1" />
  <property name="name" value="HTTP and IMAP on acme.com" />
  <property name="description" value="Allows HTTP(S) and IMAP(S)
  protocols on acme.com" />
  <property name="serviceId" value="ContentServer" />
  <property name="evaluationOrder" value="1" />
  <property name="allowedAttributes">
   <list>
    <value>dmCSLdapUserDN</value>
   </list>
  </property>
  </bean>
```

   You must set `serviceId` to `ContentServer`. Additionally, you must set the `allowedAttributes` property with a list of attributes.

   Method 2: Register Content Server by using the Service Management application

   - Create an administrative user for the Service Management application of CAS. To do this, edit the `userDetailsService` bean in `WEB-INF/deployerConfigContext.xml` as shown in the following sample:

```
<sec:user-service id="userDetailsService">
   <sec:user name="admin" password="notused" authorities="ROLE_ADMIN"/>
</sec:user-service>
```

   - Access `<cas_server_url>/services` with the administrative user. A Page with the list of services added to Service Registry is displayed.
   - Click `Add New Service` or access `<cas_server_url>/services/add.html`.
   - Fill the form to create a new service for Content Server and then save the service. In the `Service URL` field, you must enter the string you specified for the `service_param` parameter in the CAS plug-in configuration file (`dm_cas_auth.ini`).

7. Connect the CAS server to a directory service, such as active directory.

   To do this, locate and open `WEB-INF/deployConfigContext.xml`, and then add the following element in the `authenticationHandlers` list:

```
<bean class="org.jasig.cas.adaptors.ldap.BindLdapAuthenticationHandler"
   p:filter="sAMAccountName=%u"
   p:searchBase="OU=testou,DC=iigplat,DC=com"
   p:contextSource-ref="contextSource"
   p:ignorePartialResultException="true"/>
```

Note: `sAMAccountName` is used in active directory only. For LDAP servers, such as ApacheDS, replace it with `uid`.

Still in `deployConfigContext.xml`, add the following element:

```
<bean id="contextSource"
class="org.springframework.ldap.core.support.LdapContextSource">
   <property name="pooled" value="false"/>
   <property name="url" value="ldap://domainctlr.iigplat.com:389" />
   <property name="userDn"
      value="CN=Administrator,CN=Users,DC=iigplat,DC=com"/>
   <property name="password" value="password"/>
   <property name="baseEnvironmentProperties">
   <map>
      <entry key="com.sun.jndi.ldap.connect.timeout" value="3000" />
      <entry key="com.sun.jndi.ldap.read.timeout" value="3000" />
      <entry key="java.naming.security.authentication"
value="simple" />
   </map>
   </property>
</bean>
```

8. Customize CAS responses.

   By default, the CAS server only responds with user names for PT validation requests. You must customize CAS responses to include user distinguished names because Content Server utilizes this information for user verification.

   To do this, perform the following configurations:

   - In `WEB-INF/deployerConfigContext.xml`, add the following element in the `CredentialsToPrincipalResolvers` list for LDAP:

     ```
     <bean class="org.jasig.cas.authentication.principal.
     CredentialsToLDAPAttributePrincipalResolver">
        <property name="credentialsToPrincipalResolver">
            <bean class="org.jasig.cas.authentication.principal.
            UsernamePasswordCredentialsToPrincipalResolver"/>
        </property>
        <property name="filter" value="(sAMAccountName=%u)"/>
        <property name="principalAttributeName" value="sAMAccountName"/>
        <property name="searchBase" value="OU=testou,DC=iigplat,DC=com"/>
        <property name="contextSource" ref="contextSource"/>
        <property name="attributeRepository" ref="attributeRepository"/>
     </bean>
     ```

     A principal describes an authenticated user. The principal contains attributes describing the user. The `CredentialsToPrincipalResolver` bean maps credential attributes to principal attributes. Principals are used by View to create responses with user attributes defined in the `AttributeRepository` bean.

   - In `WEB-INF/deployerConfigContext.xml`, modify the `attributeRepository` bean that defines the attributes that the CAS server returns to Content Server and add the `dmCSLdapUserDN` attribute whose value will be set to the user distinguished name as shown in the following sample:

     ```
     <bean id="attributeRepository" class="org.jasig.services.persondir.
     support.ldap.LdapPersonAttributeDao">
        <property name="contextSource" ref="contextSource" />
        <property name="baseDN" value="OU=testou,DC=iigplat,DC=com" />
        <property name="requireAllQueryAttributes" value="true" />
        <property name="queryAttributeMapping">
     ```

```
      <map>
         <entry key="username" value="sAMAccountName" />
      </map>
   </property>
   <property name="resultAttributeMapping">
      <map>
         <entry value="dmCSLdapUserDN" key="distinguishedName"/>
      </map>
   </property>
</bean>
```

Note:

— The `username` key in `queryAttributeMapping` is used in active directory only. For LDAP servers, such as ApacheDS, replace it with `uid` as the key.

— The `distinguishedName` key in `resultAttributeMapping` is used in active directory only. For LDAP servers, such as ApacheDS, replace it with `entryDN` as the key.

- Update View to include the user distinguished name in responses sent to Content Server for PT validation requests. To do this, locate and open `WEB-INF/view/jsp/protocol/2.0/casServiceValidationSuccess.jsp`, and then append the following elements after `<cas:user>` ... `</cas:user>`:

```
<c:forEach var="auth" items="${assertion.chainedAuthentications}">
   <c:forEach var="attr" items="${auth.principal.attributes}" >
      <cas:attribute name="${fn:escapeXml(attr.key)}"
                     value="${fn:escapeXml(attr.value)}"/>
   </c:forEach>
</c:forEach>
```

9. Set up domain users.
10. Create a CAS server certificate.
11. Import the CAS server certificate to the CAS server's JRE keystore with the following command:
    `keytool -import -keystore <path-of-the-jre-cacert> -storepass "<password>" -alias "<cas-alias>" -file <path-of-the-cas-certificate -file>`
12. Import the REST server certificate to the CAS server's JRE truststore with the following command:
    `keytool -import -trustcacerts -alias "<rest-alias>" -file <path-of-the-rest-certificate-file> -keystore <path-of-the-jre-cacert>`
13. Enable HTTPS on the web container where you plan to deploy the CAS WAR file.
14. Deploy the CAS WAR file on the web container.

REST Server

To Enable CAS authentication for Documentum REST Services, the following configurations are required on the REST server:

1. Create a REST server certificate.
2. Import the REST server certificate to the CAS server JRE keystore with the following command:
   `keytool -import -keystore <path-of-the-jre-cacert> -storepass`

"<password>" -alias "<rest-alias>" -file <path-of-the-rest-certificate -file>

3. Import the CAS server certificate into JRE keystore of the REST server with the following command:
   keytool -import -trustcacerts -alias "<cas-alias>" -file <path-of-the-cas-certificate-file> -keystore <path-of-the-jre-cacert>

4. Update `server.xml` of the application server where Documentum REST Services is deployed to enable HTTPS.

   For example, in Tomcat, incorporate the following content into `server.xml`.

   ```
   <Connector port="8443" protocol="HTTP/1.1"
                  maxThreads="150" scheme="https" secure="true"
                  clientAuth="false" sslProtocol="TLS"
         keystoreFile="path-of-the-rest-certificate-file.jks"
         keystorePass="password"
         keyAlias="rest-alias"
         truststoreFile="path-of-the-cas-certificate-file.jks"
         truststorePass="password"/>
   ```

   If the REST server is placed behind a reverse proxy server, you do not have to set SSL on the REST server. The setting can be configured on the proxy server. For more information about how to configure this setting on a proxy server, see Reverse Proxy Server, page 100.

5. For Linux operating systems, if client tokens are used in your deployment, add the following option to the startup script the application server where Documentum REST Services is deployed to achieve better performance:

   ```
   JAVA_OPTS="$JAVA_OPTS -Djava.security.egd=file:/dev/./urandom"
   ```

6. Locate and open `rest-api-runtime.properties` , and then update the following security properties according to the instructions in the file.

   - rest.security.auth.mode
   - rest.security.realm.name
   - rest.security.cas.server.url
   - rest.security.cas.server.login.url
   - rest.security.cas.server.logout.url
   - rest.security.cas.server.tickets.url
   - rest.security.cas.proxy.service
   - rest.security.server.url
   - rest.security.cas.callback.service.url
   - rest.security.auth.cas.client.pgt.storage

   Note: The default settings of these properties work in most cases. Keep the original settings unless you have special business requirements.

Reverse Proxy Server

For more information about the configurations on the reverse proxy server, see Reverse Proxy Configuration, page 115

CAS Server Clustering

You can deploy CAS authentication in a clustering environment to achieve high availability (HA).

Follow the instructions on the following web site to configure CAS clustering.

https://wiki.jasig.org/display/CASUM/Clustering+CAS

Note: If CAS clustering utilizes Ehcache to make all nodes in the cluster recognize and validate each other's tickets, make the following modifications:

- In `${CAS}\WEB-INF\spring-configuration\ticketRegistry.xml`, set `shared` of the `cacheManager` bean to `true`.
- The default Ehcache configuration recommends that you set TGT/PGT replication in async mode. However, in Documentum REST Services, an ST/PT ticket request may happen immediately after the TGT/PGT generation (in milliseconds). Therefore, we strongly recommend that you use sync mode for both ST and TGT replications.

REST Server Clustering

Similar to CAS, you can deploy REST servers in a clustering environment. A reverse proxy server is placed in front of the REST server cluster.

During the PT negotiation between the REST server and CAS server, the CAS server has to make a callback to the REST server requesting the PGT (see Proxy Ticket Negotiation between REST and CAS, page 91). If you deploy REST servers in a cluster, the callback may not find the REST server requesting the PGT. Therefore, all REST servers must maintain the same PGT IOU/PGT mappings. To do this, REST servers utilizes Ehcache to perform the replication of PGT IOU/PGT mappings across the cluster.

To enable Ehcache for PGT IOU/PGT mappings in Documentum REST Services, the following setting must be configured in `rest-api-runtime.properties`:

```
rest.security.auth.cas.client.pgt.storage=ehcache
```

For more information about this property, see the instruction in `rest-api-runtime.properties`.

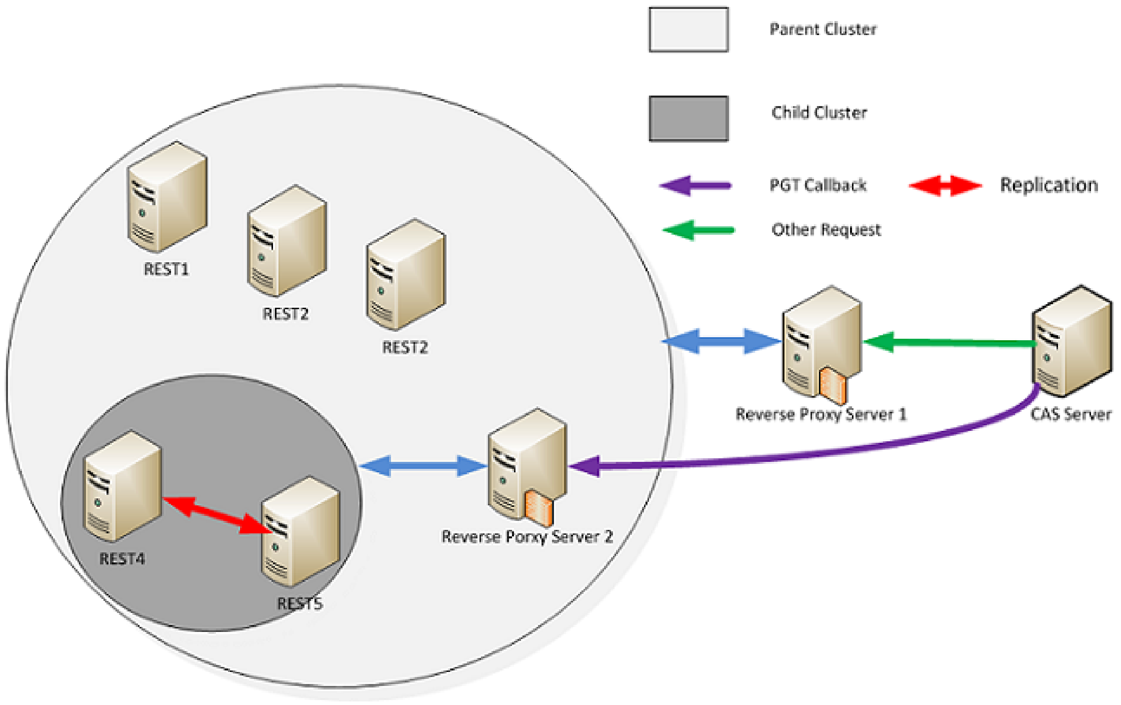
Additionally, you must follow the instructions on the following web site to configure Ehcache in `<dctm-rest>\WEB-INF\classes\ehcache-cas.xml`.

http://ehcache.org/documentation/replication/rmi-replicated-caching

Performance Consideration

The replication of PGT IOU/PGT mappings across the whole cluster may degrade performance, especially when the number of REST server is large. In this case, you can create a child cluster behind a reverse proxy server and limit the replication of PGT IOU/PGT mappings within the child cluster, then you have to designate the child cluster to handle PGT callbacks. This method reduces the amount of replication and thus improves performance.

The following diagram illustrates the network topology of this method:



This method requires the following configurations:

- For all REST servers in the parent cluster, the callback URL must be set to the address of the reverse proxy server that is placed in front of the child cluster. (In the diagram, Reverse Proxy Server 2)
- For all REST servers in the child cluster, the following setting must be configured in `rest-api-runtime.properties`:

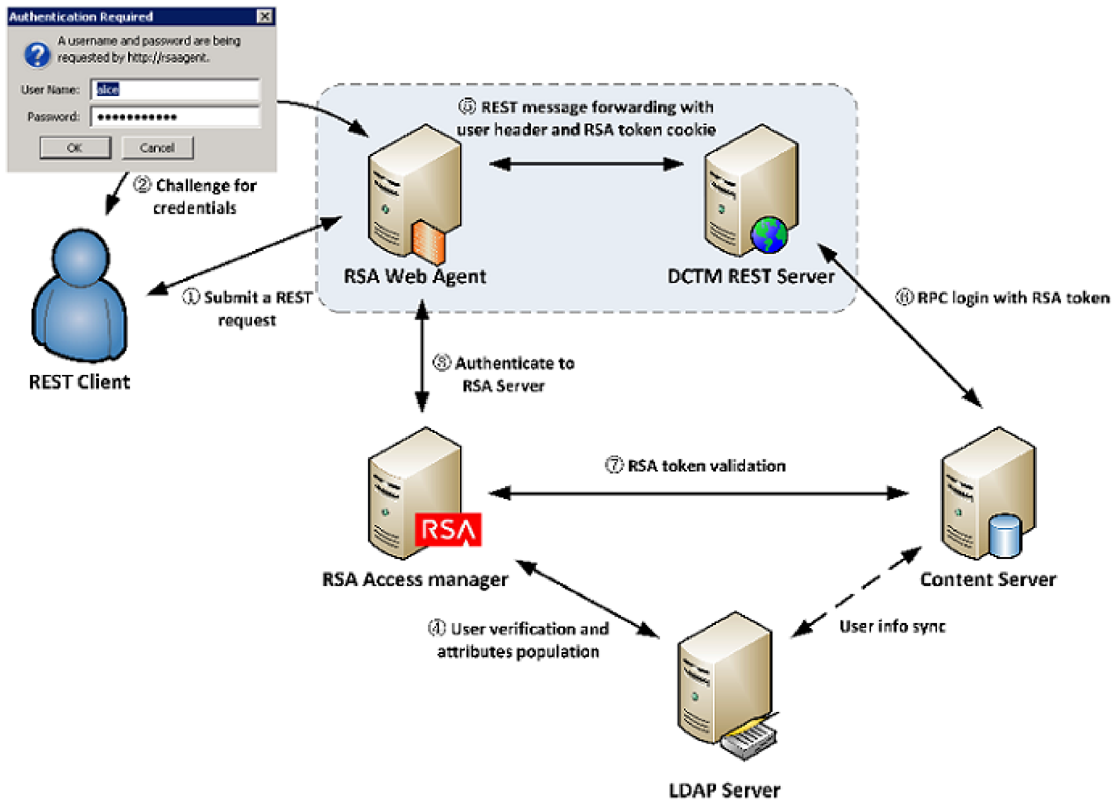
  ```
  rest.security.auth.cas.client.pgt.storage=ehcache
  ```
- You must configure peer discovery for all REST servers in the child cluster and all of these servers must be set as peers.

RSA Authentication

Documentum Content Server has native RSA Access Manager (originally named as ClearTrust) SSO support. In RSA Access Manager SSO authentication, an RSA web agent (usually running as a module of a web server, such as Apache and IIS) runs as a reverse proxy to filter resource requests and dispatch authentication requests. The resource server (Documentum REST Services) runs behind the entry point of the RSA web agent so that these two parties are within a sub trusted system.

Authentication workflow

The following diagram illustrates the workflow of RSA Access Manager SSO authentication.


Authentication Required
A username and password are being requested by http://rsaagent.
User Name: alice
Password: ••••••••••••
OK
Cancel
② Challenge for credentials
① Submit a REST request
REST Client
③ REST message forwarding with user header and RSA token cookie
RSA Web Agent
DCTM REST Server
⑥ RPC login with RSA token
⑤ Authenticate to RSA Server
⑦ RSA token validation
RSA Access manager
Content Server
④ User verification and attributes population
User info sync
LDAP Server

RSA Authentication Workflow

1. A REST client submits a resource request to the RSA web agent: http://rsa-web-agent/dctm -rest/repositories/acme01.

   Note: The client does not communicate with the REST server directly because the REST server is protected by the RSA web agent.

2. The RSA web agent challenges the user credentials if the client did not provide in step 1.RSA Access Manager supports many authentication types, such as, HTTP Basic, Client Certificate, and Smart Card. You can configure RSA Access Manager to specify the authentication type to negotiate the credentials between the client and the RSA server. The REST client must be able to handle the credential challenge and submit the corresponding credentials to the RSA server.

3. The RSA web agent dispatches the credentials to the RSA Access Manager.Upon a successful login, The RSA web agent retrieves a list of user attributes together with a ClearTrust token from the RSA Access Manager.
4. The RSA Access Manager communicates with the back-end user data store to verify the credentials.The back-end user data store can be an LDAP server or a database server depending on the RSA Access Manager configuration.
5. The RSA web agent forwards the original REST resource request to the REST server.The RSA ClearTrust token is sent to the REST server as a cookie. Additionally, a list of user attributes is populated to the REST server as HTTP headers. Among these headers, the remote user name header is required. The REST server uses this header to log in to Content Server.
6. The REST server uses the RSA token and user name to log in to Content Server.A session is created for the REST server to perform further resource operations.
7. Content Server validates the RSA token against the RSA Access Manager.Upon a successful validation, Content Server returns a session to the REST server. Content Server also has to know from which RSA Access Manager host this token was obtained. Only the hosts in the trusted RSA servers are allowed.

Finally, the REST server returns the requested resource to the REST client. When forwarding the resource back to the client, the RSA web agent appends an RSA token to the response as a cookie. Therefore, the client can reuse the RSA token for further resource requests.

Configuration

RSA authentication on Documentum REST Services requires configurations on the following servers:

- Configuration on the REST Server, page 104
- Configuration on Content Server, page 104
- Configuration on the RSA Web Agent, page 105
- Configuration on the RSA Administrative Console, page 105

Configuration on the REST Server

To enable RSA authentication, the following configurations are required on the REST server:

1. Open `/WEB-INF/classes/rest-api-runtime.properties` and set the following parameters according to the instructions in the file:

   ```
   rest.security.auth.mode=rsa
   # rest.security.rsa.cleartrust.cookie.name=
   # rest.security.rsa.user.header.list=
   # rest.security.rsa.dispatcher.address=
   ```

   The `rest.security.rsa.cleartrust.cookie.name`, and `rest.security.rsa.user .header.list` parameters are commented out in `rest-api-runtime.properties`. These parameters must be uncommented even if you want to keep the default settings. To use the default settings, uncomment these parameters and leave them blank. Additionally, you must uncomment the `rest.security.rsa.dispatcher.address` parameter and specify it with a non-empty value.

2. Start the REST server.

Configuration on Content Server

To enable RSA authentication, the following configurations are required on Content Server:

1. Install the RSA plug-in.
   To do this, navigate to `%DM_HOME%\install\external_apps\authplugins\rsa`, and then follow the steps in `readme.txt` to install the plug-in.

2. Synchronize LDAP users to Content Server repositories. For more information about how to synchronize LDAP users to Content Server repositories, see the section "Configuring LDAP synchronization for Kerberos users" in the *EMC Documentum Content Server Administration and Configuration Guide*.

In some cases, an LDAP user that has been authenticated to RSA may encounter a token validation failure against Content Server, and the REST server may return the following message:

```
Status Code: 401 Unauthorized
Content-Type: application/json;charset=UTF-8
WWW-Authenticate: RSA realm="ACME.COM"
```

```
{
  "status": 401,
  "code": "E_BAD_CREDENTIALS_ERROR",
  "message": "Authentication failed because an invalid credential is provided.",
  "details": "User authentication has been passed in RSA Access Manager
             but the clearTrust cookie validation is failed in Content Server;
             (DM_SESSION_E_AUTH_FAIL) Authentication failed for user tom with
             docbase acme01."
}
```

This problem mainly occurs when the LDAP user is not synchronized to Content Server, or the RSA plug-in is not correctly installed.

Configuration on the RSA Web Agent

On the HTTP server that the RSA web agent is deployed, you must configure a reverse proxy for REST Services. Here is an example in Apache HTTP Server 2.x:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_http_module modules/mod_proxy_http.so

ProxyRequests Off
ProxyPreserveHost On
ProxyPass /dctm-rest http://localhost:8080/dctm-rest
ProxyPassReverse /dctm-rest http://localhost:8080/dctm-rest
```

Optionally, you may need to update the RSA web agent for the REST deployment by modifying the following parameters in `${RSA_AGENT}\conf\webagent.conf`:

```
cleartrust.agent.cookie_name=
cleartrust.agent.user_header_list=
cleartrust.agent.exported_headers=
cleartrust.agent.cookie_touch_window=
cleartrust.agent.form_based_enabled=False
cleartrust.agent.auth_resource_list=/*=BASIC
```

Note: The parameter `cleartrust.agent.cookie_touch_window` determines the amount of time the web agent will wait before updating the cookie for an authenticated user. The cookie value has an inner relationship with the authenticated DFC session manager. To get better performance, we recommend that you set a longer touch window so that the session manager does not need to re-authenticate frequently.

The RSA web agent runs as a reverse proxy to filter resource requests and dispatches authentication requests. Follow the instructions in Reverse Proxy Configuration, page 115 to configure the RSA web agent.

Configuration on the RSA Administrative Console

You must register REST resources into RSA web agent. To do this, follow these steps on the RSA Administrative Console:

1. Add a new server for the RSA Web Agent, for example **`Apache HTTP Server`**.

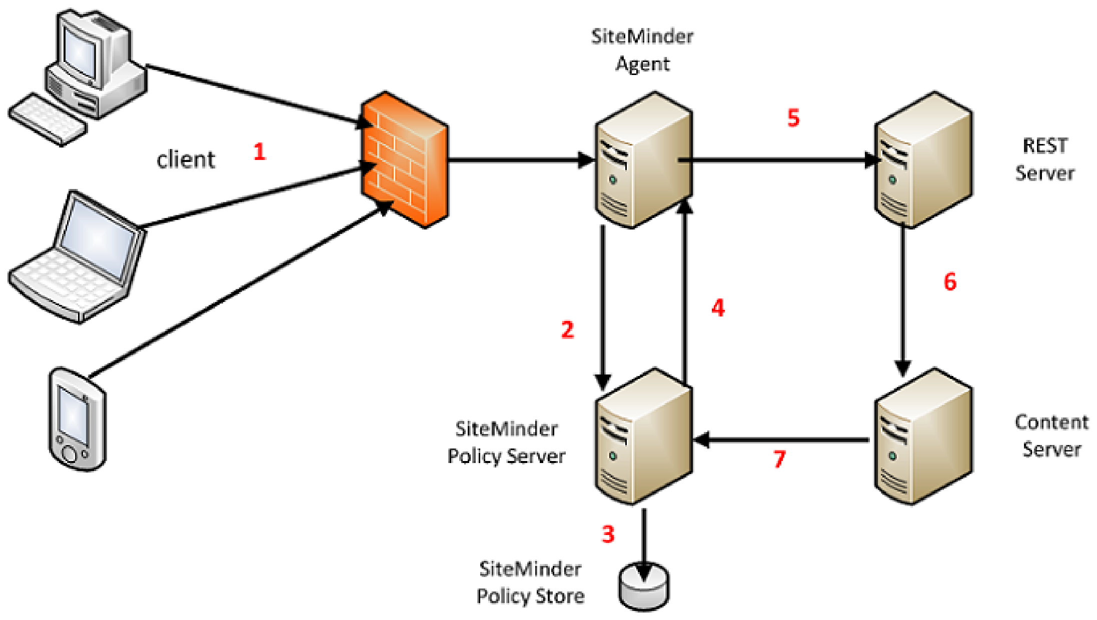
2. Create a new application, for example **`CoreREST`**,
3. Add a new resource to the application you created in step 2 with the resource type **`URL`** and set URL to **`dctm-rest/repositories/*`**
4. Navigate to the Entitlements settings of the user or groups that the user belongs to, and select **`Allow Access`** on the application and resource.

Siteminder Authentication

Documentum REST Services supports CA SiteMinder authentication to provide secure Single Sign-On (SSO) and protect resources in Content Server.

Authentication workflow

The following diagram illustrates the workflow of SiteMinder authentication:



Siteminder Authentication Workflow

1. A client sends a request to the SiteMinder agent (a reverse proxy server for the REST server). The agent challenges the client for credentials if the client does not provide.
2. The agent passes the credentials to the SiteMinder policy server.
3. The policy server validates user credentials according to the information in the policy store which can be an LDAP server or a database server.
4. After the validation, the user name and password token is sent back to the SiteMinder agent.
5. The SiteMinder agent forwards the client request together with the HTTP headers, which include the token, to the REST server.
6. The Rest server sends the token to Content Server.
7. Content Server validates the token by using the netegrity plug-in.

After the validation of the token, the REST server returns the response to the SiteMinder agent. Then, the agent forwards the response to the client.

Configuration

A SiteMinder environment includes multiple components. This section introduces the configurations that you must perform on the REST server and Content Server.

- Configuration on the REST Server, page 108
- Configuration on Content Server, page 108

For the configurations of other components that are involved in a SiteMinder environment, such as policy servers and policy stores, refer to CA SiteMinder documentation.

Configuration on the REST Server

To enable SiteMinder authentication, the following configurations are required on the REST server:

1. Open `/WEB-INF/classes/rest-api-runtime.properties` and set the following parameters according to the instructions in the file:

   ```
   rest.security.auth.mode=siteminder
   # rest.security.siteminder.cookie.name=
   # rest.security.siteminder.user.header=
   ```

   The `rest.security.siteminder.cookie.name` and `rest.security.siteminder.user.header` parameters are commented out in `rest-api-runtime.properties`. These parameters must be uncommented even if you want to keep the default settings. To use the default settings, uncomment these parameters and leave them blank.

2. Start the REST server.

Configuration on Content Server

To enable SiteMinder authentication, the following configurations are required on Content Server:

1. Install the SiteMinder plug-in.
   To do this, navigate to `%DM_HOME%\install\external_apps\authplugins\netegrity`, and then follow the steps in `readme.txt` to install the plug-in.

2. Synchronize LDAP users to Content Server repositories. For more information about how to synchronize LDAP users to Content Server repositories, see the section "Configuring LDAP synchronization for Kerberos users" in the *EMC Documentum Content Server Administration and Configuration Guide*.
   If the policy store is set up on a database, you must manually synchronize users to Content Server. In this case, the login name must be the username in the database, and the user source must be set to `dm_netegrity`.

Special Considerations

You may need to take the following issues into consideration when configuring SiteMinder authentication:

Consideration 1: SessionGracePeriod Setting

On the SiteMinder web agent, the `SessionGracePeriod` parameter specifies the number of seconds during which a SiteMinder session (SMSESSION) cookie will not be regenerated. By default, this parameter is set to `30`. Every time the SMSESSION value changes a new session manager is created because the SMSESSION cookie value is used as a part of the session manager key. Therefore, we recommend that you increase the value of this parameter to achieve better performance.

Consideration 2: Reverse Proxy Configuration

For more information about the configurations on the reverse proxy server, see Reverse Proxy Configuration, page 115.

Consideration 3: CssChecking setting

By default, the SiteMinder web agent protects web sites against Cross-Site Scripting by setting the `CssChecking` parameter to `yes`. In this case, URLs containing the following characters do not work:

- single quote (')
- left and right angle brackets (< and >)

To use these characters in a URL, disable CSS Checking by setting `CssChecking` to `no` in the SiteMinder web agent configuration object.

Consideration 4: Cache-related HTTP Headers

Content-media resources support the cache mechanism by using HTTP headers (the `etag` header in a response and the `if-none-match` header in a request). However, the default behavior of the SiteMinder web agent removes the cache-related HTTP headers from the request before the web agent passes a request to the REST server, which causes the REST server always return HTTP 200 when you try to retrieve a content-media resource.

To use cache-related HTTP headers, you must set the `AllowCacheHeaders` parameter to `yes` in the SiteMinder web agent configuration object.

Consideration 5: BadQueryChars and BadUrlChars Setting

Documentum REST Services does not have any restriction on characters used in URLs. However settings of the `BadUrlChars` and `BadUrlChars` parameters may cause restrictions on characters. Consider modifying these parameters when a request is rejected for bad URL character reasons.

Consideration 6: MaxUrlSize Setting

Documentum REST Services does not limit the length of URLs that a web agent can handle. However, the `MaxUrlSize` parameter specifies the maximum size (in bytes) of a URL, which defaults to 4096. Consider increasing the value of this parameter when a request is rejected for long URL reasons.

Multiple Authentication Schemes

Documentum REST Services allows you to enable multiple authentication schemes concurrently in your production environment. You can specify which scheme or combination of schemes to use by modifying `rest.security.auth.mode` in `<dctm-rest>\WEB-INF\classes\rest-api-runtime.properties`. In this release, the following combinations of authentication schemes are supported:

- HTTP Basic and Kerberos (`rest.security.auth.mode=basic-kerberos`)
- HTTP Basic and Kerberos with client tokens (`rest.security.auth.mode=basic-ct-kerberos`)
- Kerberos with client tokens (`rest.security.auth.mode=ct-kerberos`)
- HTTP Basic and CAS with client tokens (`rest.security.auth.mode=basic-ct-cas`)
- CAS with client tokens (`rest.security.auth.mode=ct-cas`)
- New in 7.2: HTTP Basic with client tokens and Kerberos with client tokens (`basic-dual_ct-kerberos`)

When communicating with a REST server that has multiple authentication schemes configured, a REST client relies on the `WWW-Authenticate` header in the server's response to discover the authentication schemes the server offers.

- When the client tries to access a resource without an `Authorization` header or with an `Authorization` header that does not match any of the supported authentication schemes, the REST server returns a 401 status code with supported schemes specified in the `WWW-Authenticate` headers.
- When the client tries to access a resource with a matching `Authorization` header, the REST server authenticates the client with the corresponding authentication scheme (other supported schemes are ignored). If the authentication successes, the server returns the requested resource. Otherwise, the server returns an authentication failure response corresponding to the specific scheme. The error message in the response body is specific to the scheme.
- When the client tries to access a resource with multiple matching `Authorization` headers, the REST server authenticates the client by using the matching scheme with the highest precedence. The following list shows the authentication schemes in order of precedence, with the highest-precedence one at the top.
  - HTTP Basic authentication header
  - Client token
  - Kerberos or CAS

  When the matching authentication scheme with the highest precedence fails, the REST server returns an authentication failure response.

Samples for Multi Authentication Schemes (HTTP Basic and Kerberos)

Example 5-5. No Authorization header

```
// request
GET /${resource-url} HTTP/1.1

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
WWW-Authenticate: Negotiate
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"Full authentication is required to access this resource"
}
```

Example 5-6. Authorization header not matching

```
// request
GET /${resource-url} HTTP/1.1
Authorization: CAS <TICKET>

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
WWW-Authenticate: Negotiate
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"Full authentication is required to access this resource"
}
```

Example 5-7. Matching basic Authorization header

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ

// response
HTTP/1.1 200 OK
// resource body
```

Example 5-8. Matching negotiate Authorization header

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Negotiate YIIZG1hZG1pbjpwYXNzd29yZ.....

// response
HTTP/1.1 200 OK
// resource body
```

Example 5-9. Matching basic Authorization header with bad credential

```
// request
GET /${resource-url} HTTP/1.1
```

```
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ++bad++credential

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
{
"status":401,
"code":"E_BAD_CREDENTIALS_ERROR",
"message":"Authentication failed because an invalid credential is provided.",
"details":"(DM_SESSION_E_AUTH_FAIL) error: \
"Authentication failed for user badboy with docbase space01.\""
}
```

Example 5-10. Matching negotiate Authorization header with bad credential

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Negotiate YIIZG1hZG1pbjpwYXNzd29yZ++bad++credential

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Negotiate
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"The given token is a NTLM token, not Kerberos token."
}
```

Samples for Multi Authentication Schemes (HTTP Basic and CAS)

Example 5-11. No Authorization header

```
// request
GET /${resource-url} HTTP/1.1

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
WWW-Authenticate: CAS <MY_REALM>
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"Full authentication is required to access this resource"
}
```

Example 5-12. Mismatching Authorization header

```
// request
GET /${resource-url} HTTP/1.1
Authorization: CAS <MY_REALM>

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
WWW-Authenticate: Negotiate
```

```
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"Full authentication is required to access this resource"
}
```

Example 5-13. Matching basic Authorization header

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ

// response
HTTP/1.1 200 OK
// resource body
```

Example 5-14. Matching CAS ticket

```
// request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ-CAS.EMC.COM HTTP/1.1

// response
HTTP/1.1 200 OK
// resource body
```

Example 5-15. Matching basic Authorization header with bad credential

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ++bad++credential

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
{
"status":401,
"code":"E_BAD_CREDENTIALS_ERROR",
"message":"Authentication failed because an invalid credential is provided.",
"details":"(DM_SESSION_E_AUTH_FAIL) error: \"Authentication failed for user
           badboy with docbase space01.\""
}
```

Example 5-16. Matching CAS ticket with bad credential (no redirect)

```
// request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ++bad++credential HTTP/1.1
DOCUMENUM-NO-CAS-REDIRECT: true

// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: CAS <MY_REALM>
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"ticket 'ST-29-HeJcl956dMmTttZhLBPZ++bad++credential' not recognized."
}
```

Example 5-17. Matching CAS ticket with bad credential (redirect)

```
// request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ++bad++credential HTTP/1.1
DOCUMENUM-NO-CAS-REDIRECT: false

// response
HTTP/1.1 302 Moved
Location: https://cas-server/cas/login
```

Reverse Proxy Configuration

If the REST server is placed behind a reverse proxy server, the following configurations are required on the proxy server. These configurations make the REST server generate correct links in a response.

- The proxy server must retain the original the HOST header when forwarding a request to the REST server, instead of modifying it.

  — If you deploy the proxy server on an Apache server, set `ProxyPreserveHost` to `on`.

  — If you deploy the proxy server on an Nginx server, use the following directive:

  ```
  proxy_set_header Host $host;
  ```

  — If you deploy the proxy server on an IIS server, you must use the URL Rewrite module to rewrite the URLs the REST server generates to the original HOST value. For more information about how to use the URL Rewrite module, visit the following web site:

  http://www.iis.net/learn/extensions/url-rewrite-module/using-the-url-rewrite-module

  For more information about how to set this header on other application servers, refer to the documentation of the application server.

- If the communication between the client and proxy server and that between the proxy server and rest server uses different protocols (for example, the client uses HTTPS to communicate with the proxy server while the proxy server uses HTTP to communicate with the REST server), unify the communication protocol by using the `X-Forwarded-Proto` header on the proxy server.

  On Nginx, use the following directive:

  ```
  proxy_set_header X-Forwarded-Proto <protocal>
  ```

  For example, the following directive forces the proxy server to use HTTPS to communicate with the REST server:

  ```
  proxy_set_header X-Forwarded-Proto https
  ```

  On Apache, add the following line in the virtual host configuration:

  ```
  RequestHeader set X-Forwarded-Proto <protocal>
  ```

  For more information about how to set this header on other application servers, refer to the documentation of the application server.

- In most cases, the port is included in the HOST header. In this case, you do not need to specify the port on the proxy server. Otherwise, unify the port by using the `X-Forwarded-Port` header on the proxy server if the proxy server and the REST server use different ports.

  On Nginx, use the following directive:

  ```
  proxy_set_header X-Forwarded-Port <Port>
  ```

  On Apache, add the following line in the virtual host configuration:

  ```
  RequestHeader set X-Forwarded-Port <Port>
  ```

  For more information about how to set this header on other application servers, refer to the documentation of the application server.

- Performance may degrade when many clients send requests cocurrently. To improve concurrent access performance, we recommend that you modify the multi-processing module of the proxy server according to the related documentation. For example, if you use an Apache-based proxy server on a Windows machine, you may need to increase the value of `ThreadsPerChild` and

MaxRequestsPerChild, or even remove the limitation on the number of requests that an individual child server handles by setting MaxRequestsPerChild to 0.

Configuration Samples

The following sample shows how to configure an Nginx-based reverse proxy server to generate correct links.

Example 5-18. Reverse Proxy Configuration on Nginx to Generate Correct Links

```
location / {
        proxy_pass              http://rest_servers;
        proxy_set_header        Host                $host;
        proxy_set_header        X-Forwarded-Proto   https;
        proxy_set_header        X-Forwarded-For     $proxy_add_x_forwarded_for;
        proxy_set_header        X-Forwarded-Port    443;
        proxy_set_header        X-Real-IP           $remote_addr;
        proxy_redirect          off;
    }
```

The following sample shows how to improve concurrent access performance for an Apache-based proxy server on a Windows machine.

Example 5-19. Reverse Proxy Configuration on Apache to improve concurrent access performance

```
<IfModule mpm_winnt_module>
ThreadsPerChild 512
MaxRequestsPerChild 0
</IfModule>
```

Client Token

When Client Token is enabled, the REST server generates a Client Token cookie after a successful Kerberos or CAS authentication and sends it back to the REST client in the response. The purpose of producing a client token is to provide an authenticated REST client with a temporary and expirable token to access the REST server without a need to negotiate a new session ticket. The Client Token cookie has no session state stored on the REST server. Therefore, it works in cluster environment as well.

A client token cookie is encrypted and validated by the REST server. The REST client is not expected in any means to persist or decrypt the token. A validation failure of the client token cookie leads to an authentication failure. In this case, the Basic, Kerberos or CAS token negotiation happens again.

The following example describes the authentication negotiation workflow between the client and server using Kerberos and Client Token cookie

Authentication using Kerberos and Client Token Cookie

1. A REST client sends a resource request with no credentials to the REST server.

   ```
   GET /${resource-url} HTTP/1.1
   ```

2. The REST server responds with the 401 status error and a Negotiate header.

   ```
   HTTP/1.1 401 Unauthorized
   WWW-Authenticate: Negotiate
   ```

3. The REST client negotiates a SPNEGO-based Kerberos token and re-sends the resource request.

   ```
   GET /${resource-url} HTTP/1.1
   Authorization: Negotiate YIIZG1hZG1pbjpwYXNzd29yZ.....
   ```

4. The REST server returns the requested resource and a Client Token cookie for reuse, identified by the cookie name `DOCUMENTUM-CLIENT-TOKEN`.

   ```
   HTTP/1.1 200 OK
   Set-Cookie: DOCUMENTUM-CLIENT-TOKEN="H5VaJz8Vn..."
   ```

5. The REST client sends a subsequent resource request with the Client Token cookie received in Step 4.

   ```
   GET /${other-resource-url} HTTP/1.1
   Cookies: DOCUMENTUM-CLIENT-TOKEN="H5VaJz8Vn..."
   ```

6. The REST server returns the requested resource.

   ```
   HTTP/1.1 200 OK
   ```

Enabling Client Tokens

By default, the security configuration in Documentum REST Services disables client tokens for Kerberos and CAS authentication. To enable client tokens, add `ct` to the `rest.security.auth.mode` property:

- `rest.security.auth.mode=ct-kerberos` (Kerberos authentication with client token cookie)
- `rest.security.auth.mode=basic-ct-kerberos` (HTTP Basic authentication and Kerberos authentication with client tokens)
- `rest.security.auth.mode=ct-cas` (CAS with client tokens)
- `rest.security.auth.mode=basic-ct-cas` (HTTP Basic and CAS with client tokens)

Documentum REST Services supports the following expiration policies for the client token cookie:

Policy	Description
com.emc.documentum.rest.security.ticket.impl .HardTimeoutExpirationPolicy	The client token expires after a specified duration. If the REST client sends a request before the duration, the REST server accepts the client token. If the REST client sends a request after the duration, the REST server rejects the client token, and the client has to authenticate again.

Policy	Description
com.emc.documentum.rest.security.ticket.impl .TolerantTimeoutExpirationPolicy (default)	The client token expires after two times of the specified duration. The REST server issues new client tokens under certain conditions. For details, see the following: • If the REST client sends a request before the duration, the REST server accepts the client token. • If the REST client sends a request after the duration and before two times of the duration, the REST server accepts the client token and issues another client token with the same duration to the client for subsequent requests. • If the REST client sends a request after two times of the duration comes to an end, the REST server rejects the client token, and the client has to authenticate again.
com.emc.documentum.rest.security.ticket.impl .TouchedTimeoutExpirationPolicy	The client token expires after a specified duration. The REST server issues new client tokens under certain conditions. For details, see the following: • If the REST client sends a request before the duration, the REST server accepts the client token, and issues another client token with the same duration to the client for subsequent requests. • If the REST client sends a request after the duration, the REST server rejects the client token, and the client has to authenticate again.

Explicit Logoff

Starting from release 7.2, all authentication schemes utilizing client tokens enable client applications to explicitly log off. When a client application obtains a client token, the link relation `http://identifiers.emc.com/linkrel/logoff` is generated in the Repository resource. The client application can log off by using this link relation, which notifies the REST server to invalidate the client token. After that, the client application must perform authentication again when submitting new requests.

Client Token Encryption and Decryption

Client tokens can be encrypted and decrypted using various cryptography algorithms. Follow the instructions in `rest-api-runtime.properties` to configure the cryptography algorithm–related parameters.

Note:

- The default cryptography algorithms used for the client token encryption and decryption are strong enough in most cases. Therefore, we recommend that you keep the original settings unless you have special business requirements.
- When you use a Crypto provider other than JsafeJCE (RSA provider) or BC (Bouncy Castle provider), you must set the `rest.security.crypto.provider.class` parameter correspondingly after modifying `rest.security.crypto.provider`.
- When you deploy Documentum REST Services on IBM WebSphere and use one of the following combinations of authentication schemes:
  - — HTTP Basic with Client Token
  - — CAS with Client Token
  - — Kerberos with Client Token

  The following configurations are required in `rest-api-runtime.properties`:
- rest.security.crypto.provider=IBMJCE
- rest.security.crypto.provider.class=com.ibm.crypto.provider.IBMJCE
- rest.security.random.algorithm=IBMSecureRandom

For a multi-node deployment of REST servers, you must set the `rest.security.crypto.key.salt` parameter consistently across all REST servers.

- Some web applications may contain security providers that share the same names with security providers in Documentum REST Services. The `rest.security.crypto.provider.force.replace` property determines whether to replace a security provider in web applications with the one in Documentum REST Services if the two providers share the same name. The default value is `false`, meaning that Documentum REST Services uses the security provider registered in the web application. The default setting is recommended.

Algorithms with a Key Size Lager than 128 bits

The default version of Java Cryptography Extension (JCE) policy files bundled in the JDK(TM) environment limits the key size of cryptography algorithms to 128 bits. To remove this restriction, download Unlimited Strength Jurisdiction Policy Files from the Oracle web site.

Bypassing Prompted Dialog Boxes from Browsers upon HTTP 401 Unauthorized

When a user provides invalid credentials, an HTTP 401Unauthorized status code is returned, which contains a WWW-Authenticate header indicating the authentication scheme.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic My Realm
```

This behavior may trigger web browsers to prompt a login dialog box for re-authentication. However, when you design your own login screen, it could be blocked by the one the web browser prompts. To bypass login dialog box web browsers prompt, append a suffix to the authentication scheme in the `WWW-Authenticate` header. This prevents web browsers from recognizing the scheme therefore no login dialog boxes are prompted.

To enable this feature, client applications must set the custom header DOCUMENTUM-CUSTOM -UNAUTH-SCHEME to `true` in requests. On the REST server side, set the value of the `rest.api.unauthorized.response.scheme.suffix` property to the suffix to append in `rest-api-runtime.properties`.

Example 5-20. Request

```
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ
DOCUMENTUM-CUSTOM-UNAUTH-SCHEME: true
```

Example 5-21. Runtime Property Setting

```
rest.api.unauthorized.response.scheme.suffix=MyScheme
```

Example 5-22. Response

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic_MyScheme My Realm
```

Chapter 6

Explore Documentum REST Services

This section provides a sample which guides you through some of the basic concepts and tasks in Documentum REST Services. The sample focuses on common tasks involving folders, documents, and contents. By exploring the sample, you will be familiar with the following tasks:

- Navigating to a repository from the service node
- Navigating to a cabinet in the repository
- Using the pagination feature in a folder
- Creating a document under a folder
- Adding a content to a document
- Deleting a document

Note:

- This sample uses the JSON representation.
- Documentum REST Services is assumed to be deployed on `localhost:8080`.

Prerequisites

- The web browser must be able to render the JSON representation automatically.
- The web browser must have a REST client plug-in installed.

Common Tasks on Folders, Documents, and Contents

Follow these steps after you deploy Documentum REST Services:

1. Type the following URL in your web browser to navigate to the service node (Home Document).

   http://localhost:8080/dctm-rest/services.json

   The service node contains a list of the available services.

```
HTTP/1.1 200 OK
   Content-Type: application/json;charset=UTF-8

   {
     "resources": {
       "http://identifiers.emc.com/linkrel/repositories": {
         "href": "http://localhost:8080/dctm-rest/repositories.json",
         "hints": {
           "allow": [
             "GET"
           ],
           "representations": [
             "application/xml",
             "application/json",
             "application/atom+xml",
             "application/vnd.emc.documentum+json"
           ]
     }
   },
       "about": {
         "href": "http://localhost:8080/dctm-rest/product-info.json",
         "hints": {
           "allow": [
             "GET"
           ],
           "representations": [
             "application/xml",
             "application/json",
             "application/vnd.emc.documentum+xml",
             "application/vnd.emc.documentum+json"
           ]
     }
   }
     }
   }
```

   Typically, you will see the `resources` service as the first node of services. In the link relation `http://identifiers.emc.com/linkrel/repositories`, note the URI to the `repositories` resource that resembles the following:

   `http://identifiers.emc.com/linkrel/repositories`

2. Click the repositories link you got from step 1 to navigate to the list of all available repositories. Explore the output, and note the information in the `entries` element.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "id": "http://localhost:8080/dctm-rest/repositories",
  "title": "Repositories",
  "updated": "2013-05-22T14:41:29.672+08:00",
  "author": [
    {
      "name": "EMC Documentum"
}
  ],
  "total": 2,
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/dctm-rest/repositories.json"
}
  ],
  "entries": [
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme01",
      "title": "acme01",
      "content": {
        "content-type": "application/json",
        "src": "http://localhost:8080/dctm-rest/repositories/acme01.json"
  },
      "links": [
        {
          "rel": "edit",
          "href": "http://localhost:8080/dctm-rest/repositories/acme01.json"
    }
      ]
},
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme02",
      "title": "acme02",
      "content": {
        "content-type": "application/json",
        "src": "http://localhost:8080/dctm-rest/repositories/acme02.json"
  },
      "links": [
        {
          "rel": "edit",
          "href": "http://localhost:8080/dctm-rest/repositories/acme02.json"
    }
      ]
}
  ]
}
```

3. Click the `href` link of the `edit` link relation of a repository in the `entries` element to retrieve the details of the repository. Enter your credentials if you are prompted for authentication.

   ```
   GET http://localhost:8080/dctm-rest/repositories/acme01.json
   Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ
   ```

   You will get an output that resembles the following:

   ```
   HTTP/1.1 200 OK
   Content-Type: application/json;charset=UTF-8

   {
     "id": 1234,
     "name": "acme01",
     "servers": [
       {
         "name": "acme01",
         "host": "CS70_Main",
   ```

```
        "version": "7.2.0000.0000  Win64.SQLServer",
        "docbroker": "CS70_Main"
}
  ],
  "links": [
    {
      "rel": "self",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/users",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/users.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/current-user",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/currentuser.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/groups",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/groups.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/cabinets",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/cabinets.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/formats",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/formats.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/network-locations",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/
      network-locations.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/relations",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/relations.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/relation-types",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/relation-types.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/checked-out-objects",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/
      checked-out-objects.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/types",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/types.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/dql",
      "hreftemplate":
      "http://localhost:8080/dctm-rest/repositories/acme01.json
      {?dql,page,items-per-page}"}
  ]
}
```

By clicking the link relations in the `links` element, you can drill down various resources in the repository.

4. By clicking the link relation `http://identifiers.emc.com/linkrel/cabinets` in the `links` element, you can navigate to the list of all available cabinets. Explore the output, and note the information in the `entries` element.

   Note: You can set the `inline` parameter to `true` to retrieve the entire content of each entry in the collection. In the following output, the `inline` parameter uses the default value `false` so that `content` of an entry only contains `content-type` and `src`.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "id": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets",
  "title": "Cabinets",
  "updated": "2013-05-22T14:55:24.594+08:00",
  "author": [
    {
      "name": "EMC Documentum"}
  ],
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets.json"}
  ],
  "entries": [
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0c0004d280000d1f",
      "title": "dmadmin",
      "updated": "2012-10-15T23:31:27.000+08:00",
      "author": [
        {
          "name": "dmadmin",
          "uri":
          "http://localhost:8080/dctm-rest/repositories/acme01/users/dmadmin"}
      ],
      "content": {
        "content-type": "application/json",
        "src":
        "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
        0c0004d280000d1f.json"},
      "links": [
        {
          "rel": "edit",
          "href":
          "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
          0c0004d280000d1f.json"}
      ]
},
    {
      "id":
      "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0c0004d280000107",
      "title": "Temp",
      "updated": "2012-10-15T15:27:31.000+08:00",
      "author": [
        {
          "name": "acme01",
          "uri":
          "http://localhost:8080/dctm-rest/repositories/acme01/users/acme01"}
      ],
      "content": {
        "content-type": "application/json",
        "src":
        "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
        0c0004d280000107.json"},
      "links": [
        {
          "rel": "edit",
          "href":
          "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
          0c0004d280000107.json"}
      ]
}
  ]
}
```

5. Click the `src` link in the `content` element of a cabinet in the `entries` element to retrieve the details of the cabinet. You will get an output that resembles the following:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "name": "object",
  "type": "dm_cabinet",
  "definition":
  "http://localhost:8080/dctm-rest/repositories/acme01/types/dm_cabinet",
  "properties": {
    "r_object_id": "0c0004d280000107",
    "object_name": "Temp",
    "r_object_type": "dm_cabinet",
    "title": "Temporary Object Cabinet",
    ...,  },
  "links": [
    {
      "rel": "self",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0c0004d280000107.json"
},
    {
      "rel": "edit",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0c0004d280000107.json"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/delete",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0c0004d280000107.json"
},
    {
      "rel": "canonical",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
      0c0004d280000107.json"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/folders",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/folders.json"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/documents",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/documents.json"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/objects",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/objects.json"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/child-links",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/child-links.json"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/relations",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/relations.json?
      related-object-id=0c0004d280000107&related-object-role=any"
}
  ]
}
```

By clicking the link relations in the `links` element, you can drill down various resources in the cabinet.

6. Click the `href` link of link relation `http://identifiers.emc.com/linkrel/folders` of the cabinet to retrieve the details of the child folders under the cabinet. You will get an output that resembles the following:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "id": "http://localhost:8080/dctm-rest/repositories/acme01/
  folders/0c0004d280000107/folders",
  "title": "Folders under folder 0c0004d280000107",
  "updated": "2013-05-22T15:07:54.156+08:00",
  "author": [    {      "name": "EMC Documentum"}  ],
  "page": 1,
  "items-per-page": 100,
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/folders.json" },
    { "rel": "next",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/folders.json?items-per-page=100&page=2"},
    { "rel": "first",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107/folders.json?items-per-page=100&page=1"}
  ],
  "entries": [
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0b0004d280009646",
      "title": "REST-API-TEST-FOLDER43fd1f60-fc9a-4402-9aca-30d86ab9f4b4",
      "updated": "2013-05-22T15:07:39.000+08:00",
      "author":
      [{"name": "dave",  "uri": "http://localhost:8080/dctm-rest/repositories/
      acme01/users/dave"}],
      "content": {
        "content-type": "application/json",
        "src": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
        0b0004d280009646.json"
  },
      "links": [{"rel": "edit","href": "http://localhost:8080/dctm-rest/
      repositories/acme01/folders/0b0004d280009646.json"    }]
},
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0b0004d280009647",
      "title": "REST-API-TEST-FOLDERa0cb9000-2b85-47ea-a427-9108a43c5097",
      "updated": "2013-05-22T15:07:39.000+08:00",
      "author": [{ "name": "dave",  "uri": "http://localhost:8080/dctm-rest/
      repositories/acme01/users/dave"  } ],
      "content": {
        "content-type": "application/json",
        "src": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
        0b0004d280009647.json"
  },
      "links": [{ "rel": "edit", "href": "http://localhost:8080/dctm-rest/
      repositories/acme01/folders/0b0004d280009647.json"} ]
},
    ...
}
```

Click the `href` link in the `next` link relation to navigate to the next page.

7. Click the `href` link of the `edit` link relation of a folder in the `entries` element to retrieve the details of the folder.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "name": "folder",
  "type": "dm_folder",
  "definition": "http://localhost:8080/dctm-rest/repositories/acme01/types/dm_folder",
  "properties": {
    "r_object_id": "0b0004d280009646",
    "object_name": "REST-API-TEST-FOLDER43fd1f60-fc9a-4402-9aca-30d86ab9f4b4",
    "r_object_type": "dm_folder",
    ...
  },
  "links": [
    {
      "rel": "self",
      "href":
      "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646.json"},
    {
      "rel": "edit",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/delete",
      "href":  "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/parent-links",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      0b0004d280009646/parent-links.json"},
    {
      "rel": "parent",
      "title": "0c0004d280000107",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0c0004d280000107.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/folders",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646/folders.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/documents",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646/documents.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/objects",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646/objects.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/child-links",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646/child-links.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/cabinet",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
      0c0004d280000107.json"},
    {
      "rel": "http://identifiers.emc.com/linkrel/relations",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/
      relations.json?related-object-id=0b0004d280009646&related-object-role=any"}
  ]
}
```

In the output you received from step 7, click the `href` link in the `http://identifiers .emc.com/linkrel/documents` link relation to navigate to the list of available documents in this folder. You will notice that the structure of the Documents resource is similar to that of the Folders resource.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8

{
  "id": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
  0b0004d280009646/documents",
  "title": "Documents under folder 0b0004d280009646",
  "updated": "2013-05-22T15:18:41.844+08:00",
  "author": [
    {
      "name": "EMC Documentum"
}
  ],
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
      0b0004d280009646/documents.json"
}
  ],
  "entries": [
    {
      "id": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      090004d280009651",
      "title": "REST-API-TEST-DOC",
      "updated": "2013-05-22T15:07:39.000+08:00",
      "author": [
        {
          "name": "dave",
          "uri": "http://localhost:8080/dctm-rest/repositories/acme01/
          users/dave"
    }
      ],
      "content": {
        "content-type": "application/json",
        "src": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
        090004d280009651.json"
  },
      "links": [
        {
          "rel": "edit",
          "href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
          090004d280009651.json"
    }
      ]
}
  ]
}
```

8. Send a POST request to create a document under the folder with the following configuration:
   - Set the URL to the `href` link in the `documents` link relation you got in step 7.
   - Set the content type to `application/vnd.emc.documentum+json`.
   - Enter the following data in the request body:

     ```
     {
       "properties":{
         "object_name":"Vienna",
         "r_object_type":"dm_document"
       }
     }
     ```

   - Set the method to POST, and then send the request.

   Note: The detailed steps may vary depending on the tool you use to send the request.

You will receive an `HTTP 201 Created` status upon a successful document creation. Also, you will receive the URI pointing to the document in the `Location` header of the response. Enter this URI in the web browser to navigate to the newly-created document. The output resembles the following:

```
HTTP/1.1 201 OK
Content-Type: application/json;charset=UTF-8
Location: http://localhost:8080/dctm-rest/repositories/acme01/documents/
090004d280009894.json

{
  "name": "document",
  "type": "dm_document",
  "definition": "http://localhost:8080/dctm-rest/repositories/acme01/
  types/dm_document",
  "properties": {
    "r_object_id": "090004d280009894",
    "object_name": "Vienna",
    "r_object_type": "dm_document",
    ...
  },
  "links": [
    {      "rel": "self",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
```

```
        090004d280009894.json"},
      {      "rel": "edit",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
        090004d280009894.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/delete",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
        090004d280009894.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/parent-links",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
        090004d280009894/parent-links.json"},
      {      "rel": "parent",
        "title": "0b0004d280009646",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
        0b0004d280009646.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/cabinet",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
        0c0004d280000107.json"},
      {      "rel": "contents",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
        090004d280009894/contents.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/primary-content",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
        090004d280009894/contents/content.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/checkout",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
        090004d280009894/lock.json"},
      {      "rel": "version-history",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
        090004d280009894/versions.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/current-version",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
        090004d280009894/versions/current.json"},
      {      "rel": "http://identifiers.emc.com/linkrel/relations",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/
        relations.json?related-object-id=090004d280009894&related-object-role=any"}
  ]
}
```

Send a POST request to create content for this document with the following configuration:

- Set the URL to the `href` link in the `contents` link relation you got in step 8.
- If the plug-in allows you to set the request body from a local file, select the local Content Media that you want to import, and then set the corresponding content type.

  If the plug-in does not allow you to set the request body from a local file, input the content file binary to the request body, and then set the corresponding value for the content type.

  If the plug-in does not allow you to set the request body from a local file, input the content file binary to the request body, and then set the corresponding value for the content type. For example:

  ```
  POST  http://localhost:8080/dctm-rest/repositories/acme01/objects/
  090004d280009894/contents.json
  Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ
  Content-Type: text/plain

  a test string content: hello, rester!
  ```

- Set the method to POST, and then send the request.

Note: The detailed steps may vary depending on the tool you use to send the request.

You will receive an `HTTP 201 Created` status upon a successful content creation.

```
HTTP/1.1 201 OK
Content-Type: application/json;charset=UTF-8
Location: http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/contents/content.json?format=atd&modifier=&page=0

{
  "name": "content",
  "properties": {
    "r_object_id": "060004d280004a5f",
    "rendition": 0,
     …
  },
  "links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      090004d280009894/contents/content.json?format=atd&modifier=&page=0"
},
    {
      "rel": "http://identifiers.emc.com/linkrel/content-media",
      "title": "ACS",
      "href": "http://localhost:9080/ACS/servlet/ACS?command=read&version=2.3
      &docbaseid=0004d2&basepath=C%3A%5CDocumentum%5Cdata%5Cacme01%5Ccontent_
      storage_01%5C000004d2&filepath=80%5C00%5C26%5C0a.atd&objectid=090004d280
      009894&cacheid=dAAEAgA%3D%3DCiYAgA%3D%3D&format=atd&pagenum=0&signature=
      QQj3oFudCLohPno49lwoVFnPQihxQGNRiv0W7U%2BrMqzCD%2FngiDKM7sBKpsk4S6a%2B%2F
      nBPcR6cW1qmXW2SIuP%2FeIbtI4upEs3%2B4aMZVeac9njIJ6zRosgm8yBIYAgm038KhDVOLF
      1Bxrb8Wsx%2BvQMSZyZpcHmuMOpovpjZHtwCiJ8%3D&servername=CS70RC2_MAINACS1&mo
      de=1&timestamp=1369207791&length=38&mime_type=text%2Fplain&parallel_stream
      ing=true&expire_delta=360"
},
    {
      "rel": "parent",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
      090004d280009894.json"
}
  ]
}
```

Click the `href` link of the link relation `http://identifiers.emc.com/linkrel/content-media` to open the imported content.

Note: The ACS link may have been URL encoded.

```
HTTP/1.1 201 OK
Content-Type: text/plain
Date: Wed, 22 May 2013 07:38:52 GMT
ETag: W/"38-1369207790346"
Expires: 0
Last-Modified: Wed, 22 May 2013 07:29:50 GMT

a test string content: hello, rester!
```

Send a DELETE request to delete the document with the following configuration:

- Set the URL to the URI pointing to the document you got in step 8.
- Set the method to DELETE, and then send the request.

```
DELETE  http://localhost:8080/dctm-rest/repositories/acme01/objects/
   090004d280009894.json
   Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ
```

Note: The detailed steps may vary depending on the tool you use to send the request.

You will receive an `HTTP 204 No Content` status upon a successful deletion.

Chapter 7

Tutorial: Consume REST Services Programmatically

This tutorial provides you with information about how to navigate collection hierarchies, read documents, and add documents in Python, using the requests library for HTTP requests, and the JSON representation of resources.

Note: Documentum REST Services is programming language independent. You can use other languages to consume REST Services as well.

Basic Navigation

Documentum REST Services has a rich hierarchy as follows:

- Documentum REST Services starts with the service node as the root.
- The service node contains a set of repositories.
- Each repository contains a set of cabinets or folders.
- Either folders or cabinets can contain documents or sysobjects.
- A document contains metadata, and can contain a primary content and multiple renditions.

The Documentum REST Services allows clients to traverse these structures using navigation or queries. The following code enables you to navigate from the service entry point to a repository named `tagsalad`. From the `tagsalad` repository, you can access the cabinet San Francisco.

```
import requests
import json
homeResource = "http://example.com/documentum-rest/services"
drel="http://identifiers.emc.com/linkrel/"
repository = "tagsalad"
cabinet = "San Francisco"
credentials = ('tagsalad', 'password')
home = requests.get(homeResource)
home.raise_for_status()
repositories_uri = home.json()['resources'][drel+'repositories']['href']
repository = get_repository(repositories_uri, repository)
cabinets_uri = get_link(repository, drel+'cabinets')
sanfran = get_atom_entries(cabinets_uri, 'title="San Francisco"')[0]
documents = get_link(sanfran, drel+'documents')
```

In this code sample, The function `get_link` returns a link based on a link relation. The function `get_repository` gets a repository with a given name. The function `get_atom_entries` returns Atom entries from a collection based on their properties. The rest of this section explains the code above in more details, shows the JSON representation of resources, and discusses some REST design principles.

Service Entry Point

Documentum REST Services is a hypermedia-driven API, with a single entry point, which is associated with the Home Document resource. You can use the requests library to read the Home Document resource:

```
home = requests.get(homeResource)
```

By running the `home.json()` method, you get the JSON representation of the Home Document resource:

```
{
 "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/repositories"
   },
   "about": {
     "hints": {
       "allow": [
         "GET"
       ],
     "representations": [
       "application/xml",
       "application/json",
       "application/vnd.emc.documentum+xml",
       "application/vnd.emc.documentum+json"
     ]
   },
     "href": "http://localhost:8080/dctm-rest/product-info"
  }
 }
}
```

Link Relations

In a hypermedia-driven Documentum REST Services, 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. Instead, it is a name encoded as a URI, which identifies the purpose of a link. A link relation is associated with the `href` entry that contains the physical address of the link. For example, in the JSON representation of the Home Document resource shown above, the `http://identifiers.emc.com/linkrel/repositories` link relation identifies a URI for an Atom feed containing repository entries, and the `href` entry indicates the physical address of 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 resource, the following expression returns the physical address of the repositories feed:

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

Alternatively, you can use the following code to get the physical address of the repositories feed.

```
#Retrieving a link based on a link relation
def get_link(e, linkrel, default=None):
    return [ l['href'] for l in e['links'] if l['rel'] == linkrel ][0]
repositories_uri = get_link(home, 'http://identifiers.emc.com/linkrel/ repositories')
```

Code explanation:

The `get_link()` function returns the link (physical address) 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, and then returns the first entry (there will never be more than one entry matching a given link relation).

Feeds and Entries

You may have followed the instructions in the Chapter Chapter 6, Explore Documentum REST Services and have familiarized yourself with the JSON representation of the Repositories feed. In that sample, each entry represents one repository and only contains a small subset of the content of a repository resource. If you need to get a specified repository from the feed and return its entire content programmatically, refer to the following code as an example:

```
#Retrieving the entire content of a repository specified by title
def get_repository(repositories_uri, 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):
    if filter:
        params = {'inline': 'true', 'filter': filter }
    else:
        params = {'inline': 'true' }
    response = requests.get(feed_uri, params=params, auth=credentials)
    response.raise_for_status()
    return [ e['content'] for e in response.json()['entries'] ]
repository = get_repository(repositories_uri, repository)
```

Code explanation:

The `get_repository` function calls `get_atom_entries` to return a list of the repositories contained in the feed, and 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. This function sets the `inline` parameter to `true` so that the resulting entries contain the entire resource, instead of a subset of the content.

The return expression contains the following list comprehension:

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

For collections within a given repository, you can specify conditions that are used to select results by using Filter Expression, page 41. For example, a repository contains cabinets, so we can find the San Francisco cabinet using the following code:

```
cabinet = get_atom_entries(cabinets_uri, 'title="San Francisco"')[0]
```

From the Home Document Resource to Documents

By now, you should be able to understand the code at the beginning of the Basic Navigation, page 135 section.

Three kinds of resources are used. The first kind is the Home Document resource. Here is the code that retrieves this resource and finds the repositories URI in it:

```
homeResource = "http://example.com/documentum-rest/services"
drel="http://identifiers.emc.com/linkrel/"
home = requests.get(homeResource)
home.raise_for_status()
repositories_uri = home.json()['resources'][drel+'repositories']['href']
```

The second kind is an EDAA feed (the JSON representation of an Atom Feed). The following code retrieves a repository from the JSON representation using the title of the corresponding entry, and then retrieves the URI of the Cabinets resource:

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

The third kind is the single resources (in our tutorial, the Repository resource and the Cabinet resources). The following code retrieves a cabinet from the cabinets feed, and then retrieves the URI of the documents feed from it:

```
sanfran = get_atom_entries(cabinets_uri, 'title="San Francisco"')[0]
documents = get_link(sanfran, drel+'documents')
```

Read Entries

After you get a collection of documents, you can read each entry and print its properties.

If all returned documents can fit on one page (by default, 100 entries), and you only print properties that are present in the feed when `inline` is set to `false`, refer to the following code:

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

Note: The code sample above prints the title of the entry instead of the document title under the properties element, as the properties element is not returned when `inline` is set to `false`.

The following code sample prints two properties that are returned only when `inline` is set to `true` (`object_name` and `title`) for each document in the collection. Additionally, the sample supports pagination by using the next link relation so that the results can span multiple pages when the number of results is large.

```
documents = get_link(sanfran, drel+'documents')
while True:
    response = requests.get(documents, params='inline=true', auth=credentials)
    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 similar to our previous sample. After printing the items on the given page, the `get_link` function looks for the `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.

Filter, Sort, and Pagination

When searching for the San Francisco cabinet, we used a simple filter expression. Here is an example of a slightly more complex filter:

```
contains(object_name, "COFFEE") and r_modify_date >= date("2012-12-03")
```

With this expression, the request returns resources whose object_name contains COFFEE. Additionally, resources that were modified earlier than 2012-12-03 are filtered out. For more examples, see Filter Expression Examples, page 49.

A filter always returns an entire resource. However, you can use Property View, page 49 to specify a set of properties that should be returned. The sort order can also be specified, as can the number of items on a page. For more information, see Common Definition - Query Parameters, page 19.

The following code sample 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_modify_date >= date("2012-12-03")',
    'items-per-page' : 50
}
response = requests.get(documents, params=params, auth=credentials)
response.raise_for_status()
for e in response.json()['entries']:
    print (prettyprint(e))
```

Create Entries

You can create entries in a feed by using POST and setting the corresponding content type.

At least the object name and the object type must be specified. In a real application, you may need to create an object type that allows us to represent the properties of a given kind of document. To keep it simple, the following sample only sets the `title` property.

```
body = json.dumps(
 {
   "properties" : {
    "object_name" : "Earthquake McGoon's",
    "r_object_type" : "dm_document",
    "title" : "50 California Street, 94111"
   }
  }
)
headers = { 'content-type' : 'application/vnd.emc.documentum+json' }
response = requests.post( documents, data=body, headers=headers, auth=credentials)
```

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

```
>>> response = requests.post( documents, data=body, headers=headers, auth=credentials)
>>> response.status_code
201
>>> response.raise_for_status()
>>> response.reason
'Created'
>>> response.json()
 {
   "properties" : {
    "object_name" : "Earthquake McGoon's",
    "r_object_type" : "dm_document",
    "title" : "50 California Street, 94111"
   }
  }
```

Update Entries

You can update a resource by using POST with the URI of the resource and setting the corresponding content type.

Suppose `response.json` contains a document. The following code updates the `object_name` property in the document.

```
document = response.json()
document['properties']['object_name'] = 'Kilroy was Here!'
headers = { 'content-type' : 'application/vnd.emc.documentum+json' }
uri = get_link(document, 'edit')
response = requests.post(uri, json.dumps(document), headers=headers, auth=credentials)
```

Note that the above code uses the `edit` link relation, which is 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 the response contains the updated document resource.

```
>>> response.status_code
200
>>> response.reason
'OK'
>>> response.raise_for_status()
>>> responsed .json()
 {
   "properties" : {
    "object_name" : "Earthquake McGoon's",
    "r_object_type" : "dm_document",
    "title" : "Kilroy was Here!"
   }
  }
```

Delete Entries

You can delete a resource by using DELETE with the URI of the resource.

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

The response contains the HTTP status code 204 with no content.

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

Chapter 8

Resource Extensibility

Overview

Resource extensibility is an API infrastructure that enables you to extend Documentum REST Services by composing, customizing, and creating new REST resources. These newly-created resources are finally discoverable from the Home Document, existing core REST resources, or new custom resources via HATEOS relations. By wielding the power of resource extensibility, you can tailor Documentum REST Services to your needs with limited amount of coding.

The following diagram illustrates the lifecycle of custom resource development.

Figure 1. Lifecycle of Building Custom Resources

1. Design and implement custom resources.

   The main objective of the REST extensibility is to add custom resources to build a rich-functioned web application, except for the out-of-box Core resources. Documentum REST Services provides you with a set of toolkits and libraries to facilitate the programming of custom resources, which are:

   - A set of Core REST Java libraries for custom resource development which optimizes the REST services development upon Core REST services. Java docs and code samples for the shared library are also provided.
   - A marshalling framework which frees you from writing codes to handle the XML and/or JSON message marshalling and unmarshalling in a coherent way with Core REST resources.
   - A Maven-based toolkit to manage the build process of custom resource projects. An archetype is included to create a sample project for your organization.
   - An Ant-based toolkit to manage the build process of custom resource projects.
     A lot of code samples for the custom resource development.

2. Customize Core REST resources.

   Typically, the newly developed resources need to interact with Core resources via link relations or other representations. Documentum REST Services provides you with a number of features to customize Core resources.

   - Add topmost custom resources to the Home document. Topmost resources are those resources that do not link from other resources.
   - Add new link relations to Core resources. The newly developed resource can be a forwarded state of a Core resource, so that the Core resource needs to give a link relation to the new resource.

3. Repackage the REST WAR file.Documentum REST Services provides the build toolkits and scripts to build custom resources and Core resources into a single WAR file.
4. Deploy the WAR file to an application server.
   The custom REST services can be deployed into the same type of application servers supported by Documentum REST Services, as long as the custom REST services does not bring any library conflict to the application servers.
5. Consume the extended Documentum REST Services.
   When the custom resources are developed in the same pattern as Core resources, the client consumes the Core resources and custom resources in the same pattern, too. And both of them share the same authentication scheme.

Get Started With the Development Kit

Documentum REST Services SDK provides Maven and Ant development kits to build up the custom REST project for users. Please follow SDK instructions to set up the first custom REST project.

Maven-Based Toolkit

Since the 7.2 release, a Maven toolkit is available in the SDK of Documentum REST Services. The toolkit makes the build process of custom resource projects much easier. Furthermore, the kit introduces a set of deliverables to improve the productivity of custom resource development.

Note: Maven is the recommended build tool in custom resource development. However, it is not required. You can use other tools to build your custom resource projects.

The Maven kit introduces the following deliverables:

- A core REST Java library and dependencies for REST extensibility development
- Maven pom files that describe the dependencies of the Core REST Java library

  The pom files describe the internal and external library dependencies of Core REST JAR files. You can install these Core REST JAR files into your local Maven repository as third-party JARs.
- A Maven archetype project for custom resource development

  The archetype project can be used as a template project for custom resource development. This project can be installed in both local and remote repositories. To get started with the archetype project, run the following command to create a `HelloWorld` project:

  ```
  mvn archetype:generate -DarchetypeGroupId=com.emc.documentum.rest.project
  -DarchetypeArtifactId=documentum-rest-project-archetype -DgroupId=your_group_name
  -DartifactId=your_artifact_name -DarchetypeCatalog=<local or remote>
  ```

  The `HelloWorld` project is a structured and runnable Java project containing a sample of custom resource development. You can later create your custom resource project based on this archetype project.
- A guide to install the archetype project in a local Maven repository
- A guide to install the archetype project in a remote Maven repository
- A guide to install the core REST Java library in a local Maven repository
- A guide to install the core REST Java library in a remote Maven repository
- A guide to build a REST extensibility project

Creating a Custom Resource Project from the Maven Archetype

Instead of requiring you to create a custom resource project from scratch, Documentum REST Services SDK provides you with a Maven archetype to reduce your efforts in project building and coding. You can generate a new project from the archetype and start custom resource development from there.

The Maven archetype helps you create a multi-module project that looks like the following:

Figure 2. Maven Archetype Project Structure

System Requirements:

- Java 6 or Java 7 must be installed and the location of which must be added to the `classpath` system variable.
- Maven 3 must be installed and the location of which must be added to the `classpath` system variable.

Creating a project from the Maven archetype consists of three stages:

1. Installing the Maven archetype to your local repository
2. Installing the Maven archetype dependencies
3. Creating a project based on the archetype

Installing the Maven Archetype

1. Extract the documentum-rest-sdk-*version-number*.zip (tar) package to your local drive.
2. Navigate to the `documentum-rest-sdk-`*`version-number`*`/maven-kit/archetype` directory and run the following Maven task to install the archetype:

   ```
   mvn install
   ```

After the task completes, the core REST archetype is installed to your local Maven repository. By default, the archetype is under the following directory:

```
user_profile/.m2/repository/com/emc/documentum/rest/extension
```

Installing the Maven Archetype Dependencies

1. Install the DFC dependency with the following command:

   ```
   mvn install:install-file -Dfile=sdk-root/lib/dependencies
   /dctm/dfc-version-number.jar
   -DgroupId=com.emc.documentum.dfc -DartifactId=dfc -Dversion
   =version-number -Dpackaging=jar
   ```

2. Install Documentum REST Services dependencies. All dependencies (JAR files) in the lib/core directory of the SDK must be installed to your local Maven repository by using a command that resembles the following.

   ```
   mvn install:install-file -Dfile=sdk-root/lib/core/JAR_package_filename
                             -DpomFile=sdk-root/lib/core/POM_filename
   ```

   For example:

   ```
   mvn install:install-file -Dfile=sdk-root/lib/core/documentum-rest
   -core-resource-version-number.jar
                            -DpomFile=sdk-root/lib/core/documentum
   -rest-core-resource-version-number.pom
   ```

   You can create a script to automate the installation of these dependencies.

3. The WAR dependency is used to package a custom project into a WAR file. Run the following command to install the WAR dependency:

   ```
   mvn install:install-file -Dfile=sdk-root/lib/war/documentum
   -rest-web-version-number.war
                             -DgroupId=com.emc.documentum.rest
                             -DartifactId=documentum-rest-web -Dversion=version-number
   ```

```
                          -Dpackaging=war
```

4. Documentum REST Services leverages an annotation validator plugin to scan packaged WAR files and report annotation issues of your custom project. Run the following command to install the validator plugin:

   ```
   mvn install:install-file -Dfile=sdk-root/tools/
                             documentum-rest-extension-validating-version-number.jar
                            -DpomFile=sdk-root/tools/pom.xml
   ```

Creating a Project from the Maven Archetype

After the Maven archetype for Documentum REST services is installed, a custom REST project can be created either in an IDE or a command-line prompt.

Creating a Project from the Maven Archetype in an IDE

Eclipse has the wonderful support for Maven archetype project build up. In this section, we use Eclipse as an example to demonstrate how to create a project from the Maven archetype in an IDE.

1. Click **File** -> **New** -> **Project**, select **Maven Project**, and then click **Next**. The New Maven Project wizard appears.
2. In the Select project name and location phase, leave **Create a simple project (skip archetype selection)** unchecked. Click **Next** to proceed to the Select an Archetype phase.
3. In Select an Archetype, click **Configure** -> **Add Local Catalog**. The Local Archetype Catalog box dialog appears.
4. In the **Catalog File** field, click **Browse** to navigate to your .m2 folder and then select the `archetype-catalog.xml` file. In the **Description** field, enter the description of the archetype, and then click **OK**.
5. Click **OK** to go back to Select an Archetype. In the **Catalog** field, select the catalog you specified in Step 4, and then select the **Include snapshot archetypes** check box. The documentum-rest-extension-archetype artifact appears. Select this artifact and click **Next**.
6. Enter the information about group ID, artifact ID, version, and package, and then click **Finish**. Eclipse starts to build a project from the Maven archetype.

Creating a Project from the Maven Archetype in a Common-Line Prompt

When the working IDE environment is not the Eclipse, you can alternatively set up the archetype project with command lines. Maven and Java are required to be set into the system path.

1. Create a new directory to hold source code of the project and then enter the directory. Example:

   ```
   mkdir tmp
   cd tmp
   ```

2. Run the following Maven task to create the project:

   ```
   mvn archetype:generate
   -DarchetypeCatalog=local
   -DarchetypeGroupId=com.emc.documentum.rest.extension
   -DarchetypeArtifactId=documentum-rest-extension-archetype
   -DinteractiveMode=false
   -DgroupId=new_project_group_id
   -DartifactId=new_project_artifact_id
   -Dpackage=new_project_package_name
   ```

In this command:

- *new_project_group_id* represents the group ID of the new project. Example: `com.acme`
- *new_project_artifact_id* represents the artifact ID of the new project. Example: `acme-rest`
- *new_project_package_name* represents the Java package prefix of the new project. Example: `com.acme`

After the task completes, a new project is created under current directory

Verifying the Project

Out of the box, two custom resources – alias-sets and alias-set are embedded in the Maven archetype. Follow these steps to build your project, deploy the WAR file to an application server, and then access the alias-sets resource to verify the project.

1. Enter the directory *`artifact_id`*`-web/src/main/resources` of your project and create the `dfc.properties` file according to your Content Server installation. For more information, see the DFC Configuration section of the *EMC Documentum Platform REST Services Release Notes*.
2. Navigate to the project folder (the *artifact_id* directory) and then run the following Maven task to build the project:

   ```
   mvn install
   ```

   The *`artifact_id`*`-web-1.0.war` file is created under the *`artifact_id`*`/`*`artifact_id`*`-web/target` directory.
3. Deploy the WAR file to an application server. For more information, see the Installation chapter of the *EMC Documentum Platform REST Services Release Notes*.
4. Access the alias-sets with a GET request to a URL that looks like the following:

   ```
   http://localhost:port/acme-rest/repositories/repositoryName/alias-sets
   ```

Upon a successful deployment, the operation returns a collection of alias sets in the repository.

```
<feed xmlns="http://www.w3.org/2005/Atom"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <id>http://localhost:8080/dctm-rest/repositories/dctm72/alias-sets </id>
 <title>Alias Sets</title>
 <author>
  <name>EMC Documentum</name>
 </author>
 <updated>2014-08-20T17:00:36.197+08:00</updated>
 <link rel="self"
        href="http://localhost:8080/dctm-rest/repositories/dctm72/alias-sets"/>
 <entry>
  <id>http://localhost:8080/dctm-rest/repositories/dctm72/alias-sets/
     6600000b80000105</id>
  <title>AdminAccess</title>
  <author>
   <name>Administrator</name>
   <uri>
   http://localhost:8080/dctm-rest/repositories/dctm72/users/Administrator
   </uri>
  </author>
  <updated>2014-08-20T17:00:36.197+08:00</updated>
  <published>2014-08-20T17:00:36.197+08:00</published>
  <content src="http://localhost:8080/dctm-rest/repositories/dctm72/alias-sets/
     6600000b80000105"/>
```

```
  <link rel="self" href="http://localhost:8080/dctm-rest/repositories/dctm72/
     alias-sets/6600000b80000105"/>
 </entry>
 <entry>
  <id>
  http://localhost:8080/dctm-rest/repositories/dctm72/alias-sets/6600000b8000050a
  </id>
  <title>auxiliary_alias_set</title>
  <author>
   <name>dctm72</name>
   <uri>
   http://localhost:8080/dctm-rest/repositories/dctm72/users/dctm72
   </uri>
  </author>
  <updated>2014-08-20T17:00:36.197+08:00</updated>
  <published>2014-08-20T17:00:36.197+08:00</published>
  <content src="http://localhost:8080/dctm-rest/repositories/dctm72/alias-sets/
    6600000b8000050a"/>
  <link rel="self" href="http://localhost:8080/dctm-rest/repositories/dctm72/
    alias-sets/6600000b8000050a"/>
 </entry>
 <entry>...</entry>
 ...
 <entry>...</entry>
</feed>
```

Ant-Based Toolkit

The Documentum REST SDK contains an Ant-based toolkit that helps Apache Ant users manage the build process of custom resource projects. A folder named `ant-kit` is located under the root directory of the SDK, which contains data needed for the Ant build creation.

Creating a Custom Resource Build with Ant

1. Navigate to the `ant-kit/war/web-info/classes` folder and then edit the `dfc.properties` file according to your Content Server configuration.

   For more information about how to configure `dfc.properties`, see the DFC Configuration section of the *EMC Documentum Platform REST Services Release Notes*.

2. Navigate to the ant-kit folder where the `build.xml` file is located and then run the `ant all` command to create the `WAR` file.

   Enter the application name and version for your web archive when you are prompted to.

   When the command completes, a multi-module project is created together with a WAR file under the `ant-kit/dist` directory.

Figure 3. Ant Project Structure

Out of the box, a custom resource `alias-sets` is embedded in the Ant-based toolkit. After you deploy the WAR file, access `alias-sets` with a GET request to a URL that looks similar to:

```
http://host:port/acme-rest/repositories/MyRepository/alias-sets
```

If both the build creation and WAR file deployment are successful, the operation returns a collection of alias sets in the repository.

If you want to add more custom models, controllers, and resources, follow the instructions in `ant-kit/readme.txt`.

Architecture of Documentum REST Extensibility

The extensibility feature of Documentum REST Services allows you to create custom resources in the same RESTful pattern as Core resources, with the objective to reuse the Core REST infrastructure as much as possible. To understand where the extension points are within the whole infrastructure, we need to have an overview of the architecture of Documentum REST Services. The following diagram illustrates the overall architecture of Documentum REST Services.

Figure 4. Documentum REST Services Architecture

The whole Documentum REST Services is built as a single WAR application, including both Core resources and custom resources.

Documentum REST Services leverages Spring Security to provide various authentication schemes. Any authentication scheme configured in a deployment applies to both Core resources and custom resources. In the current release, the security component is not exposed for customization.

Documentum REST Services leverages Spring Web MVC Framework to build all REST resources. The Spring Web MVC sets clear separations of roles and makes implementations of components pluggable. By taking advantages of Spring Web MVC, Documentum REST Services provides various means of extensibility features, enabling you to add custom resources or customize Core resources with flexibility and efficiency. We call this as Documentum REST MVC.

On the top of the Documentum REST MVC, Documentum REST Services provides a unique REST annotation framework which helps to marshal REST Java resource models into JSON and/or XML representations, and unmarshal the JSON and/or XML representations into resource models. When developing custom resources, you just need to focus on the resource model design. The marshalling and unmarshalling are done by Documentum REST Services at the framework level.

Under the bottom of the Documentum REST MVC, persistence APIs communicate with Content Server repositories. Documentum REST Services provides a lot of common APIs to manipulate persistent data in the Content Server repositories. Besides, Documentum REST Services provides the hands-on session APIs that integrate to the Security component for different authentication schemes and exposes them as the uniform interfaces in the persistence component.

Documentum REST Security

In the current release, Documentum REST Services does not expose extension points for you to develop a custom authentication scheme. Commonly, custom resources use the same authentication scheme as Core resources within the same WAR deployment. More precisely, Documentum REST Services uses a set of Spring security filters to apply specific authentication schemes, and each security filter determines which resource URI pattern(s) are under the umbrella of the specified authentication scheme. Documentum REST Services applies all kinds of authentication schemes to the resource URI pattern `/repositories/{repositoryName}`and all its subsequent path segments. This means when a custom resource's URI is designed with the prefix `/repositories/{repositoryName}/`

(e.g., `/repositories/{repositoryName}/alias-sets`), the resource access is authenticated by the configured authentication scheme(s). When not (e.g., `/help`), the resource access is anonymous.

The security filters perform the real authentication to Content Server repositories or third party Single-sign-On parties. Once the authentication is successful, the persistence layer in the resource implementation can get the authenticated DFC session manager by using the `com.emc.documentum.rest.dfc.RepositorySession` and `com.emc.documentum.rest.dfc.RepositorySessionManager` APIs. A custom persistence API can extend `com.emc.documentum.rest.dfc.SessionAwareAbstractManager` to get the repository session. For more information, see Persistence: Session Management.

Documentum REST MVC

Documentum REST MVC performs customizations on Spring Web MVC to facilitate REST resource development. A typical resource implementation contains three layers, the model, the controller, and the view.

- The model layer is the data holder for a resource where Documentum REST annotations are defined for marshalling and unmarshalling.
- The controller layer is the center of the resource implementation where it defines the request and response mappings, as well as calls the persistence to manipulate the data. The controller implementation mainly uses Spring annotation `@Controller` to implement the resource. The controller method accepts and returns the model instances.
- The view layer is a wrapper of the model where representation-related information, such as links and atom attributes, are resolved. The output of the view is still the model instance since the model is the only entity for marshalling and unmarshalling. The view implementation binds to specific models and controllers.

The following diagram illustrates the relationship of the three layers:

Figure 5. Documentum REST MVC

A resource controller can be bound to one or several view definitions. Each view definition renders a specific model with regard to links and other customizations. The input of output of a controller method is the model class which is annotated with Documentum REST annotations. Developing Custom Resources discusses details of how Documentum REST MVC is used in custom resource development.

Documentum REST Marshalling Framework

Documentum REST Services provides a set of Java annotations to design resource model classes for the custom resources. The annotations support both XML and JSON marshalling and unmarshalling. By default, both Core resources and custom resources use the annotation marshalling framework to convert XML and JSON messages. But since Documentum REST Services supports Spring message conversions, this can actually be altered by users who have the deep knowledge of Spring message conversions. The section Core REST Marshalling Framework provides more detail.

Documentum REST Persistence

Documentum REST Services provides a number of APIs to manipulate persistent data in Content Server repositories, such as folder, document, content, etc. Persistence APIs are managed as Spring beans, and loaded by resource implementations with Spring annotation `@Autowired`. These APIs are mainly in the Java package "com.emc.documentum.rest.dfc" and its sub packages. Custom resources may also need to write new persistence APIs or integrate the existing type-based objects (TBOs) or service-based objects (SBOs) into the REST implementation. The Custom persistence APIs must be written in the same pattern as other Core persistence APIs and be loaded with Spring bean configurations.

Typically, you can develop a new persistence API with following procedure:

1. Write a Java interface.

```
public interface UserManager {
    UserObject createUser(String name, String password);
}
```

2. Write a Java class to implement this interface.

```
public class UserManagerImpl extends SessionAwareAbstractManager
        implements UserManager {
public UserObject createUser(String name, String password) {
     …
    }
}
```

3. Put the implementation as a Java bean in class-path file `/META-INF/spring/custom.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans ..>
    <bean id="customUserManager" class="com.acme.UserManagerImpl" />
</beans>
```

4. Reference the persistence in resources.

```
public CustomUserController extends AbstractController {
@Autowired UserManager customUserManager;
…
```

See persistence programming for the details of implementing a persistence API.

Documentum REST Marshalling Framework

Overview

Documentum REST Services provides a simple yet powerful marshalling framework to facilitate your custom resource development. This marshalling framework introduces a set of Java annotations, which binds models, to the XML and JSON representations directly so that you do not need to write, or message converters for custom data models. Primary features of the REST framework are summarized as follows:

- Out of the box message marshallers

  The marshalling framework handles Java objects to HTTP messages marshalling and HTTP messages to Java object unmarshalling under Documentum REST MVC.

- Out of the box REST data models

  A set of annotated Java classes for the common REST data models are packaged within the Core REST library and can be reused or even extended.

- Supports both XML and JSON

  The default XML and JSON message converters are implemented within the Core REST library. A common set of Java annotations are used for both XML and JSON representations. Therefore, once a Java class is annotated, both XML and JSON representation are supported.

Typically, the resource controller returns the annotated class instance directly, leaving all the message conversion work to Documentum REST MVC and the marshalling framework.

```
@Controller
public class MyDaoController extends AbstractController {
   @RequestMapping(value = {"/my-dao/{id}"}, method = RequestMethod.GET})
   @ResponseBody
   public MyDao get(@PathVariable("id") String id, @RequestUri final UriInfo uriInfo) {
      MyDao myDao = dfcObjectManager.get(id);
      return myDao;
   }
   ...
}
```

Annotations

This marshalling framework introduces the following annotations:

- @SerializableType

  Used on a Java class to serialize an object of the class to a REST structural representation.

- @SerializableField

  Used on a Java class field to serialize the Java object field to a sub element/property of a REST representation.

Both annotations work for both REST message marshalling and unmarshalling.

@SerializableType

The annotation `@SerializableType` under Java package `com.emc.documentum.rest.binding` indicates that a Java object is intended to be serialized to a REST representation. This annotation provides a number of attributes enabling you to customize the marshalling/unmarshalling behavior.

Table 7. SerializableType Attributes

Attributes	Description	Supported Formats	Marshal / Unmarshal	Default
value	Specifies the serializable name	JSON/XML	in and out	not set
fieldVisibility	Specifies whether to serialize all or partial fields according to the access modifiers at the class level: • ALL - all fields are intended to be serialized • PUBLIC - public fields are intended to be serialized • NONE - no fields are intended to be serialized Settings of the annotation `@SerializableField` take precedence over this attribute. For example, when `fieldVisibility` is set to `none`, properties with the annotation `@SerializableField` are still intended to be serialized. See Example A and Example B for detail.	JSON/XML	in and out	ALL
fieldOrder	Specifies the order of fields in the REST representation for a serialized object. For fields being serializable but not appearing in this list, they are presented at the end of the REST representation. For fields appearing in this list but not being serialized, they are ignored. See Example C for detail.	JSON/XML	out	not set

Attributes	**Description**	**Supported Formats**	**Marshal / Unmarshal**	**Default**
ignoreNullFields	Specifies whether to ignore fields with null values when marshalling. See Example D for detail.	JSON/XML	out	true
inlineField	Indicates that a non-primitive type field is intended to be moved to an upper level in the REST representation. Note that when you set `inlineField` to a certain field, all other field in the class are not serialized. See Example E for detail.	JSON/XML	out	not set
xmlValueField	Specifies a field from which the value will be taken as the XML element value of this type. See Example F for detail.	XML	in and out	not set
jsonWriteRootAsField	For JSON marshalling, specifies whether to write the root name (specified by the `value` attribute) into a field specified by the annotation `jsonRootField`. See Example G for detail. If `jsonWriteRootAsField` is `false` and the annotated type has a super class annotated as well, the JSON root is marshaled so that the type information exists in JSON representation which is useful during unmarshalling.	JSON	out	false
jsonRootField	Specifies the field where the JSON root name is displayed.	JSON	in and out	json-root
xmlNS	Specifies the namespace for XML representation. See Example H for detail.	XML	out	not set inherited from the parent class

Attributes	Description	Supported Formats	Marshal / Unmarshal	Default
xmlNSPrefix	Specifies the namespace prefix for XML representation. See Example H for detail.	XML	out	not set inherited from the parent class
inheritValue	Specifies whether to reuse the serializable name of the super type. • true: the serializable name of this type is set to the serializable name of the super type.Only types that do not need to be unmarshalled can have this attribute set to `true`. • false: the serializable name of this type depends on the value attribute.	JSON/XML	in and out	false

Examples

This section lists several examples explaining the detail usage of the attributes in the `@SerializableType` annotation.

Example A - Public Field Visibility In the following example, the non-public field `id` is not serialized.

Example B - None Field Visibility In the following example, only the field `active` annotated by `@SerializableField` is serialized.

Example C - Field Order In the following example, fields are serialized according to the order specified in `fieldOrder`.

Example D - Ignore Null Fields In the following example, the null field `id` is not serialized.

Example E - Inline Field In the following example, the inline filed `event` is moved to an upper level in the REST representation and only fields in `event` are serialized.

Example F - XML Value Field In the following example, `vstamp` is serialized as the value of the business-object element and `id` is serialized as an attribute.

Example G - JSON Write Root In the following example, `business-object`, which is specified in `value` is added to `name`, which is the default value of `jsonRootField`.

Example H - XML Namespace In the following example, XML namespaces and prefixes are added.

@SerializableField

The annotation `@SerializableField` indicates that a Java object field is intended to be serialized to a sub element or attribute of a REST representation. This annotation provides a number of attributes enabling you to customize the marshalling/unmarshalling behavior.

Table 8. SerializableField Attributes

Attributes	Description	Supported Formats	Marshal /Unmarshal	Default
value	Specifies the node name of the serializable type on one field of the class, for example, the name of an element in XML. See Example I for detail.	XML/JSON	in and out	not set
required	Specifies whether a field is required to be non-null for marshalling. If the field is required but its value is null, an exception is thrown during marshalling.	XML/JSON	out	false
override	Specifies whether to override the representation of parent's field with same serializable name.	XML/JSON	in and out	false
xmlWriteTypeRoot	This setting applies to a field of complex type. A complex type refers to a custom type annotated with `SerializableType`. Specifies whether to write `SerializableType.value` instead of `SerializableField .value` as the direct XML child element for a complex type. • true - write `SerializableType .value` as the direct XML child element for a complex type. • false - write `SerializableField.value`	XML	in and out	false

Attributes	**Description**	**Supported Formats**	**Marshal /Unmarshal**	**Default**
	as the direct XML child element for a complex type.			
defaultImpl	Specifies the default implementation class of a field for unmarshalling when the field implements an interface or extends an abstract class.	XML/JSON	in	DEFAULT .class
xmlAsAttribute	Specifies whether the field is represented as an attribute in the XML element attribute or a child element. See Example J for detail.	XML	in and out	false
xmlListUnwrap	Specifies whether to unwrap the items from a `java.lang.List` type field as the direct member of the serialized object. • When being unwrapped, the collection members of this field are moved to an upper level to be direct members of the serialized object in REST representation. • When not being unwrapping, this field is marshalled and unmarshalled as usual. See Example K for detail.	XML	in and out	false
xmlListItemName	Specifies the XML element name for the value list of a list data type. This attribute takes effect only when the field to serialize is composed of a list of simple data types, such as `List<String>`. If the field is a list of custom classes, such as `List<MyType>`, the attribute does not work and the default value `item` is used as the element name for the list. See Example L for detail.	XML	in and out	item

Attributes	**Description**	**Supported Formats**	**Marshal /Unmarshal**	**Default**
xmlNS	Specifies the namespace for a certain field in XML representation. If the annotated field is an instance of a certain class that has a different namespace specified, the class namespace overrides this setting. See Example M for detail.	XML	out	not set inherited from the parent class
xmlNSPrefix	Specifies the namespace prefix for a certain field in XML representation. If the annotated field is an instance of a certain class that has a different namespace prefix specified, the class namespace prefix overrides this setting. See Example M for detail.	XML	out	not set inherited from the parent class

Examples

This section lists several examples explaining the detail usage of the attributes in the `@SerializableField` annotation.

Example I - Field Serializable Name In the following example, the field `vstamp` is serialized as `version-stamp`.

Example J - Field as XML Attribute In the following example, the field `vstamp` is serialized as an attribute.

Example K - Field as XML Unwrapped List In the following example, the list of `outEvents` are unwrapped.

Example L - Field List XML Item Name In the following example, the `inEvent` list uses the default name `item` for list items while the `outEvent` uses `out-event`.

Example M - Field XML Namespace In the following example, XML namespaces and prefixes are added.

Out-of-box Annotated Models

Documentum REST Services provides a set of core models that are annotated with the REST annotations out of the box. You can use them directly as the model of custom resources or extend them with additional fields.

- AtomFeed
- RestError
- Repository
- PersistentObject
- Other model types in the Java package `com.emc.documentum.rest.model`

Additional Examples

This section lists several examples explaining the usage of REST annotations. Most of the examples use the out-of-box annotated models.

Example Q- Core Document Object The following example illustrates how out-of-box annotated core document objects are serialized.

Example R- Core Repository Object The following example illustrates how out-of-box annotated core repository objects are serialized.

Example S- Core Error Object The following example illustrates how out-of-box annotated core error objects are serialized.

Example T- Core Atom Feed With Src Entry The following example illustrates how out-of-box annotated atom feed objects with `scr` entries are serialized.

Example U- Core Atom Feed With Inline Entry The following example illustrates how out-of-box annotated atom feed objects with embedded entries are serialized.

Java Primitive Types Support

Documentum REST marshalling framework supports the following Java primitive types together with their wrapper classes:

Table 9. Supported Primitive Types and Wrapper Classes

Primitive Type	Wrapper Class
byte	Byte

Primitive Type	Wrapper Class
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

The primitive type `char` is not supported in the marshalling framework. Instead, the framework supports `String` to cover all the functionalities that `char` provides.

The following diagram illustrates the mapping between Java primitive types and XML/JSON data types.

Figure 6. Java Primitive Types and XML/JSON Data Types Mapping

In JSON representation, the type `Number` does not distinguish between integer and float-point values, which means:

- During marshalling, the data of the following types, including their wrapper classes, are represented as the type `Number` in JSON.
  - byte
  - short
  - int
  - long
  - float
  - double
- During unmarshalling, a JSON `Number` value is converted based on the type information of the Java class model.

  In this process, if a JSON `Number` value contains a higher precision than what the Java class model defines, the unmarshalling fails.

  For example, if JSON representation contains `"score":  9.5`, and the model has a field `int score`, the unmarshalling fails. In this case, modify the value of `score` to an integer, such as 9, or modify the type of `score` in the model to a type with a higher precision, such as `float`.

Limitation in Unmarshalling Number Lists

The current framework has a limitation in unmarshalling XML representations that contain number lists. This is because the framework cannot tell the level of precision of a number from an `item` element. For example, `<item>10</item>` can be a `byte`, but it can also be an `int`. Similarly, `<item>9.5</item>` can be a `float`, but it can also be a `double`. In this case, the framework adopts the lowest precision type that is compatible with the value. Therefore, `<item>10</item>` is unmarshalled to a `byte` and `<item>9.5</item>` is unmarshaled to a `float`.

Array Support

For all supported Java primitive types in the REST marshalling framework, the corresponding array form is also supported. For example, the following code snippet serializes an array of boolean values. **Java Code**

```
@SerializableField
private boolean[] feedCustomizedBooleanArray = {true, false, false};
```

XML Representation

```
<feedCustomizedBooleanArray>
        <item>true</item>
        <item>false</item>
        <item>false</item>
</feedCustomizedBooleanArray>
```

JSON Representation

```
"feedCustomizedBooleanArray": [
        true,
        false,
        false
    ],
```

Moreover, the framework supports the array form of any custom type instances. For an instance of a custom type, the return of its `toString` method is used as the value when being marshaled to XML or JSON. **Custom type Color**

```
public enum Color {
    RED(255,0,0),
    BLUE(0,0,255),
    BLACK(0,0,0),
    YELLOW(255,255,0),
    GREEN(0,255,0);

    private Color(int redValue,int greenValue,int blueValue){
        this.redValue=redValue;
        this.greenValue=greenValue;
        this.blueValue=blueValue;
    }

    public String toString(){
        return super.toString()+"("+redValue+","+greenValue+","+blueValue+")";
    }

    private int redValue;
    private int greenValue;
    private int blueValue;
}
```

Java Code

```
@SerializableField(xmlNS = "http://ns.customization.com/", xmlNSPrefix = "customization")
```

```
private Color[] colorArray = {YELLOW, BLUE};
```

XML Representation

```
<customization colorArray
            xmlns:customization="http://ns.customization.com/">
            <customization:item> YELLOW(255,255,0) </customization:item>
            <customization:item> BLUE(0,0,255)</customization:item>
</customization:colorArray >
```

JSON Representation

```
" colorArray ": [
                " YELLOW(255,255,0) ",
                " BLUE(0,0,255)",
            ]
```

Java Collection Support

The marshalling framework supports `List` and its subtypes. `Map` is not supported.

In JSON representation, `List` is bound to array. In XML representation, each element in a list is a sub-element of the element representing the list. See Example K for detail.

Circular References NOT Supported

The Documentum REST marshalling framework does not support circular references. If a data model has a field referencing itself and the field is annotated, then marshalling framework returns a stack overflow error.

Specifying Default Namespaces

You can specify default namespaces for custom resources by modifying `rest-api-base-marshalling.xml` under the `WEB-INF\classes\META-INF\spring` folder of the WAR file.

1. Locate the following bean in `WEB-INF\classes\META-INF\spring`:

   ```
   <bean id="xmlWriter" class="com.emc.documentum.rest.wire.xml
   .AnnotatedXmlMessageWriter">
   ```

2. In the `xmlWriter` bean, specify the prefixes and namespaces in the following pattern:

   ```
   <property name="namespaceBinding">
               <map>
                   <entry key="Prefix_1" value="Namespace_1"/>
                   <entry key="Prefix_2" value="Namespace_2"/>
                   …
                   <entry key="Prefix_N" value="Namespace_N"/>
               </map>
           </property>
           <property name="suggestedNamespace" value="Suggested_Namespace />
           <property name="showPreDefinedNamespacesInRoot" value="true"/>
   ```

For example:

```
    <bean id="xmlWriter"
     class="com.emc.documentum.rest.wire.xml.AnnotatedXmlMessageWriter">
         <property name="namespaceBinding">
             <map>
                 <entry key="" value="http://identifiers.emc.com"/>
                 <entry key="acc" value="http://acme.cc"/>
                 <entry key="acme" value="http://acme.com"/>
                 <entry key="ecc" value="http://emc.cc"/>
             </map>
         </property>
         <property name="suggestedNamespace" value="http://identifiers.emc.com"/>
         <property name="showPreDefinedNamespacesInRoot" value="false"/>
     </bean>
</beans>
```

The `showPreDefinedNamespacesInRoot` property determines whether or not to display namespaces defined in child objects, which defaults to `true`.

Settings of the `xmlNSPrefix` and `xmlNS` attributes in @SerializableType override the default namespaces settings in this Spring context file.

Annotation Scanner

An annotation scanner is available in the `tools` directory of the SDK to validate custom resources focusing on the annotation aspect. This annotation scanner runs when you build up the WAR package and generates a report listing all invalidated annotation occurrences in your code, such as `Field not serializable` and `Duplicated annotation values`.

The annotation scanner can be used as a runnable JAR or a Maven plug-in.

Runnable JAR

To use the annotation scanner as a runnable JAR, navigate to `tools` directory of the SDK and then run the following command:

```
java -jar annotation-validator-runnable [TARGET] [option]
```

`[TARGET]` represents the source files to scanner, which can be:

- A single .JAR, WAR, or EAR file, such as `./tmp/dctm-rest.jar`
- Multiple JAR or WAR files separated by comma, such as `./tmp/databinding.jar, ./tmp/test-annotation.jar`
- A directory containing the binary, JAR, or EAR files, such as `./tmp`

`[option]` represents the options of this command:

- -h Display help messages
- -o Specify the output directory, such as `-o ./output`
- -X Display debug messages

Maven Plug-In

To use the annotation scanner as a Maven plug-in:

1. Navigate to `tools` directory of the SDK and then run the following command to install the plug-in to your local repository:

   ```
   mvn install:install-file -Dfile=documentum-rest-annotation-plugin
   -version-number.jar -DpomFile=pom.xml
   ```

2. Open the `pom.xml` file of your project and then append the following plug-in to the `plugins` block .

   ```
   <plugin>
       <groupId>com.emc.documentum.rest</groupId>
       <artifactId>documentum-rest-extension-validating</artifactId>
       <version>7.2</version>
       <inherited>true</inherited>
       <executions>
           <execution>
               <id>validate-annotation</id>
               <phase>verify</phase>
               <goals>
                   <goal>check-annotation</goal>
               </goals>
               <configuration>
                   <input>${project.basedir}/target/${scan.artifactId}-${version}.war</input>
                   <outputDir>${project.basedir}/target</outputDir>
                   <debug>false</debug>
               </configuration>
           </execution>
       </executions>
   </plugin>
   ```

   In the plug-in configuration:

   - The `input` element specifies the source files to scan.
   - The `outputDir` element specifies directory to hold the report.
   - The `debug` element determines whether the debug mode is turned on when the scanner runs.

When the scan completes, an HTML file named `Scan Report of Documentum Rest Extensibility Annotation` is generated in the output directory. This file lists all annotation violations and fix recommendations. All issues should be resolved before you build the custom REST WAR file.

If necessary (though not recommended), you can disable this scanner by removing the plugin configuration in the `pom.xml` file.

Developing Custom Resources

This section guides developers to develop simple custom resources with the Documentum REST Extensibility.

Typically, custom resources development can be divided into the following phases:

1. **Designing Custom Resources**

   This phase focuses on the following tasks:

   - Designing the XML and/or JSON representation of your custom resources
   - Determining how client applications discover your custom resources
   - Determining the operations to support

2. **Setting up a Custom Resource Project**

   In this phase, you will set up a project with the appropriate structure to hold various packages, source files, and configuration files. We recommend that you leverage the Maven or Ant toolkit provided in the SDK for project setup.

3. **Programming for Custom Resources**

   In this phase, you will take advantage of the marshalling framework and core REST Java library to develop custom resources.

4. **Linking Custom and Core Resources**

   In this phase, you will make the newly-created resources discoverable from existing core REST resources or the Home Document via HATEOAS links.

5. **Packaging and Deploying**

   In this phase, you will package all your source files to build up a WAR file and then deploy the WAR file to your application server.

Designing Custom Resources

Typically, a custom resource can represent a persistent object, a collection of persistent objects, or a certain state of the persistent objects. A key property that distinguishes the REST architecture from other software architectures is that REST is resource-oriented. So the first step for custom resource development is to model the persistent data into resources. If the persistent data is too large, has deep hierarchy, or there are a lot of operations on it, you may consider designing more than one resources for the large persistent data.

URI

The URI design for a resource is implementation-specific. Theoretically, you can design any URI pattern for a custom resource. Practically, however, the custom resource URI pattern should be similar to that of the Core resources. One benefit is to apply the same authentication schemes as Core resources, because Core REST security component checks the resource URLs to determine whether authentication is required on the custom resources. For instance, all repository level resources are required for authentication by default, so the URI of a custom resource under a particular repository should follow this pattern: `/repositories/`*`repositoryName`*`/`*`customSegments`*.

Note: When the designed resource URI template contains path variables where their values come from the object properties, there could be URI encoding/decoding issues if the property values contain URI preserved characters. Documentum REST Services SDK provides a utility `com.emc.documentum.rest.utils.NameAsPathCoder` helping you encode and decode the path variable values to escape special characters.

HTTP Methods

HTTP /1.1 specification provides several standard HTTP methods for clients to perform on a resource. Although you are allowed to create custom HTTP methods on a custom resource, the challenges it brings is usually far more than the given benefits. So it is recommended to stick to standard HTTP methods for resource operations. In case there are a lot of logic operations applied to the same persist data that the resource represents for, you may consider dividing the operations into multiple resources.

Core resources use standard HTTP methods GET, POST, PUT and DELETE in all places.

Representations

Core resources support both JSON and XML representations. For a custom resource, whether to support one or both representations depends on your business requirements. Documentum REST Services provides an annotation-based marshalling framework so that an annotated Java model can support both JSON and XML representations out of the box. You do not need to handle the message marshalling or unmarshalling except for the Java model design.

If you want to limit the custom resource to support JSON or XML representation only, specific media type constraints can be applied to the resource controller to precisely define the supported representation with the Spring annotation `@RequestMapping`.

Content Negotiation

Custom resources follow the same content negotiation mechanism as Core resources to determine a specific representation format for a client request.

In both Core and custom resources, all operations support the following media types:

- application/atom+xml
- application/vnd.emc.documentum+xml
- application/vnd.emc.documentum+json

The GET operation also supports the following two generic media types:

- application/xml
- application/json

The current Documentum REST Extensibility feature does not support custom media types.

Link Relations

In most cases, a custom resource design has one or more outer links pointing to Core resources or other custom resources. This will be done by designing new link relations on the custom resource representation. It is recommended to look up well-known link relations first from IANA (http://www.iana.org/assignments/link-relations/link-relations.xhtml) and use them as much as possible before considering inventing new link relation names for the custom resources. Besides, a good practice for inventing new link relation names is that nouns are preferred other than verbs to name a link relation.

It is possible to add new links to Core resources. The section Adding Links to Core Resources has the details.

Setting up a Custom Resource Project

Documentum REST Services SDK provides you with a convenient way to set up the first custom resource project with a Maven archetype. This section introduces the general project structure for custom resource development.

Documentum REST resource development mainly leverages Spring Web MVC to construct the resources and uses Maven to build projects. Therefore, each resource implementation must have well-formed code structure as follows.**Typical project structure**

```
rest-api-foo
    |---rest-api-foo-model
    |---rest-api-foo-persistence
    |---rest-api-foo-resource
```

This code structure is a typical Documentum REST resource project structure, which should be aligned with in your custom resource structure so that the implementation is well layered. This code structure separates the custom resource implementation into three modules:

- model

  Designs annotated Java model classes for custom resources

- persistence

  Creates persistence API beans by referencing DFC or other local persistence APIs

- resource

  Creates resource controllers for custom resources

With this code structure, the custom model, persistence, and controller implementations are built as separate JAR files.

For a custom resource that does not require additional models or persistence APIs, the corresponding module and/or persistence can be omitted.

For each sub module, the code structure is organized as a typical Maven module.

Typical module structure

```
rest-api-foo
    |---rest-api-foo-xxx
        |---src
            |---main
                |---java
                |---resources
            |---test
                |---java
                |-resources
        |---pom.xml
|---pom.xml
```

The quick start is to use the Maven archetype project in Documentum REST Services SDK to create such a project structure.

Programming for Custom Resources

This section introduces the important features of the custom resource programming. See Tutorial: Documentum REST Services Extensbilitiy Development for a comprehensive review of the typical custom resource development.

Model: Programming

The model package defines the data structure of the resource representation for both input and output. Being decorated with Documentum REST marshalling framework, an annotated model class has direct data binding to its resource representation. You only need to focus on the decision of what domain information to be exposed in the model, and the marshalling framework takes full responsibility for marshalling and unmarshalling the data model into/from XML and JSON messages. Here is an example of a new resource model.

```
/**
   * Define a server model which has three properties.
  */
@SerializableType(value = "server",
        fieldVisibility = SerializableType.FieldVisibility.ALL,
        fieldOrder = {"name", "host", "version"},
        xmlNS = "http://identifiers.emc.com/vocab/documentum",
        xmlNSPrefix = "dm")
public class Server {
    private String name;
    private String host;
    private String version;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getHost() {
        return host;
    }
    public void setHost(String host) {
        this.host = host;
    }
    public String getVersion() {
        return version;
    }
    public void setVersion(String version) {
        this.version = version;
 }
}
```

Note: An annotated model class must have one parameterless constructor. Otherwise, an exception is thrown during unmarshalling. In the sample above, we do not define any constructor for the `Server` class so that the default (no-argument) one provided by the complier is applied. If you define a constructor with arguments for the model class, you must explicitly define one parameterless constructor at the same time.

Model: Extending Core

You can also extend Core REST resource model classes for holding additional information on the custom resource representation. The Core REST library provides a persistent object model class `com.emc.documentum.rest.model.PersistentObject` that provides the fundamental properties and links collection for a persistent Documentum object. Here is an example of an extended persistent object model.

```
/**
   * Define a business object model which has one additional property 'uuid'.
  */
@SerializableType(value = "biz-object",
        jsonWriteRootAsField = true,
        fieldVisibility = SerializableType.FieldVisibility.NONE,
        ignoreNullFields = true,
        jsonRootField = "name",
        fieldOrder = {"type", "definition", "uuid", "properties"},
        xmlNS = "http://identifiers.emc.com/vocab/documentum",
        xmlNSPrefix = "dm")
public class BusinessObject extends com.emc.documentum.rest.model.PersistentObject {

    @SerializableField (xmlAsAttribute = true)
    private String uuid;

    public String getUuid() {
        return uuid;
    }

    public void setUuid(String uuid) {
        this.uuid = uuid;
    }
}
```

Note: There are more model samples in the SDK: `documentum-rest-`*`version`*`.zip/samples /documentum-rest-model-samples/`

Model: Validation

It is always good to validate the designed models with the Documentum REST Annotation Scanner. The tool can be run during the process of Maven project creation or run separately in a command line. The scanning report tells the detail of the model analyze result, as well as fix instructions. All ERROR level rule violations must be fixed. Here is a sample of the scan report.

Figure 7. Scan Report

Persistence: Programming

The persistence API wraps the DFC operations to provide persistent data operations for the resources. The best practice for creating a new persistence API is to separate the API into one interface and one implementation class, and load the implementation class by using a Java bean.

1. Create a Java interface.

   ```
   public interface UserManager {
       com.emc.documentum.rest.model.UserObject createUser(UserObject newUser);
   }
   ```

2. Create a Java class to implement this interface.

   ```
   public class UserManagerImpl
       extends com.emc.documentum.rest.dfc.SessionAwareAbstractManager
           implements UserManager {
   public com.emc.documentum.rest.model.UserObject createUser(UserObject newUser) {
        …
       }
   }
   ```

3. Put the implementation as a Java bean in class-path `/META-INF/spring/custom.xml`.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <beans xmlns=http://www.springframework.org/schema/beans ..>
       <bean id="customUserManager" class="com.acme.UserManagerImpl" />
   </beans>
   ```

The beans then could be referenced by the resource controllers with Spring annotation `@Autowired`.

Here is an example of the persistence API reference in a resource controller.

```
@Controller("acme#user")
@RequestMapping("/repositories/{repositoryName}/users-x/{userName}")
public class CustomUserController extends AbstractController {

    // reference the persistence API user manager in resource controller
    @Autowired
    UserManager userManager;

    @RequestMapping(method = RequestMethod.GET)
    @ResponseStatus(HttpStatus.OK)
    @ResponseBody
    @ResourceViewBinding(UserView.class)
    public UserObject createUser(
```

```
            @PathVariable("repositoryName") final String repositoryName,
            @ PathVariable ("userName") final String userName,
            @RequestUri final UriInfo uriInfo) throws DfException {
        UserObject user = userManager.get(userName, password);
        return getRenderedObject(repositoryName, user, param.isLinks(), uriInfo, null);
    }
}
```

Persistence: Referencing Core

Documentum REST Services SDK library provides lots of persistence APIs for developers to reuse to operate the persistent Documentum objects. For example, `com.emc.documentum.rest.dfc.ObjectManager` provides basic CRUD operations for any persistent objects; `com.emc.documentum.rest.dfc.SysObjectManager` provides additional methods for sysobject related operations, such as copy and checkout. Custom resource controllers can reference the instance of these Core persistence APIs as same as to custom persistence APIs.

Note: The complete persistence API docs can be found in the SDK: `documentum-rest-`*`version`*`.zip/apidocs/`

Persistence: Session Management

Documentum REST Services SDK library provides the interfaces `com.emc.documentum.rest.dfc.RepositorySessionManager` and `com.emc.documentum.rest.dfc.RepositorySession` for the session management in the persistence APIs. The default implementations for these two interfaces have been integrated with the security module of the REST services. Once a user logs in, the persistence API retrieves the right user session without a need to instantiating the DFC session manager explicitly. The custom persistence API can extend `com.emc.documentum.rest.dfc.SessionAwareAbstractManager` to get the session easily. Here is an example.

```
/**
   * A user manager implementation extends SessionAwareAbstractManager to reuse the session beans.
**/
public class CustomUserManager extends SessionAwareAbstractManager implements UserManager {

    @Override
    public UserObject get(String userName, AttributeView attributeView) throws DfException {
        IDfSession session = null;
        try {
            // user the method from super class to get a session for the login user
            session = getSessionRepository().getSession(false);
            IDfUser dfUser = session.getUser(userName);
            if(dfUser == null) {
                return null;
            } else {
                return convert(dfUser, attributeView);
            }
        } finally {
                //do not forget to release the session
                release(session);
        }
    }
}
```

When not extending the `SessionAwareAbstractManager`, the persistence API can use the auto wired `RepositorySession` instance to retrieve a session by using the Spring annotation `@Autowired`. Here is the other example.

```
public class CustomUserManager implements UserManager {
    // reference the repository session bean directly
    @Autowired
    private RepositorySession sessionRepository;
    @Override
    public UserObject get(String userName, AttributeView attributeView) throws DfException {
        IDfSession session = null;
        try {
            session = sessionRepository getSession(false);
            IDfUser dfUser = session.getUser(userName);
            if(dfUser == null) {
                return null;
            } else {
                return convert(dfUser, attributeView);
            }
        } finally {
            release(session);
        }
    }
```

In the unit testing of the persistence API where the security module of Documentum REST Services does not take place, the test code can set the login user name and password as shown in the following snippet to make the session available for the test code. The code mocks the user login from a servlet request.

```
com.emc.documentum.rest.config.RepositoryContextHolder.setRepositoryName(..)
com.emc.documentum.rest.config.RepositoryContextHolder.setLoginName(..)
com.emc.documentum.rest.config.RepositoryContextHolder.setPassword(..)
```

If transaction is required on the resource controller, the class `com.emc.documentum.rest.dfc .ContextSessionManager` can be used to commit a transaction for the persistence API as shown in the following example.

```
public class CustomSysObjectManager implements SysObjectManager {
 @Autowired
 ContextSessionManager contextSessionManager;
             @Override
 public <T extends SysObject> T copy(final String objectId, final String folderId)
                   throws DfException {
  // put a regular operation into the transaction manager
                           return contextSessionManager.executeWithinTheContextTran
                           (new Callable<T>() { public T call() throws Exception {
  return doRegularCopy(objectId, folderId);
  }});
 }
}
```

Note: There are more session management samples in the SDK: `documentum-rest-`*`version`*`.zip /samples/documentum-rest-java_api-samples/`

Persistence: Creating Feed Pages from DQL Result

For a feed resource implementation, the persistence API may execute a DQL query to get the object collection. Documentum REST Services SDK library provides the class `com.emc.documentum.rest.paging.Page<T>` to produce the object collection as a feed page. Another class `com.emc.documentum.rest.dfc.query.PagedQueryTemplate<T>` is the basic DQL template to produce a DQL query expression. A custom feed resource extends `PagedQueryTemplate` to construct a custom DQL query expression, and calls `com.emc.documentum.rest.paging.PagedDataRetriever<T>` to get a `Page<T>`. Here are several samples.

Use the PagedPersistentDataRetriever to get a page for a custom PagedQueryTemplate

Documentum REST Services SDK library provides a default implementation of `PagedDataRetriever` `com.emc.documentum.rest.paging.PagedPersistentDataRetriever` to retrieve a page of persistent objects. The return type is `com.emc.documentum.rest.model.PersistentObject` or its sub types.

```
// Define a custom DQL for the document collection
PagedQueryTemplate pagedQueryTemplate
= new CustomPagedQueryTemplate(getUsername());

// Retrieve a page of document objects with the default object collection
implementation and paging parameters.
PagedDataRetriever<DocumentObject> pagedDataRetriever
= new PagedPersistentDataRetriever<DocumentObject>(
    pagedQueryTemplate, 1, 5, true, AttributeView.DEFAULT, DocumentObject.class);
Page<DocumentObject> page = pagedDataRetriever.get();
```

Use the PagedDataRetriever to get a page for a custom PagedQueryTemplate with custom models

You can also create a custom object collection manager if the data model in the page does not extend `com.emc.documentum.rest.model.PersistentObject`.

```
PagedQueryTemplate pagedQueryTemplate
= new CustomPagedQueryTemplate(getUsername());

PagedDataRetriever<MyDocument> pagedDataRetriever
= new PagedDataRetriever<MyDocument>(
        new MyDocumentCollectionManager(),
pagedQueryTemplate, 1, 5, true, AttributeView.DEFAULT);
Page<MyDocument> page = pagedDataRetriever.get();
```

Use the Paginator to produce the page from any generic collection

For a feed resource implementation where its persistent data does not come from a simple DQL, you can create your own implementation of the persistence layer and call `com.emc.documentum.rest.paging.Paginator` to generate a page from the object collection.

```
String dql = "select * from dm_document";
session = dfcSessionRepository.getSession();
List<IDfTypedObject> results = QueryExecutor.run(session, dql);
List<DocumentObject> docs = convert(results);
List<IDfTypedObject> counts = QueryExecutor
            .run(session, "select count(r_object_id) as total from dm_document");
int total = counts.get(0).getInt("total");
Paginator<DocumentObject> paginator = new Paginator<DocumentObject>(docs, total);
int pageNumber = 3;
int itemsPerPage = 5;
```

```
Page<DocumentObject> currentPage = paginator.paginate(pageNumber, itemsPerPage);
```

Note: There are complete page samples in the SDK: `documentum-rest-`*`version`*`.zip/samples /documentum-rest-java_api-samples/`

Resource: Programming the Controller

The resource implementation leverages the Spring controller to define the REST mapping of the resource. The controller class has Spring REST annotations to define its request mapping and response mapping. It loads the auto wired persistence API to manipulate the model data. The return type of the controller method is the model class. Here is an example of the resource controller.

```
@Controller("format")
@RequestMapping("/repositories/{repositoryName}/formats-x/{formatName}")
@ResourceViewBinding(value = FormatView.class)
public class MyFormatController extends AbstractController {
 @Autowired
 private FormatManager dfcFormatManager;
 @RequestMapping(
   method = RequestMethod.GET, produces = {
  SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
  SupportedMediaTypes.APPLICATION_VND_DCTM_XML_STRING,
   MediaType.APPLICATION_JSON_VALUE,
   MediaType.APPLICATION_XML_VALUE
 })
 @ResponseBody
 @ResponseStatus(HttpStatus.OK)
 public Format get(
   @PathVariable("repositoryName") String repositoryName,
   @PathVariable("formatName") String formatName,
   @TypedParam final SingleParam param,
   @RequestUri final UriInfo uriInfo) throws DfException {
  Format format = dfcFormatManager.getFormat(NameAsPathCoder.decode(formatName),
     param.getAttributeView());
  if (format == null) {
   throw new DfNoMatchException(NameAsPathCoder.decode(formatName));
  }
  return getRenderedObject(repositoryName, format, param.isLinks(), uriInfo, null);
 }
}
```

The controller class specifies a Spring annotation `@Controller` to indicate it as a resource controller. We recommend that you put a unique name on the `@Controller`. The name on the controller is taken as the code name of this resource and will be used by configurations and loggings to reference this resource.

The controller class defines its request mapping with the Spring annotation `@RequestMapping`. There are some additional mapping rules except for the URI on this annotation. Refer to Spring framework documentation for the usage.

On the controller definition, a Core REST annotation `@ResourceViewBinding` defines the default view implementation for this resource.

The controller class extends from `com.emc.documentum.rest.controller .AbstractController`, which provides some useful methods that can be reused by the extended controller.

In the controller method definition, `@RequestMapping` can be further defined to map the HTTP method and media types for the resource operation.

The controller method defines the Spring annotation `@ResponseBody`, which indicates that the returning object format can be directly sent to the marshalling framework.

The controller method defines the Spring controller annotation `@ResponseStatus`, which specifies the HTTP status of this resource operation. In case there is an exception thrown from this method, an error mapping will take place and the status will be reset to corresponding error status.

The resource method should call its controller super class method `getRenderedObject` or `getRenderedPage` to invoke dynamic view definitions to render the model instance to produce the links and the other representation customizations.

Note: There are more samples for the resource controllers in the SDK: `documentum-rest-`*`version`*`.zip/samples/documentum-rest-resource-samples/`

Resource: Programming the View

Views are implementations that customize the resource model instances for further information in the representation, especially for creating links for the resources.

There are three abstract view classes for view implementations: `LinkableView`, `EntryableView`, and `FeedableView` .

- `LinkableView` A non-collection resource (also called single data object resource) must have a `LinkableView` implementation. The extended class implementation must implement the abstract methods. Optionally, the protected methods can be overridden.
- `EntryableView` If the non-collection resource (also called single data object resource) can further be presented into the inline feed of a collection resource. Its view implementation instead should extend `EntryableView`. The extended class implementation must implement the abstract methods. Optionally, the protected methods can be overridden.
- `FeedableView`A collection resource must have a `FeedableView` implementation. The extended class implementation must implement the abstract methods. Optionally, the protected methods can be overridden.

Documentum REST Services library provides two view implementations to implement part of the methods. Custom views can extend these classes to reuse their methods.

- com.emc.documentum.rest.view.impl.PersistentLinkableView
- com.emc.documentum.rest.view.impl.DefaultFeedView

There are two view annotations used for the view implementations. A view implementation class must declare one of the following annotations for the corresponding usages.

A `@DataViewBinding` annotation is applied to a `com.emc.documentum.rest.view.LinkableView` or `com.emc.documentum.rest.view.EntryableView` implementation. The `@DataViewBinding` resolves which data model the view definition is bound to. Here is an example.

```
@DataViewBinding(modelType = RelationObject.class)
public class RelationView extends EntryableView<RelationObject> { .. }
```

A `@FeedViewBinding` annotation is applied to a `com.emc.documentum.rest.view.FeedableView` implementation. This annotation resolves inline `EntryableView` for a feed view. The entry views can be more than one since a feed may contain different types of models. Each entry view should correspond to a unique data model class defined by `@DataViewBinding`. Here is an example.

```
@FeedViewBinding(RelationView.class)
public class RelationsFeedView extends FeedableView<RelationObject> { .. }
```

Note: There are more samples for the resource views in the SDK: `documentum-rest-`*`version`*`.zip/samples/documentum-rest-resource-samples/`

Resource: Binding the view and the controller

As previously mentioned, a resource can be bound to a view implementation by putting the Core REST annotation `@ResourceViewBinding` on the resource controller. A `@ResourceViewBinding` could be applied on a controller class level or on the controller method level. The annotation has an attribute to specify which view definitions the resource controller (or method) uses to render the resource model in the controller method. There could be more than one view definitions bound to a controller or a method (repeating value). But each view definition should be corresponding to a unique resource model type (e.g., feed, document, folder, and etc.) Here are some differences between applying the annotation on the class level and on the method level.

- When the annotation is applied on the class level, all methods in the controller by default uses the view definitions in the class level annotation.
- When the annotation is applied on a class method level, the specific method in the controller by default uses the view definitions in the method level annotation. And it overrides the class level annotation.

Here is an example of using the `@ResourceViewBinding` on both class and method levels. The class level supports the feed representation by default, while the create method returns a single relation resource as the response.

```
@Controller("acme#relations")
@RequestMapping("/repositories/{repositoryName}/relations-x")
@ResourceViewBinding({RelationsFeedView.class})
public class RelationsController extends AbstractController  {

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public AtomFeed get(
            @RequestParam(value = “name”, required = false) final String relationName,
            @RequestUri final UriInfo uriInfo)
            throws MissingServletRequestParameterException, DfException {
        ...
    }
    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    @ResourceViewBinding(RelationView.class)
    public ResponseEntity<RelationObject> createRelation(
            @PathVariable("repositoryName") final String repositoryName,
            @RequestBody final RelationObject relationObject,
            @RequestUri final UriInfo uriInfo)
            throws DfException, MissingServletRequestParameterException {
```

```
        ...
    }
}
```

Note:

- A `@ResourceViewBinding` can bind to multiple View definitions. When multiple View definitions are registered for a `@ResourceViewBinding`, each view definition should be corresponding to a unique data model. There can be at most one `FeedableView` definition in the `@ResourceViewBinding` definition since `FeedableView` is corresponding to the `AtomFeed` data model .
- When `@ResourceViewBinding` is used on the method level, `@RequestUri final UriInfo uriInfo @RequestUri final UriInfo uriInfo` is required in the method definition.

In summary, resource controllers, models, and views are bound with one another by using Java annotations shown as in the following diagram.

Figure 8. Java Annotations and Documentum MVC

Resource: Building resource links

The abstract view classes `EntryableView`, `LinkableView`, and `FeedableView` provide some fundamental links for single object resources and feed resources. A custom resource view class extending one of these abstract classes can override or extend the links provided by the abstract class.

LinkableView

By default, the class `LinkableView` returns the `self` link for the resource model in the protected method `xselfLinks()`. The class provides several abstract or protected methods for overriding.

- To customize `self` link generation, you can override the method `selfLinks()`.
- To add additional links, you can implement the method `customize()`, and call `makeLink()`, `makeLinkIf()`, `makeLinkTemplate()`, or `makeLinkTemplateIf()` in the `customize()` method.
- To remove a specific link, you can call `removeLink()` in the `customize()` method.
- To clear all default links, you can call `clearLinks()` in the `customize()` method.

The method `makeLinkIf()` adds a link relation to the resource only when the condition is met at runtime. This method is useful for adding link relations upon certain conditions.

In the following sample implementation of the `customize()` method, we add an additional link relation 'author' to the alias set resource upon the existence of the `owner_name` attribute.

```
@Override
    public void customize() {
  // add the author link relation
  makeLinkIf(
    serializableData.getAttributeByName("owner_name") != null,
    LinkRelation.AUTHOR.rel(),
    getUriFactory().
        userUri((String)
        serializableData.getAttributeByName("owner_name"), null));
 }
```

EntryableView

The class `EntryableView` extends `LinkableView` with additional atom entry customizations. Besides of the link methods the class `LinkableView` provides, you can define your own entry links in the atom entry representation with method `entryLinks()`. If your custom resource view class extends the class `PersistentLinkableView`, a default link relation `edit` is added to the entry link collection.

FeedableView

By default, the class `FeedableView` returns the `self` link and pagination links (`first`, `next`, `previous`, or `last` depending on the current page position) for the atom feed model. You can override the method `feedLinks()` to modify these link relations or add other link relations.

In the following sample, we override the `feedLinks()` method to add an additional link relation `about`to the alias set feed resource.

```
@Override
 public List<Link> feedLinks() {
  List<Link> links = super.feedLinks();
  // add the about link relation to feed links
  links.add(new Link("about", getUriFactory().productInfoUri()));
  return links;
 }
```

Resource: Making It Queryable

The DQL query resource returns a collection of query result items as atom entries. The entry can contain a link pointing to the canonical resource of the typed object if there is a resource implementation for the specified object type.

To make the custom resource linkable from the DQL query resource, add query types on the controller annotation `@ResourceViewBinding` for your custom resource controller, as shown in the following sample:

```
@Controller("acme#alias-set")
@RequestMapping("/repositories/{repositoryName}/alias-sets/{aliasSetId}")
// make query results of type "dm_alias_set" linkable to the acme#alias-set resource
@ResourceViewBinding(value = AliasSetView.class, queryTypes = "dm_alias_set")
public class AliasSetController extends AbstractController { … }
```

Note: The *EMC Documentum Platform REST Services Resource Reference Guide* introduces the detail rules for what DQL expressions can produce resource links.

Resource: Deciding Whether to Be Batchable

By default, custom resources can be executed in the batch. You will find all custom resources in the `batchable-resource` list when you send a GET request to the Batch Capabilities resource. However, in some scenarios, you may not want to make a custom resource batchable. To help you manage the batch property of a custom resource or even a specific method of the custom resource, Documentum REST Services provides the following two annotations:

- com.emc.documentum.rest.model.batch.annotation.BatchProhibition
- com.emc.documentum.rest.model.batch.annotation.TransactionProhibition

@BatchProhibition

This annotation prevents client applications from embedding a custom resource in batch requests. To use the `@BatchProhibition` annotation, add it to the controller class of this resource as shown in the following code snippet:

```
@Controller
@BatchProhibition
public class MyResourceController {…}
```

When the controller class of a custom resource is annotated with **@BatchProhibition** , the resource is moved to the non-batchable-resources list of the Batch Capabilities resource.

Besides of adding `@BatchProhibition` to the class level, you can add this annotation to one or more methods in the controller class to prevent certain operations of the resource from being embedded to batch requests.

```
@Controller
public class MyResourceController {
…

    @RequestMapping(
            value = {"/users/{userName}/MyResources/{MyResourceId}"},
            method = RequestMethod.GET, produces = {
            SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
            SupportedMediaTypes.APPLICATION_VND_DCTM_XML_STRING,
            MediaType.APPLICATION_JSON_VALUE,
            MediaType.APPLICATION_XML_VALUE


    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    @BatchProhibition
public MyResource getMyResource (…)

   @RequestMapping(
            value = {"/users/{userName}/MyResources/{MyResourceId}"},
            method = RequestMethod.DELETE)
    @ResponseBody
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteMyResource(…)
…
}
```

This code snippet prevents the GET operation on `MyResource` from being embedded in batch requests. However, you can still perform a bulk delete on `MyResource` in a batch request because neither the `MyResourceController` class nor the `deleteMyResource` method is annotated with `@BatchProhibition`.

@TransactionProhibition

The `@TransactionProhibition` annotation enables you to remove the transaction support from a custom resource. Typically, you add the `@TransactionProhibition` annotation to the controller class of this resource as shown in the following code snippet:

```
@Controller
@TransactionProhibition
public class MyResourceController {…}
```

This setting prevents the custom resource from being embedded in transactional batch requests.

Similar to `@BatchProhibition`, the `@TransactionProhibition` annotation can also be added to one or more methods in the controller class to prevent certain operations of the resource from being embedded to transactional batch requests.

Note that the `@TransactionProhibition` annotation only affects custom resources' applicability in transactional batch requests. To prevent a custom resource from being embedded in all batch requests, use `@BatchProhibition`.

For more information about batch requests, see the Batch and Batch Capabilities sections in the *EMC Documentum Platform REST Services Resource Reference Guide*.

Resource: Extending Atom Feed and Entry

The custom resource which returns the object collection as a feed may need to add some additional attributes to the atom feed or the atom entry. This can be done by extending the model classes `com.emc.documentum.rest.model.AtomFeed` and `com.emc.documentum.rest.model.AtomEntry`, and then creating custom views for them.

The extended feed model should extend com.emc.documentum.rest.model.AtomFeed and specify the `@SerializableType` attribute `inheritValue` as true. Here is an example.

```
@SerializableType(inheritValue = true,
 xmlNS = "http://www.w3.org/2005/Atom", xmlNSPrefix = "atom")
public class AliasSetFeed extends AtomFeed {
 @SerializableField(xmlNS = "http://ns.acme.com/", xmlNSPrefix = "acme")
 private int score;
 public int getScore() {
   return score;
 }
}
```

This sample model contains a custom feed attribute 'score' but will still be marshaled as the same feed root `<feed …/>` for XML.

If the custom attributes are added to the atom entry, the view must extend `com.emc.documentum .rest.model.AtomEntry`, too. Here is the sample for the extended atom entry.

```
@SerializableType(inheritValue = true, xmlNS = ", xmlNSPrefix = "atom")
public class AliasSetEntry extends AtomEntry {
    @SerializableField("own-by-login-user")
    private boolean ownByLoginUser;
    public boolean isOwnByLoginUser() {
        return ownByLoginUser;
    }
    public void setOwnByLoginUser(boolean ownByLoginUser) {
        this.ownByLoginUser = ownByLoginUser;
    }
}
```

After the model definition, you need to create custom views for the extended feed and entry to populate the custom attribute data. For a feed view definition, the class needs to explicitly declare its binding feed type as the extended feed type. Here is the feed view example.

```
@FeedViewBinding(value = AliasSetViewX.class, feedType = AliasSetFeed.class)
public class AliasSetsFeedViewX extends FeedableView<AliasSet> {
    public AliasSetsFeedViewX(Page<AliasSet> page, UriInfo uriInfo,
    String repositoryName, Boolean returnLinks, Map<String, Object> others) {
        super(page, uriInfo, repositoryName, returnLinks, others);
    }
    @Override
    public String feedTitle() {
        return "Alias Sets";
    }
    @Override
    public Date feedUpdated() {
        return new Date();
    }
    @Override
    protected void customizeFeed(AtomFeed feed) {
        AliasSetFeed aliasSetFeed = (AliasSetFeed) feed;
        aliasSetFeed.setScore(new SecureRandom().hashCode());
    }
}
```

For an entry view definition, it needs to explicitly declare the binding entry type is the extended atom entry type. Here is the entry view example.

```
@DataViewBinding(queryTypes = "dm_alias_set",
modelType = AliasSet.class, entryType = AliasSetEntry.class)
public class AliasSetViewX extends PersistentLinkableView<AliasSet> {
    public AliasSetViewX(AliasSet aliasSet, UriInfo uriInfo,
    String repositoryName, Boolean returnLinks, Map<String, Object> others) {
        super(aliasSet, uriInfo, repositoryName, returnLinks, others);
    }
    ...

    @Override
    protected void customizeEntry(AtomEntry atomEntry) {
        AliasSetEntry aliasSetEntry = (AliasSetEntry) atomEntry;
        aliasSetEntry.setOwnByLoginUser(
                RepositoryContextHolder.getUserName().equals
                (getDataInternal().getAttributeByName("owner_name")));
    }
}
```

Last, in the resource controller, the implementation is same to non-extended atom feed model. The actual returning instance extends atom feed. Here is the controller example.

```
@Controller("acme#alias-sets")
@RequestMapping("/repositories/{repositoryName}/alias-sets")
public class AliasSetCollectionController extends AbstractController {
    @RequestMapping(
            method = RequestMethod.GET,
            produces = {
            SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
            MediaType.APPLICATION_ATOM_XML_VALUE,
            MediaType.APPLICATION_JSON_VALUE,
            MediaType.APPLICATION_XML_VALUE
    })
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    @ResourceViewBinding(AliasSetsFeedViewX.class)
    public AliasSetFeed getAliasSets(
            @PathVariable("repositoryName") final String repositoryName,
            @TypedParam final CollectionParam param,
            @RequestUri final UriInfo uriInfo)
            throws Exception {
        PagedQueryTemplate template = new AliasSetCollectionQueryTemplate()
                .filter(param.getFilterQualification())
                .order(param.getSortSpec().get())
                .page(param.getPagingParam().getPage(), param.getPagingParam().getItemsPerPage());
        PagedDataRetriever<AliasSet> dataRetriever = new PagedPersistentDataRetriever<AliasSet>(
                template,
                param.getPagingParam().getPage(),
                param.getPagingParam().getItemsPerPage(),
                param.getPagingParam().isIncludeTotal(),
                param.getAttributeView(),
                AliasSet.class);
        return (AliasSetFeed) getRenderedPage(
                repositoryName,
                dataRetriever.get(),
                param.isLinks(),
                param.isInline(),
                uriInfo,
                null);
    }
}
```

Resource: Error Handling and Representation

Documentum REST Services leverages the Spring framework to resolve the error mapping from Java exceptions to error messages. The Documentum REST Services SDK library provides the default implementation `com.emc.documentum.rest.error.http.GeneralExceptionMapping` that defines a lot of error mappings for various Java exception classes. If you want to define your own error mappings in custom resources, you must extend `GeneralExceptionMapping` and update the following bean with the custom class, which is in the `rest-api-base-marshalling.xml` file of the WAR classpath.

```
<bean id="generalExceptionMapping"
          class="com.emc.documentum.rest.error.http.GeneralExceptionMapping"/>
```

The extended exception mapping class adds more methods for the exception mapping with Spring annotation `@Exceptionhandler`. The output of the method must be an instance of `com.emc.documentum.rest.model.RestError`. Here is an example of the default error mapping for `ConversionFailedException`.

```
@ResponseBody
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(ConversionFailedException.class)
public RestError onConversionFailedException(ConversionFailedException e) {
 return buildRestError(new GenericExceptionConverter(e,
  HttpStatus.BAD_REQUEST, "E_INPUT_ILLEGAL_ARGUMENTS"));
}
```

Linking Custom and Core Resources

To make the custom REST services hypermedia driven, there needs to establish link relations between Core resources and custom resources. Documentum REST Services SDK provides an approach to add link relations to Core resources for the custom links. Please refer to the section Adding Links to Core Resources for the details.

Packaging and Deployment

For Maven users, the custom resource project can leverage the Maven war overlay plugin to repackage the REST war file by bundling both Core resources and custom resources into the single WAR file. Documentum REST Services SDK provides the sample Maven pom file in its Maven archetype project to illustrate how to build the custom resource WAR. The war overlay plugin configuration looks similar to the following.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <overlays>
                    <overlay>
                        <groupId>com.emc.documentum.rest</groupId>
                        <artifactId>documentum-rest-web</artifactId>
                        <excludes />
                    </overlay>
                </overlays>
            </configuration>
        </plugin>
    </plugins>
</build>
```

With the new custom resource WAR file, both Core resources and custom resources are running in the same server box and work seamlessly. In your development environment, you can enable `DEBUG` level in log4j for the package `emc.emc.documentum.rest`. All resource controller and view information will be printed out in the log file or on the server console.

Registering URI Templates

A RESTful style to make your custom resources discoverable is to add HATEOAS links (link relations) into the existing Core resources, including the Home Document, for the custom resources. Documentum REST Services allows you to register custom resource URI templates and then add them as link relations into resources. This section introduces the registration of URI templates.

The YAML file `rest-api-custom-resource-registry.yaml`, which is located in `dctm-rest.war/WEB-INF/classes/com/emc/documentum/rest/script`, provides a configuration entry uri-template-registry to register custom URI templates. You can modify this YAML file to add new URI templates.

The syntax for an URI template definition is shown below:

```
uri-template-registry:
- name:                         <template name>
  [href|hreftemplate]: <URI template pattern for 'href' or 'hreftemplate'>
  encoding:                   <encoding method>
  external:                     <true or false>
```

Example:

```
uri-template-registry:
- name: X_ALIAS_SET_RESOURCE_TEMPLATE
   href:  '{repositoryUri}/alias-sets/{aliasSetId}{ext}?owner={userName}'
   encoding: dual-url-encoding
- name: X_SEARCH_RESOURCE_TEMPLATE
   hreftemplate: '{baseUri}/x-search{?q,facet}'
- name: X_MODULE_RESOURCE_TEMPLATE
   href:  '{baseUri}/global-modules/{moduleId}{ext}'
```

Note: YAML is very strict with indentation and spaces. Incorrect indentation and redundant spaces may trigger errors. Tab characters (\t) are never allowed as indentation in YAML.

Alternatively, if you want to modify `rest-api-custom-resource-registry.yaml` before generating the WAR file, the file can be put into the web module path: `*custom-resource-web*/src/main/resources/com/emc/documentum/rest/script` and use the Maven overlay plugin to overwrite the default YAML file.

This configuration enables you to register new URI templates to the shared URI factory. The system uses these URI templates to resolve new links added on existing resources. A URI template consists of the following elements (key-value pairs):

Table 10. URI Template Elements

Key	Value
name *	Name of a custom URI template. The name must start with the prefix `X_`. We recommend that you use uppercase words delimited by underscore (_) to name a URI template. Example: `X_ALIAS_SET_RESOURCE_TEMPLATE`

Key	Value
[href|hreftemplate] *	URI template pattern for a link. The key is either href or hreftemplate. One and only one href or hreftemplate exists in a URI template. The URI template pattern can contain variables or query parameters enclosed in curly brackets `{}`. Example: `{repositoryUri}/<item1>/<item2>/<...>?<param1>&<param2>&<...>{ext}` For an href link, all variables in the URI pattern must be resolved on the server side, meaning that you must modify `value-mapping` according to the ADD LINKS ON EXISTING RESOURCES section for variables that are not predefined. By contrast, an `hreftemplate` link must contain variables that are treated as placeholders, such as `{?foo, bar}`, which does not need resolving on the server side. Predefined variables are resolved without additional settings in value-mapping. They are: • `{baseUri}`: URI root of the current deployment. Example: `{baseUri}` may refer to `https://current-deploy-host:8443/dctm-rest` • `{repositoryUri}`: URI root of the current repository. Example: `{repositoryUri}` may refer to `https://current-deploy-host:8443/dctm-rest/repositories/acme` • `{ext}`: URI extension for the representation format, such as .xml or .json.
encoding	Encoding method for path variables. Valid values are: • default • safe-text-encoding, which must be used when the property value to resolve a path variable contains special characters that may impact a URI, such as the slash character (`/`).

Key	Value
external	Specifies whether this URI template is external or internal. Valid values are: • true: Indicates this is an external URI template. External URI templates are not mapped to any custom resources. The external URI templates MUST NOT use pre-defined variables {repositoryUri} or {ext}. The accessibility of an external URI template will NOT be validated by the REST server. • false: Indicates this is an internal URI template. The internal URI template is mapped to a REST resource. The internal URI template MUST start with pre-defined variables {baseUri}, {repositoryUri}. The default value is false.
* Required	

The URI template name must be unique so that other resources to build the link relations can reference the template. In the code, the URI template can also be used to create an actual URL using the method `buildUriByTemplateName` of `com.emc.documentum.rest.http.UriFactory`. For example:

```
String moduleUri = uriFactory.buildUriByTemplateName(
    "X_MODULE_RESOURCE_TEMPLATE",
    Collections.singletonMap("moduleId", getDataInternal().getId()));
```

Registering Root Resources

A top-level custom resource may not have a relevant resource providing a link relation to discover it. For such resources, you must add it into the Home document with a designed link relation. That way, custom REST services comply with the hypermedia-driven design instead of providing hard-coded URIs to REST clients. The only resource URI the REST clients need to bookmark is the Home document resource whose URI path is `/services`. All root resources should be registered to the Home document with link relations.

In the REST WAR file, the YAML file `rest-api-custom-resource-registry.yaml` provides a configuration entry `root-service-registry` to register top-level resources to the Home document. You can modify this YAML file to add top-level resources.

The syntax for a root service registry is shown as follows:

```
root-service-registry:
- name:            <descriptive information of the resource>
  link-relation:   <the link relation name of the resource>
  uri-template:    <the URI template name registered for this service>
  allowed-methods: <allowed HTTP methods on this resource>
  media-types:     <supported media types on this resource>
```

Example:

```
root-service-registry:
- name:            User defined global search resource
  link-relation:   'x-search'
  uri-template:     X_SEARCH_RESOURCE_TEMPLATE
  allowed-methods: [POST]
  media-types:     [application/vnd.emc.documentum+json]
- name:            User defined custom alias set feed resource
  link-relation:   'x-alias'
  uri-template:    X_ALIAS_SETS_RESOURCE_TEMPLATE
  allowed-methods: [GET]
  media-types:     [application/vnd.emc.documentum+json, application/vnd.emc.documentum+xml]
```

This configuration enables you to register new resources to the Home Document. You can specify the following elements (key-value pairs):

Table 11. Root Service Registry

Key	Value
name	Descriptive information of a resource registered to the Home Document.
link-relation *	Link relation name of a registered resource, which enables you to locate the resource in the Home Document. One and only one link relation can be used to refer to a registered resource.

Key	Value
uri-template *	URI template registered for this resource. The URI template must exist in the REGISTER NEW URI TEMPLATES section. For resources registered to the Home Document, only the following variables are allowed in a specified URI template: • pre-defined variables `{baseUri}` and `{ext}` • variables that are treated as placeholders, such as query strings (hreftemplate only) The pre-defined variable `{repositoryUri}` and any variables resolved via value-mapping settings are not allowed in the URI template.
allowed-methods *	HTTP methods that can be performed on this resource, separated by a comma. Example: `[POST, GET]`
media-types *	Supported media types on this resource, delimited by comma. Example: `[application/vnd.emc.documentum+json, application/vnd.emc.documentum+xml]`
* Required	

Once a custom top-level resource is registered, the Core Home document resource will contain the additional entry for the top-level resource. Here is an example.

1. In the `uri-template-registry` section, add the URI template of the link relation by specifying these elements: `name`, `href/hreftemplate`, `encoding` (optional) and `external` (optional).

   Example:

   ```
   uri-template-registry:
   - name:           X_SEARCH_RESOURCE_TEMPLATE
     hreftemplate:   '{baseUri}/x-search{?q,facet}'
     encoding:       dual-url-encoding
     external:       false
   ```

   For link relations added to the Home Document, only pre-defined variables and variables used as placeholders can appear in the URI templates. For example, the following URI template is not eligible for link relations added to the Home Document:

   Example:

   ```
   uri-template-registry:
   - name:              X_ALIAS_SET_RESOURCE_TEMPLATE
     hreftemplate:      '{repositoryUri}/alias-sets/{aliasSetId}{ext}?owner={userName}'
     encoding:          dual-url-encoding
     external:          false
   ```

2. In the root-service-registry section, specify the following elements for the link relation: `name`, `link-relation`, `uri-template`, `allowed-method`, and `media-types`.

   Example:

```
root-service-registry:
- name:               User defined global search resource
  link-relation:      'x-search'
  uri-template:       X_SEARCH_RESOURCE_TEMPLATE
  allowed-methods:    [POST]
  media-types:        [application/vnd.emc.documentum+json, application/vnd.emc.documentum+xml]
```

3. Deploy the custom resources WAR file after the YAML is updated. The following sample of Home Document has the custom link relation `x-search` added.

```
<?xml version="1.0" encoding="UTF-8"?>
   <resources xmlns="http://identifiers.emc.com/vocab/documentum"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <!-- core top resource -->
     <resource rel="http://identifiers.emc.com/linkrel/repositories">
       <link href="http://localhost:8080/acme-rest/repositories"/>
       <hints>
         <allow><i>GET</i></allow>
         <representations>
           <i>application/xml</i>
           <i>application/json</i>
           <i>application/atom+xml</i>
           <i>application/vnd.emc.documentum+json</i>
         </representations>
       </hints>
     </resource>
     <!-- core top resource -->
     <resource rel="about">
       <link href="http://localhost:8080/acme-rest/product-info"/>
       <hints>
         <allow><i>GET</i></allow>
         <representations><i>application/xml</i>
           <i>application/json</i>
           <i>application/vnd.emc.documentum+xml</i>
           <i>application/vnd.emc.documentum+json</i>
         </representations>
       </hints>
     </resource>
     <!-- custom top resource -->
     <resource rel="x-search">
       <link hreftemplate="http://localhost:8080/acme-rest/x-search{?q,facet}"/>
       <hints>
         <allow><i>POST</i></allow>
         <representations>
           <i>application/vnd.emc.documentum+json</i>
           <i>application/vnd.emc.documentum+json</i>
         </representations>
       </hints>
     </resource>
   </resources>
```

Adding Links to Core Resources

To make a custom resource discoverable via Core resources, you need to add new link relations pointing to the custom resources to one or more Core resources.

In the REST WAR file, the YAML file `rest-api-custom-resource-registry.yaml` provides a configuration entry `resource-link-registry` to add link relations to Core resources. You can modify this YAML file to add root resources.

The syntax for a link relation registry is shown as follows:

```
resource-link-registry:
- resource:          <the resource code name>
  link-relation:     <the new link relation name>
  uri-template:      <the name of the URI template defined in uri-template-registry>
  value-mapping:     <the property names from the resource to resolve the variable
                     values in the URI template>
```

Example:

```
resource-link-registry:
- resource:          object
  link-relation:     'http://identifiers.emc.com/linkrel/acl'
  uri-template:      X_ACL_RESOURCE_TEMPLATE
  value-mapping:     [objectId:r_object_id]
- resource:          folder
  link-relation:     'x-folder-attachment'
  uri-template:      X_ATTACHMENT_RESOURCE_TEMPLATE
  value-mapping:     []
```

This configuration allows you to register new links to Core resources. You can specify the following elements (key-value pairs):

Table 12. Resource Link Registry

Key	Value
resource *	The code name of the resource to which the links are added. Each Core resource has a unique code name. In the link registry, following resources are supported: [format, user, group, content, relation, type, object, document, folder, cabinet, network-location, relation-type, repository]. See Appendix C for all code names of Core resources.
link-relation *	Link relation name of the registered resource, which enables you to locate the resource in the existing resource. One and only one link relation can be used to refer to a registered resource.

Key	Value
uri-template *	The name of the URI template registered in `uri-template -registry`. The URI template can contain variables which need to be resolved during the generation of the link relation `href`.
value-mapping	Value mappings are used to resolve values of path variables or query parameters on URI templates defined in uri-template. A value mapping follows the pattern: [*variableName1*:*propertyName1*[,*variableName2*:*propertyName2*]* *variableName* represents a variable specified in a URI template. There cannot be any duplicated variables in a URI template. *propertyName* represents a property in the resource. Note that properties of the Repository resource (such as `Id`, `name`, and `description`) cannot be part of a path variable. When the URI is generated at runtime, the variables will be replaced by the property values; the property can be repeating. When any of the properties in the variables is repeating, multiple links will be generated, each with a different value in the variable segment. Besides, a 'title' attribute will be set on the link representation to differentiate href for different repeating values. There could be at most one repeating property in the variables. If the property values for the variables are not returned in the resource object (e.g. by custom view), the links will neither be generated nor be presented on the resource.
* Required	

Once the custom root resource is registered, the Core home document resource will contain the additional entry for the root resource. Here is an example.

1. In the `uri-template-registry` section, add the URI template of the link relation by specifying these elements: `name`, `href/hreftemplate`, `encoding` (optional) and `external` (optional).
   Example:

   ```
   uri-template-registry:
   - name:     X_ACL_RESOURCE_TEMPLATE
     href:     '{repositoryUri}/objects/{objectId}/acl{ext}'
   ```

2. In the `resource-link-registry` section, specify the following elements for the link relation: `resource`, `link-relation`, `uri-template`, and `value-mapping`.**Example:**

   ```
   resource-link-registry:
   - source:         object
     link-relation:  'http://identifiers.emc.com/linkrel/acl'
     uri-template:   X_ACL_RESOURCE_TEMPLATE
     value-mapping:  [objectId:r_object_id]
   ```

3. Deploy the custom resources WAR file after the YAML is updated. The following sample of Home Document has the custom link relation http://identifiers.emc.com/linkrel/acl is added.
   Example:

   ```
   <object xmlns="http://identifiers.emc.com/vocab/documentum"
   ```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="dm_cabinet"
definition="http://localhost:8080/acme-rest/repositories/
REPO/types/dm_cabinet">
<properties  xsi:type="dm_cabinet-properties">
<object_name>a;x.f</object_name>
...
<r_object_id>0c0020808000dfb6</r_object_id>
</properties>
<links>
<!-- core link relations  -->
<link rel="self"
href="http://localhost:8080/acme-rest/repositories/REPO/objects/
0c0020808000dfb6"/>
..
<link rel="http://identifiers.emc.com/linkrel/relations"
href="http://localhost:8080/acme-rest/repositories/REPO/relations?
related-object-id=0c0020808000dfb6&related-object-role=any"/>
<!--  customer defined link relations  -->
<link rel="http://identifiers.emc.com/linkrel/acl"
href="http://localhost:8080/acme-rest/repositories/REPO/objects/
0c0020808000dfb6/acl"/>
</links>
</object>
```

Creating Custom Message Files

Documentum REST Services allows you to create custom message files. You can read custom messages with `MessageBundle` as shown in the following code snippet.

```
public String feedTitle() {
 return MessageBundle.INSTANCE.get("TITLE_OF_MYRESOURCE_COLLECTION");
}
```

The name of a custom message file must follow this pattern:

`rest-*messages*.properties`
`*` stands for any string of characters.Example: `rest-myresource-messages-new.properties`

Additionally, the package where you create custom message files must be listed in the `rest.extension.message.packages` runtime property.

Tutorial: Documentum REST Services Extensibility Development

This tutorial walks through a reference implementation of custom resources using the Documentum REST extensibility feature. It's a showcase for bringing various Core REST and Spring framework eco-system technologies together to implement REST resources for your domain types. You'll learn these technologies from this tutorial:

- Design and implement new resources
- Add new links to Core resources
- Package custom and Core resources into a single WAR package

What to Build First

Assume that you need to create an alias set collection resource for the `dm_alias_set` type objects in the repository. The resource is feed-based and accepts HTTP GET method. The alias set collection resource is designed as follows:

Table 13. Resource design of the alias set collection

Resource	URI	HTTP Methods	Media Types
Alias Set Collection	/repositories /{repositoryName}/alias -sets{?view,filter,links,inline, page,items-per -page,include-total,sort}	GET	application/atom+xml application/vnd.emc .documentum+json

The resource supports both XML and JSON representations in conformity with Core REST representations.

XML representation for the alias set collection resource

```
<feed xmlns="http://www.w3.org/2005/Atom"
xmlns:dm="http://identifiers.emc.com/vocab/documentum"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <id>http://localhost:8080/acme-rest/repositories/REPO/alias-sets</id>
  <title>Alias Sets</title>
  ...
  <link rel="self" href="http://localhost:8080/acme-rest/repositories/
  REPO/alias-sets"/>
  <entry>
    <id>http://localhost:8080/acme-rest/repositories/REPO/alias-sets/
    6600208080000105</id>
    <title>AdminAccess</title>
    ...
    <content type="application/xml"
    src="http://localhost:8080/acme-rest/repositories/REPO/alias-sets/
    6600208080000105"/>
    <link rel="edit" href="http://localhost:8080/acme-rest/repositories/
    REPO/alias-sets/6600208080000105"/>
  </entry>
  ...
```

JSON representation for the alias set collection resource

```
{
  id: "http://localhost:8080/acme-rest/repositories/REPO/alias-sets",
  title: "Alias Sets",
  ...
  links: [{rel: "self", href: "http://localhost:8080/acme-rest/
  repositories/REPO/alias-sets.json"}],
  entries: [
    {
      id: "http://localhost:8080/acme-rest/repositories/REPO/alias-sets/
      6600208080000105",
      title: "AdminAccess",
      ...
      content: {
        type: "application/vnd.emc.documentum+json",
        src: "http://localhost:8080/acme-rest/repositories/REPO/alias-sets/
        6600208080000105"
      },
      links: [{rel: "edit", href: "http://localhost:8080/acme-rest/
      repositories/REPO/alias-sets/6600208080000105"}]
  },
  ...
```

Note: XML representation for collections must conform to the atom feed; and JSON representation for collections must conform to EDAA.

With the alias set collection resource in your mind, it's quite natural to think about building single alias set resources. The single alias set resource can be embedded in the feed representation of the alias set collection resource. The alias set resource is designed as follows:

Table 14. Resource design of the alias set resource

Resource	URI	HTTP Methods	Media Types
Alias Set	/repositories /{repositoryName}/alias-sets /{aliasSetId}{?view,filter,links }	GET	application/vnd.emc .docuemntum+xmlapplication /vnd.emc .documentum+json

Just like the alias set collection, single alias set resources support both XML and JSON representations.

XML representation for the alias set resource

```
<?xml version='1.0' encoding='UTF-8'?>
<alias-set xmlns="http://identifiers.emc.com/vocab/documentum"
xmlns:dm="http://identifiers.emc.com/vocab/documentum"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="dm_alias_set" definition="http://localhost:8080/acme-rest/
repositories/REPO/types/dm_alias_set">
  <properties>
    <owner_name>Administrator</owner_name>
    <object_name>AdminAccess</object_name>
    ...
  </properties>
  <links>
    <link rel="self" href="http://localhost:8080/acme-rest/repositories/
    REPO/alias-sets/6600208080000105"/>
    <link rel="author" href="http://localhost:8080/acme-rest/repositories/
    REPO/users/Administrator.xml"/>
  </links>
</alias-set>
```

JSON representation for the alias set resource

```
{
  "name":"alias-set",
  "type":"dm_alias_set",
  "definition":"http://localhost:8080/acme-rest/repositories/REPO/
   types/dm_alias_set",
  "properties":{
     "owner_name":"Administrator",
     "object_name":"AdminAccess",
     ...
   },
  "links":[
     {"rel":"self","href":"http://localhost:8080/acme-rest/repositories/
     REPO/alias-sets/6600208080000105"},
     {"rel":"author","href":"http://localhost:8080/acme-rest/repositories/
     REPO/users/Administrator"}
  ]
}
```

Quickstart

Documentum REST Services SDK provides you with a convenient way to set up your first custom resource project: using the Maven archetype. Refer to Get Started With the Development Kit for details. The code structure we will build for both resources is shown as follows.

Figure 9. Code Structure

As the diagram shows, we create four modules for the project. The model module creates the serializable alias set data model. The persistence module creates the persistence API to get and update the alias set object using DFC. The resource module creates the resource controllers. And the web module creates the WAR file for the entire custom services.

Creating Resource Model

The data model class defines the data structure of the resource representation for both input and output. With Documentum REST annotations, the model Java class has a direct mapping to the resource representation messages.

The Maven archetype project contains a sample model class AliasSet for the alias set. This is the only class added to the model module.

Figure 10. Structure of Model

The class definition for AliasSet is shown below.

```
/**
 * Data model for dm_alias_set
 */
@SerializableType(value = "alias-set",
xmlNSPrefix = "dm",
xmlNS = "http://identifiers.emc.com/vocab/documentum")
public class AliasSet extends PersistentObject {
    public AliasSet() {
        setType("dm_alias_set");
    }
}
```

`@SerializableType` is the Documentum REST Java annotation introduced to design the representation model. And the `PersistentObject` class is the out-of-box model class that implements properties and links, and allows inheritance. So with this simple class definition, the alias set representation model is built up.

Documentum REST Marshalling Framework provides more information about the Documentum REST annotations.

Validating the annotated resource model

Documentum REST Services SDK provides a Maven plugin to validate the REST model annotations during the design time. The plugin can be added to the pom file to validate the models during the build process. You can also run the scanner with command lines. A sample of the Maven plugin configuration in the pom file is shown below.

```
<plugin>
    <groupId>com.emc.documentum.rest</groupId>
    <artifactId>documentum-rest-extension-validating</artifactId>
    <version>1.0</version>
    <inherited>true</inherited>
    <executions>
        <execution>
            <id>validate-annotation</id>
            <phase>verify</phase>
            <goals>
                <goal>check-annotation</goal>
            </goals>
            <configuration>
                <input>${project.basedir}/target/${scan.artifactId}-${version}.war</input>
                <outputDir>${project.basedir}/target</outputDir>
                <isDebug>false</isDebug>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The section Annotation Scanner provides more information about Documentum REST annotation validation.

Creating Persistence

The persistence module defines the DFC interface and implementation for the operations. This module accepts the data model input. Output of this module is also a data model. Documentum REST Services SDK library provides a lot of out-of-box persistence APIs that can be used to communicate with DFC. In this sample project, you can fully leverage Core persistence library to manipulate the alias set object in the repository.

The Maven archetype project only has one class `AliasSetCollectionQueryTemplate`, which customizes the DQL query for the alias set collection.

Figure 11. Structure of Persistence

This query template is the only implementation class to retrieve a collection of alias set objects. The SDK provides other APIs to facilitate the object collection retrieval. The class definition of the `AliasSetCollectionQueryTemplate` is shown as follows.

```
/**
 * A query template to get alias set collection.
 */
public class AliasSetCollectionQueryTemplate extends PagedQueryTemplate {
    @Override
    protected List<String> defaultFields() {
        return Arrays.asList( "r_object_id", "alias_category",
        "alias_name", "alias_value", "object_name", "owner_name");
    }

    @Override
    protected String qualification() { return ""; }

    @Override
    protected String from() { return "dm_alias_set"; }

    @Override
    protected List<SortOrder> defaultSorts()
    { return Arrays.asList(new SortOrder("object_name", true)); }

    @Override
    protected boolean supportRepeatingAttributeQuery() {  return true; }
}
```

`PagedQueryTemplate` is an out-of-the-box class that defines the abstraction for the DQL query string construction for a type-specific collection. A collection resource’s persistence API can extend this class to build the DQL query string for the returning collection. The next section shows an example about how it is used in the resource controller.

Creating Resource Controller

The resource controller defines the REST resource mapping for the custom resource and is the end point for a resource operation.

The Maven archetype project defines two controllers, for the alias set collection resource and the alias set resource, respectively. The code structure of the resource module is shown as follows.

Figure 12. Structure of Controller

AliasSetCollectionController

The `AliasSetCollectionController` class is the definition for the alias set collection resource.

```
/**
 * Collection of alias sets in a repository.
 */
@Controller("acme#alias-sets")
@RequestMapping("/repositories/{repositoryName}/alias-sets")
@ResourceViewBinding(AliasSetsFeedView.class)
public class AliasSetCollectionController extends AbstractController {
    @RequestMapping(
            method = RequestMethod.GET,
            produces = {
            SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
            MediaType.APPLICATION_ATOM_XML_VALUE,
            MediaType.APPLICATION_JSON_VALUE,
            MediaType.APPLICATION_XML_VALUE
    })
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public AtomFeed getAliasSets(
            @PathVariable("repositoryName") final String repositoryName,
            @TypedParam final CollectionParam param,
            @RequestUri final UriInfo uriInfo)
            throws Exception {
        PagedQueryTemplate template = new AliasSetCollectionQueryTemplate()
                .filter(param.getFilterQualification())
                .order(param.getSortSpec().get())
                .page(param.getPagingParam().getPage(),
                param.getPagingParam().getItemsPerPage());
        PagedDataRetriever<AliasSet> dataRetriever =
        new PersistentDataRetriever<AliasSet>(
                template,
                param.getPagingParam().getPage(),
                param.getPagingParam().getItemsPerPage(),
                param.getPagingParam().isIncludeTotal(),
                param.getAttributeView(),
                AliasSet.class);
        return getRenderedPage(
                repositoryName,
                dataRetriever.get(),
                param.isLinks(),
                param.isInline(),
                uriInfo,
                null);
    }
}
```

The resource controller uses a number of Java annotations. They are mainly divided into two categories.

Spring annotations

- `@Controller` - a Spring annotation to define a resource controller. It is strongly recommended to assign a unique name for the controller as Core REST runtime uses this name to identify a resource.
- `@RequestMapping` - a Spring annotation to define the URI, headers, and parameters for a resource. This annotation can be applied to the controller class level or individual controller method level. Spring has a complicated match pattern to compare whether two controller methods or controller classes use the same URI mapping. For example, you can define two controller methods with the same URI, but with different HTTP methods; even that you can define two controller methods with the same URI and HTTP methods, but with different HTTP headers or query parameters.
- `@ResponseBody` & `@ResponseStatus` – Spring annotations to automatically resolve the model&view and HTTP status for the resource. `@ResponseBody` is mandatory for custom resource development.
- `@PathVariable` – a Spring annotation to extract variables from a path.

Documentum REST Services annotations

- `@TypedParam final CollectionParam param` - a Documentum REST Services annotation customizing Spring query parameters which assemble feed resource-related query parameters together, e.g., inline, page, items-per-page.
- `@RequestUri` - a Documentum REST annotation customizing Spring query parameters which assembles feed resource related query parameters together, e.g., inline, page, items-per-page.
- `@ResourceViewBinding` - a Documentum REST Services annotation to bind a resource to a default view. With the view binding, the correct view class is instantiated to resolve the links and atom attributes for the alias set resource representation. This annotation can be put on the controller methods, too.

The Documentum REST Services SDK also provides abstraction, model, and utility classes for code reuse in custom resource development.

Documentum REST classes

- `AbstractController` - the Core controller abstraction. All custom resources should extend this abstract class.
- `PagedDataRetriever` - the Core persistence API to execute a query for a paged query template and return a page for the query.
- `AtomFeed` - the Core model class for Atom feeds. All collection resources are returned as an atom feed.

In summary, there aren't many lines to implement an alias set collection resource, but each piece provides very useful information.

AliasSetController

Let's move on to the controller for the alias set resource. The class definition is shown as follows.

```
/**
 * An alias set.
 */
```

```
@Controller("acme#alias-set")
@RequestMapping("/repositories/{repositoryName}/alias-sets/{aliasSetId}")
@ResourceViewBinding(AliasSetView.class, queryTypes="dm_alias_type")
public class AliasSetController extends AbstractController {
    @Autowired
    private SysObjectManager sysObjectManager;

    @RequestMapping(
            value = ALIAS_SET_URI_PATTERN,
            method = RequestMethod.GET, produces = {
            SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
            MediaType.APPLICATION_ATOM_XML_VALUE,
            MediaType.APPLICATION_JSON_VALUE,
            MediaType.APPLICATION_XML_VALUE
    })
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public AliasSet getAliasSet(
            @PathVariable("repositoryName") final String repositoryName,
            @PathVariable("aliasSetId") final String aliasSetId,
            @TypedParam final SingleParam param,
            @RequestUri final UriInfo uriInfo)
            throws Exception {
        AliasSet aliasSet = sysObjectManager.getObjectByQualification(
                String.format("dm_alias_set where r_object_id='%s'", aliasSetId),
                param.getAttributeView(),
                AliasSet.class
        );
        return getRenderedObject(repositoryName, aliasSet, param.isLinks(),
        uriInfo, null);
    }
}
```

The controller is implemented similarly to the `AliasSetCollectionController`. It calls `SysObjectManager` to manipulate the sysobjects. Since an alias set is nothing special other than a sysobject, you can reuse existing Core REST APIs to operate the alias set.

mvc-custom-context.xml

In addition to the resource controller definition, you need to define the Spring configuration to scan the controller classes during the server startup. This Spring configuration file tells Spring framework where to load the controllers. A sample is shown below.

```
<context:component-scan base-package="org.acme"
use-default-filters="false">
   <context:include-filter type="regex" expression=".*controller.*Controller"/>
   <context:exclude-filter type="custom"
   expression="com.emc.documentum.rest.context.ResourceExcludeFilter"/>
</context:component-scan>
```

Creating Resource View

Till now, the resource implementation has been able to return the alias set feed and alias set resource following the URIs, but the link relations and atom attributes for them are still missing. In this section, we are going to create views for both resources to customize the links and atom attributes.

AliasSetsFeedView

The `AliasSetsFeedView` class defines the atom feed attributes and links for the alias set collection resource.

View definition of AliasSets

```
/**
 * View for the alias set feed.
 */
@FeedViewBinding(AliasSetView.class)
public class AliasSetsFeedView extends FeedableView<AliasSet> {
    public AliasSetsFeedView(Page<AliasSet>
    page, UriInfo uriInfo, String repositoryName,
    Boolean returnLinks, Map<String, Object> others) {
        super(page, uriInfo, repositoryName, returnLinks, others);
    }

    @Override
    public String feedTitle() { return "Alias Sets"; }

    @Override
    public Date feedUpdated() { return new Date(); }
}
```

The feed view class must extend `FeedableView<T>` and implement its own feed attributes.

`FeedableView<T>` - the Core REST feed view abstraction. It provides built-in link relations for an atom feed. By implementing this view, the alias set collection resource obtains the built-in `self` link and pagination links. There are also default implementations for several atom feed attributes, like atom authors, atom ID, and so on.

`@FeedViewBinding` - a Core REST annotation to bind a feed view to the corresponding entry views.

AliasSetView

The AliasSetView class defines the atom entry attributes and links for the alias set.

View definition of AliasSet

```
/**
 * View for the alias set.
 */
@DataViewBinding(AliasSet.class)
public class AliasSetView extends PersistentDataView<AliasSet> {
    public AliasSetView(AliasSet aliasSet, UriInfo uriInfo, String repositoryName,
    Boolean returnLinks, Map<String, Object> others) {
        super(aliasSet, uriInfo, repositoryName, returnLinks, others);
    }

    @Override
    public String entryTitle() { return (String) serializableData.
    getMandatoryAttribute("object_name"); }

    @Override
    public String entrySummary() { return (String) serializableData.
    getMandatoryAttribute("object_name"); }

    @Override
    public Date entryUpdated() { return new Date(); }

    @Override
    public Date entryPublished() { return new Date(); }

    @Override
    public List<AtomAuthor> entryAuthors() {
        String owner = (String)getDataInternal().
        getMandatoryAttribute("owner_name");
        String owerLink = getUriFactory().userUri((String)
        serializableData.getAttributeByName("owner_name"), null);
        return Arrays.asList(new AtomAuthor(owner, owerLink, null));
    }

    @Override
    public AliasSet entryContent() { return data(); }

    @Override
    public String entrySrc() { return canonicalResourceUri(boolean validate); }

    @Override
    public void customize() {
        String owerLink = getUriFactory().userUri((String)
        serializableData.getAttributeByName("owner_name"), null);
        makeLinkIf(getDataInternal().getAttributeByName("owner_name") != null,
                   LinkRelation.AUTHOR.rel(), owerLink);

    }

    @Override
    public String canonicalResourceUri(boolean validate) {
        return getUriFactory(validate).buildUriByTemplateName(
                "X_ALIAS_SET_URI_TEMPLATE",
                Collections.singletonMap("aliasSetId", serializableData.getId()));
    }
}
```

The domain model view must extend `EntryableView<T>` and implement its own entry attributes and resource links. In this sample class, `AliasSetView` extends `PersistentDataView` which provides the default implementation for making links. When an object is accessible through multiple resource URIs, the canonical resource URI represents the primary resource URI for this object. The method `canonicalResourceUri` returns the canonical resource URI dynamically with a validation option.

- When `validate` is `true` and the corresponding resource for the URI is inactive, the returned URI is `{@link com.emc.documentum.rest.http.UriFactory#INACTIVE_URL}`.
- When `validate` is `false`, the returned URI is a static URI defined by this method.

`@DataViewBinding` - a Core REST annotation to mark on a singe data object resource view. This view binding binds a view to a resource model.

Refer to the section Documentum REST MVC for more information about the Documentum REST MVC programming pattern.

Adding More HTTP Methods

The Maven archetype project only demonstrates the GET method on the alias set collection resource and the alias set resource. Obviously in a real application, object creation, modification, and deletion are needed, too. This section walks through the design and the implementation of object creation and deletion.

Creating an Alias Set Object

A good practice to create a new object is to POST the new resource to the corresponding collection resource. As you have the alias set collection resource, let's create alias set resources with the HTTP POST method on the alias set collection resource.

Table 15. Resource design of the alias set resource

Resource	URI	HTTP Methods	Media Types
Alias Set	/repositories /{repositoryName}/alias -sets{?view,filter,links,inline,page,items -per-page,include-total,sort}	GET	application /atom+xmlapplication /vnd.emc .documentum+json
		POST	application/vnd.emc .documentum+xmlapplication /vnd.emc .documentum+json

The new controller method is defined as follows.

```
@Controller("acme#alias-sets")
@RequestMapping("/repositories/{repositoryName}/alias-sets")
@ResourceViewBinding(AliasSetsFeedView.class)
public class AliasSetCollectionController extends AbstractController {
    ...
    @RequestMapping(method = RequestMethod.POST,
         produces = {
            SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
            SupportedMediaTypes.APPLICATION_VND_DCTM_XML_STRING,
            MediaType.APPLICATION_JSON_VALUE,
            MediaType.APPLICATION_XML_VALUE
    })
    @ResponseBody
    @ResponseStatus(HttpStatus.CREATED)
    @ResourceViewBinding(AliasSetView.class)
    public AliasSet createAliasSet(
            @PathVariable("repositoryName") final String repositoryName,
            @RequestBody final AliasSet aliasSet,
            @RequestUri final UriInfo uriInfo)
            throws DfException {
        AliasSet createdRelation = sysObjectManager.
        createObject(aliasSet, false, AttributeView.ALL);
        Map<String, Object> others = new HashMap<String, Object>();
        others.put(ViewParams.POST_FROM_COLLECTION, true);
        return getRenderedObject(repositoryName, createdRelation,
        true, uriInfo, others);
    }
    @Autowired
    private SysObjectManager sysObjectManager;
}
```

Note:

- A `ViewParams.POST_FROM_COLLECTION` parameter is needed on the view parameters for any controller create method.
- With the annotation `@ResourceViewBinding` on the controller method, the same `AliasSetView` is reused to render alias set resource responses.
- A create operation response must have a Location header pointing to the new resource URI. Documentum REST Services adds the Location header automatically for controller methods in case following conditions are met:
  - The method returns a `Linkable` object.
  - The method has the annotation `@ResponseBody`.
  - The method has the annotation `@ResponseStatus(HttpStatus.CREATED).`

Deleting an Alias Set Object

It is nature to perform an HTTP DELETE method on the alias set resource to delete an alias set. A delete method `removeAliasSet` is defined on `AliasSetController` as shown in the follow sample.

```
@Controller("acme#alias-set")
@RequestMapping("/repositories/{repositoryName}/alias-sets/{aliasSetId}")
@ResourceViewBinding(AliasSetView.class)
public class AliasSetController extends AbstractController {
    ...
    @RequestMapping(method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void removeAliasSet(
            @PathVariable("repositoryName") final String repositoryName,
            @PathVariable("aliasSetId") final String aliasSetId,
            @RequestUri final UriInfo uriInfo)
            throws Exception {
        sysObjectManager.deleteSysObject(aliasSetId, true, true,
        DeleteVersionPolicy.ALL);
    }
}
```

Making Resources Queryable

Now the alias set resource is implemented with the `dm_alias_set` type. But how to make Core DQL resource to return resource links for the query result of `dm_alias_set` type? The answer is to add a new attribute to the `@ResourceViewBinding` annotation on the `AliasSetController` class.

```
@Controller("acme#alias-set")
```

```
@RequestMapping("/repositories/{repositoryName}/alias-sets/{aliasSetId}")
@ResourceViewBinding(value = AliasSetView.class, queryTypes = "dm_alias_set")
public class AliasSetController extends AbstractController {
…
}
```

`queryTypes` makes the alias set resource linkable from DQL query result. Here is an example.

```
<?xml version='1.0' encoding='UTF-8'?>
<feed xmlns="http://www.w3.org/2005/Atom"
xmlns:dm="http://identifiers.emc.com/vocab/documentum"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  ...
  <link rel="self" href="http://localhost:8080/acme-rest/repositories/
  REPO?dql=select%20*%20from%20dm_alias_set"/>
  <entry>
    <id>http://localhost:8080/acme-rest/repositories/REPO?
    dql=select%20*%20from%20dm_alias_set&index=0</id>
    ...
    <content>
      <dm:query-result definition="http://localhost:8080/acme-rest/
      repositories/REPO/types/dm_alias_set">
        <dm:properties>
          <dm:r_object_id>6600208080000100</dm:r_object_id>
          <dm:owner_name>Administrator</dm:owner_name>
          <dm:object_name>Smart Container</dm:object_name>
          <dm:object_description></dm:object_description>
        </dm:properties>
      </dm:query-result>
    </content>
    <link rel="edit" href="http://localhost:8080/acme-rest/repositories/
    REPO/alias-sets/6600208080000100"/>
  </entry>
  ...
```

Making Resources Linkable

With implementation above, the alias set collection resource and the alias set resource are almost finished. But wait – how does a REST client discover the links for the new resources at runtime? The solution is to customize Core resources to add links for your new resources. Here are the links we want to design.

- The alias set collection resource can be found on a repository resource.
- The alias set resource can be found from the feed of alias set collection resource.

For the second one, as we have implemented the Location header on alias set creation operation, the POST on the alias set collection resource has been able to return the link for the newly created alias set resource. So what you actually need is to customize the link relations for the Core repository resource. To do this, simply add a YAML file under the right path and then modify the YAML file as shown below.

Figure 13. YAML File Location

Adding a link relation in repository for the alias set collection:

```
resource-link-registry:
  - resource:      repository
    link-relation: 'http://identifiers.emc.com/linkrel/alias'
    uri-template:  X_ALIAS_SETS_URI_TEMPLATE
    value-mapping: []
```

The section Adding Links to Core Resources has all the details of this function. Here is the sample.

By doing this, the repository resource now has one more link relation that points to the alias set collection resource.

```
<repository xmlns="http://identifiers.emc.com/vocab/documentum"
xmlns:dm="http://identifiers.emc.com/vocab/documentum"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <id>8320</id>
 <name>REPO</name>
 <description/>
 <server>
  <name>REPO</name>
  <host>CS71P01</host>
  <version xml:space="preserve">7.1.0010.0158  Win64.SQLServer</version>
  <docbroker>CS71P01</docbroker>
 </server>
 <links>
  <link rel="self" href="http://localhost:8080/acme-rest/repositories/REPO"/>
  <link rel="http://identifiers.emc.com/linkrel/alias"
    href="http://localhost:8080/acme-rest/repositories/REPO/alias-sets"/>
  ...
 </links>
</repository>
```

Making Resources Non-Batchable (Optional)

By default, all custom resources are batch-able. If you want to make a specific custom resource or a certain method non-batchable, put the `@BatchProhibition` annotation on the resource controller or the controller method.

Similarly, you can use the `@TransactionProhibition` annotation to remove the transaction support from a custom resource. Like the `@BatchProhibition` annotation, `@TransactionProhibition` can be applied to both the controller class level and the controller method level.

This tutorial does not demonstrate how to manage the batch property of a custom resource as we want the Alias Set and Alias Set collection resources to stay batchable. You can find more details on making them non-batchable from Resource: Deciding Whether to be Batchable.

Packaging and Deploying Resources

Finally, we need to package the new resources into a WAR file. This can be achieved by the Maven WAR overlay plugin in the web module. The YAML file, which customizes resources, must be put into the class-path of the web module `com/emc/documentum/rest/script`.

Here is the WAR overlay plugin configuration sample in the web module.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.4</version>
            <configuration>
                <overlays>
                    <overlay>
                        <groupId>com.emc.documentum.rest</groupId>
                        <artifactId>documentum-rest-web</artifactId>
                        <excludes />
                    </overlay>
                </overlays>
            </configuration>
        </plugin>
    </plugins>
</build>
```

A single acme-rest-web-0.0.1-SNAPSHOT.war is built under in the directory `/acme-rest-web/target`, which contains both Core resources and the new resources.

Troubleshooting

During the development, you can enable the `DEBUG` level log4j logging for the package `com.emc.documentum.rest`. Resource controller and view information about both core and custom resources will be printed.

Appendix A

Link Relations

Documentum REST Services uses the following link relations.

Table 16. Public Link Relations

Link Relation	Description	Specification
about	Returns product information	http://tools.ietf.org/html/rfc6903
canonical	Designates an Internationalized Resource Identifier (IRI) as preferred over resources with duplicative content.	http://tools.ietf.org/html/rfc6596
child	Points to a hierarchical child, or subobject, of the current object.	http://www.w3.org/MarkUp/draft-ietf-html-relrev-00.txt
collection	Points to a resource which represents the collection resource for the context IRI.	http://tools.ietf.org/html/rfc6573
contents	Points to the contents metadata feed for a contentful object.	http://www.w3.org/TR/1999/REC-html401-19991224
down	Points to a resource where child entries of an entry may be found.	http://tools.ietf.org/html/draft-divilly-atom-hierarchy-03
edit	Points to a resource that can be used to edit the link's context.	http://tools.ietf.org/html/rfc5023
enclosure	Identifies a related resource that is potentially large and might require special handling.	http://bitworking.org/projects/atom/rfc5023.html#new-link-relation
first	An IRI that refers to the furthest preceding resource in a series of resources.	http://tools.ietf.org/html/rfc5005
icon	Points to the icon for an entry, a page or a site.	http://www.w3.org/TR/html5/links.html#rel-icon

Link Relation	**Description**	**Specification**
last	An IRI that refers to the furthest following resource in a series of resources.	http://tools.ietf.org/html/rfc5005
next	Indicates that the link's context is a part of a series, and that the next in the series is the link target.	http://tools.ietf.org/html/rfc5005
parent	Refers to one of the following: • parent type of a type • parent folder of a sysobject • parent document of a document • parent object of a relation • parent group of a group, a role, or a user.	http://www.w3.org/MarkUp/draft-ietf-html-relrev-00.txt
predecessor-version	Points to a resource containing the predecessor version in the version history.	http://tools.ietf.org/html/rfc5829
previous	Points to the previous resource in an ordered series of resources.	http://tools.ietf.org/html/rfc5005
related	Points to the feed for a sysobject related with specified relation type(s).	http://tools.ietf.org/html/rfc4287
self	Conveys an identifier for the link's context.	http://www.ietf.org/rfc/rfc4287.txt
version-history	Points to a resource containing the version history for the context.	http://tools.ietf.org/html/rfc5829

Table 17. Link Relations Introduced in Documentum REST Services

Link Relation	Description
batches	Points to the batches resource under a repository.
batch-capabilities	Points to the batch capabilities resource under a repository.
cabinets	Points to the cabinets feed under a repository.
cancel-checkout	Points to the cancel-checkout behavior for a versionable object.

checkin-branch	Points to the checkin as branch version behavior for an object that is checked out.
checkin-next-major	Points to the checkin as next major version behavior for an object that is checked out.
checkin-next-minor	Points to the checkin as next minor version behavior for an object that is checked out.
checkout	Points to the checkout behavior for a versionable object .
child-links	Points to the child links feed resource of a folder.
content-media	Indicates that the content can be downloaded.
current-user	Points to the current login user resource under a repository.
current-version	Points to the current version resource for a versionable object.
default-folder	Points to the default folder resource for a user.
delete	Points to the delete behavior of an object.
documents	Points to the documents feed under a repository.
folders	Points to the folders feed under a repository.
format	Points to the format resource for a content resource.
formats	Points to the formats feed under a repository.
groups	Points to the groups feed under a repository or direct member groups under a group.
objects	lists folder contents.
parent-links	Points to the parent links resource of a non-cabinet sysobject.
primary-content	Points to the primary content resource for a contentful sysobject.
relate	Indicates that the object can be related to other objects.
relation-type	Points to the relation type resource for a relation instance.
relation-types	Points to the relation types feed under a repository.
relations	Points to the relations feed under a repository, relations feed for a specific relation type, or relations feed related to a specified object.
repositories	Points to the repositories feed from the docbrokers.
roles	Indicates xCP roles.

types	Points to the data dictionary types feed under a repository.
users	Points to the users feed under a repository, or direct member users under a group.
The fully qualified Documentum link relation path is prefixed with the following string: http://identifiers.emc.com/linkrel/	

Category for Single Resources

Table 18. Resource-Root Element Mapping

Resource	Root Element in XML/Name Property in JSON
SysObject	object
Current Version	object
Cabinet	cabinet
Folder	folder
User Default Folder	folder
Document	document
Network Location	network-location
Current User	user
User	user
Group	group
Relation	relation
Relation Type	relation-type
Repository	repository
Rendition	content
Format	format
Lock	object
Home Document	resource
Product Information	product-info
Child Link	folder-link
Batch Capabilities	batch-capabilities

Appendix C

Documentum Core Resources

This table lists the elements that you may need to reference when developing custom resources.

Resource	**Code Name**	**Model Class**	**View Class**
All versions	versions	AtomFeed	VersionsFeedView
Batch capabilities	batch-capabilities	batch .BatchCapabilities	NA
Batches	batches	batch.Batch	NA
Cabinet	cabinet	CabinetObject	CabinetView
Cabinets	cabinets	AtomFeed	CabinetView
Checked out objects	checked-out-objects	AtomFeed	SysObjectView
Child folder link	child-folder-link	FolderLink	FolderLinkView
Child folder links	child-folder-links	AtomFeed	FolderLinksFeedView
Content media resource	content-media	NA	NA
Content	content	ContentMeta	ContentView
Contents	contents	AtomFeed	ContentsFeedView
Current user	current-user	UserObject	UserView
Current version	current-version	SysObject	SysObjectView
Document	document	DocumentObject	DocumentView
Folder child documents	folder-child-document	AtomFeed	DocumentsFeedView
Folder child folders	folder-child-folders	AtomFeed	FoldersFeedView
Folder child objects	folder-child-objects	AtomFeed	SysObjectsFeedView
Folder	folder	FolderObject	FolderView
Format	format	Format	FormatView
Formats	formats	AtomFeed	FormatsFeedView
Group	group	GroupObject	GroupView
Group users	group-member-users	AtomFeed	UsersFeedView
Groups	groups	AtomFeed	GroupsFeedView

Resource	Code Name	Model Class	View Class
Home document	home-document	NA	NA
Lock	lock	SysObject	SysObjectView
Network locations	network-locations	AtomFeed	NetworkLocationsFeedView
Network location	network-location	NetworkLocation	NetworkLocationView
Parent folder link	parent-folder-link	FolderLink	FolderLinkView
Parent folder links	parent-folder-links	AtomFeed	FolderLinksFeedView
Product information	product-info	ProductInfo	ProductInfoView
Query	dql-query	AtomFeed	QueryResultFeedView
Relation	relation	RelationObject	RelationView
Relation type	relation-type	RelationTypeObject	RelationTypeView
Relation types	relation-types	AtomFeed	RelationTypesFeedView
Relations	relations	AtomFeed	RelationsFeedView
Repositories	repositories	AtomFeed	RepositoriesFeedView
Repository	repository	Repository	RepositoryView
Search	search	SearchAtomFeed	SearchFeedView
Sub groups	group-member -groups	AtomFeed	GroupsFeedView
SysObject	object	SysObject	SysObjectView
Type	type	TypeObject	TypeView
Types	types	AtomFeed	TypesFeedView
User default folder	default-folder	FolderObject	FolderView
User	user	UserObject	UserView
Users	users	AtomFeed	UsersFeedView

1. Model Classes for Core resources are in the package `com.emc.documentum.rest.model`.
2. View Classes for Core resources are in the package `com.emc.documentum.rest.view.impl`.