

EMC[®] Documentum[®] Platform REST Services

Version 7.3

Development Guide

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

Legal Notice

Copyright © 2013–2016 EMC Corporation. All Rights Reserved.

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com. Adobe and Adobe PDF Library are trademarks or registered trademarks of Adobe Systems Inc. in the U.S. and other countries. All other trademarks used herein are the property of their respective owners.

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

Preface	13
Chapter 1 Overview	15
Understanding RESTful Programming	15
Relations with Other Documentum Platform APIs	16
Chapter 2 Deploy Documentum Platform REST Services	17
General Deployment Configuration	17
Chapter 3 General REST Definitions	19
Common Definition - HTTP Headers	19
Common Definition - Query Parameters	21
HTTP Status Codes	24
Supported MIME Types	25
Other Types	25
Media Type for Home Document	25
MIME Type for Content	25
Multipart Type	26
URL Extension	26
Content Type of an Entry in a Feed	26
URI and URL	27
HTTP Methods	28
Web Client Caching	28
Representation	29
Collection Resource	29
Embedded Entry	32
Single Resource	35
Multi-part Request Representation	37
Error Representation	38
Transaction Support	40
Notes to PUT and POST Operation	40
Runtime Property Configuration	40
Runtime Profile	41
Batch Operations	41
Chapter 4 Resource Specific Features	43
Filter Expression	43
Literals	43
Numeric Literal	44
String Literal	44
Boolean Literal	45
Datetime Literal	45
Functions	47

starts-with.....	47
contains	47
between	48
type	48
nilled	49
Logical Operators	50
Comparison Operators.....	50
Comparison with Multiple Values	51
Comparison with Repeating Properties	51
Filter Expression Examples	52
Property View	52
Predefined View Expression.....	52
Custom View Expression	53
View Expression on Collections.....	53
View Expression Syntax	54
NULL in REST	55
NULL Value Representation.....	56
Whitespace in XML	56
Thumbnail Link.....	57
Feed Pagination.....	59
Full Text Query in Collection Resources	60
Simple Search Language	60
Words.....	61
Phrases	61
Implicit AND	61
Boolean Operators	62
Wildcards	62
Facet Search	62
Facet Search with URL Parameters.....	63
Facet Search with AQL.....	65
Lightweight and Shareable Objects	66
Get Lightweight and Shareable Type(s)	66
Generating Link Relation in DQL Results.....	67
Additional Information about Generating Links	68
Thumbnail support.....	69
Location of Persistent Data	72
Resolving Parent Folder	72
Resolving Sub-folder	73
Global Location Change.....	74
Chapter 5 Authentication	75
HTTP Basic Authentication	75
Enabling HTTP Basic Authentication	78
User Credential Mapping.....	78
Known Limitation	79
Kerberos Authentication	80
Authentication Workflow	80
Multi Domain Support within a Forest.....	81
Setup SPNEGO-based Kerberos.....	83
Content Server Configuration for SPNEGO-based Kerberos	83
REST Server Configuration for SPNEGO-based Kerberos	83
Register and Map the Service Principal Name	83
JAAS configuration.....	85
Configuring Runtime Properties.....	88

JAAS Configuration in Weblogic.....	89
Recommendations on Web Browser Configuration for SPNEGO-based Kerberos	90
Verify Browser Settings.....	91
Troubleshooting and Diagnostics.....	92
Retrieve debug information.....	92
Common Errors.....	93
CAS Authentication.....	95
Overview	95
Terminology.....	95
Authentication Workflow	96
CAS Single Sign Out	102
Configuration.....	103
Content Server	103
CAS Server.....	104
REST Server	107
Reverse Proxy Server	108
CAS Server Clustering	109
REST Server Clustering.....	109
Performance Consideration	109
RSA Authentication	111
Authentication workflow	111
Configuration.....	113
Configuration on the REST Server.....	113
Configuration on Content Server	113
Configuration on the RSA Web Agent	114
Configuration on the RSA Administrative Console	114
Siteminder Authentication	116
Authentication workflow	116
Configuration.....	117
Configuration on the REST Server.....	117
Configuration on Content Server	117
Special Considerations	118
SAML 2.0 Authentication	119
Documentum REST SAML based SSO Authentication Workflow	120
SAML Based SSO With One Identity Provider	120
Disable Redirecting for SAML SSO Initialization	126
SAML Based SSO with Multiple Identity Providers.....	127
Skip Identity Providers Discovery.....	128
Documentum REST SAML Logout Workflow	128
Configuration.....	130
REST Server	130
Identity Provider Server	132
Pre-authenticated Authentication	132
Workflow.....	133
Feature Highlights.....	134
Set Documentum Client Token Cookie or Not	135
Principal in Headers or Cookie	135
Principal Pattern.....	136
Multiple Authentication Schemes	136
Samples for Multi Authentication Schemes (HTTP Basic and Kerberos).....	137
Samples for Multi Authentication Schemes (HTTP Basic and CAS)	138
Reverse Proxy Configuration.....	141
Configuration Samples.....	142
Client Token.....	142

	Startup Script Modification	145
	Explicit Logoff.....	145
	Logoff Success Handling.....	145
	Client Token Encryption and Decryption	146
	Algorithms with a Key Size Larger than 128 bits	147
	Bypassing Browser Login Forms Upon HTTP 401 Unauthorized	148
Chapter 6	Deploying to Docker Containers	149
	Overview	149
	Docker Setup.....	150
	Docker Image of Documentum Platform REST Services	150
	Dockerizing your Documentum Platform REST Services Application.....	150
	Building the Docker Image.....	152
	Runtime Configuration	152
	Run the Container	154
	SSL Configuration	156
	Verify the REST SSL Communication	157
	Logging	158
	Scalability and High Availability	159
	Upgrading a Dockerized REST Service.....	160
Chapter 7	Resource Extensibility	163
	Overview	163
	Java API Deprecations	165
	Get Started With the Development Kit	167
	Maven-Based Toolkit	167
	Creating a Custom Resource Project from the Maven Archetype.....	167
	Verifying the Project	170
	Ant-Based Toolkit.....	172
	Architecture of Documentum REST Extensibility	173
	Documentum REST Security	174
	Documentum REST MVC	176
	Documentum REST Persistence	177
	Documentum REST Marshalling Framework.....	179
	Overview	179
	Features of the REST Marshalling Framework	179
	Annotations	180
	@SerializableType	180
	Examples	183
	Unmarshalling with Undefined Attributes	186
	@SerializableField	190
	Examples	193
	@SerializableField4XmlList.....	195
	@SerializableField4XmlMap	196
	Out-of-box Annotated Models.....	197
	Deprecations	197
	Additional Examples	198
	Java Primitive Types Support	200
	Java Interface, Abstract and Generic Types Support	202
	Marshalling.....	202
	Unmarshalling	204
	Java Collection and Array Support.....	210
	Array Serialization Support.....	210
	Java Collection Serialization Support	211

Supported Data Types.....	211
Exclusions.....	212
Marshalling with XML.....	212
Marshalling with JSON.....	216
Unmarshalling with XML and JSON.....	218
Unmarshalling with XML.....	218
Unmarshalling with JSON.....	220
Summary of Support for Java List Data Type.....	221
Limitation in Unmarshalling Number Lists.....	222
Java Map Support.....	222
Java Map Data Type Support.....	222
Supported Data Type.....	222
Inclusions.....	223
Marshalling.....	223
Marshalling with XML.....	223
Marshalling with JSON.....	227
Unmarshalling.....	229
Unmarshalling with XML.....	229
Unmarshalling with JSON.....	230
Summary of support for the Map Data Type.....	231
Circular References Not Supported.....	234
Custom Serializers and Deserializers.....	234
Programming Interface.....	234
Annotation Scanner.....	237
Developing Custom Resources.....	239
Designing Custom Resources.....	239
URI.....	239
HTTP Methods.....	240
Representations.....	240
Content Negotiation.....	240
Link Relations.....	241
Setting up a Custom Resource Project.....	241
Programming for Custom Resources.....	242
Model: Programming.....	242
Model: Extending Core.....	243
Model: Validation.....	244
Persistence: Programming.....	244
Persistence: Referencing Core.....	246
Persistence: Getting Login User Context.....	246
Persistence: Session Management.....	246
Persistence: Creating Feed Pages from DQL Result.....	249
Resource: Programming the Controller.....	250
Resource: Programming the View.....	251
Resource: Binding the View and the Controller.....	252
Resource: Customizing Resource Views.....	254
Customize Resource View.....	254
Hide Link Relations.....	254
Edit Link Relations.....	255
Hide Properties.....	256
Customize atom Attributes.....	257
Register Custom Resource View.....	258
View Loading Precedence.....	258
Troubleshooting.....	258
Resource: Building Resource links.....	262
Resource: Building Resource URIs with ResourceUriBuilder.....	263
Resource: Making It Queryable.....	265
Resource: Deciding Whether to Be Batchable.....	265
Resource: Extending Atom Feed and Entry.....	267

Resource: Error Handling and Representation	270
Resource: Consuming Multipart Contents in Custom Resource Controller	270
Linking Custom and Core Resources	271
Packaging and Deployment.....	271
Working with YAML Configuration.....	273
Registering URI Templates	273
Normalization of URI Templates	275
Normalization of Custom URI Templates	276
Normalization of Internal URI Templates	276
Formats for the href Template.....	277
Appending User Defined Variables onto the URI Template.....	277
Normalization of Custom URI Templates	278
Normalization of Internal URI Templates	278
Formats for href Template	279
User Defined Variables of the URI Template	279
Registering Root Resources	280
Adding Links to Core Resources.....	284
Disabling Specific Resources	289
Disable a Specific Resource with YAML	289
Impact of Disabling Core Resources.....	289
Samples	292
Overriding Specific Resources	293
Resource Overriding Configuration	294
Scenarios and Expected Results	294
Working With a URI Template When Overriding.....	296
How Resource Overriding Impacts Batchable Resources	296
Turning Off XML or JSON Media Types.....	297
Creating Custom Error Code Mapping Files	298
Creating Custom Message Files	298
Tutorial: Documentum Platform REST Services Extensibility Development.....	299
What to Build First	299
Quickstart	301
Creating Resource Model	302
Validating the annotated resource model	303
Creating Persistence.....	303
Creating Resource Controller	304
AliasSetCollectionController	305
AliasSetController	307
Spring Context Configuration.....	308
Creating Resource View	308
AliasSetsFeedView	308
AliasSetView	309
Adding More HTTP Methods.....	310
Creating an Alias Set Object	310
Deleting an Alias Set Object.....	312
Making Resources Queryable	312
Making Resources Linkable.....	313
Making Resources Non-Batchable (Optional)	315
Packaging and Deploying Resources.....	315
Troubleshooting	316
Chapter 8 Authentication Extensibility	317

Anonymous Access	317
Implementation.....	317
By Java Annotation.....	318
By Runtime Configuration	318
Extension Samples.....	319
Generic Servlet Filter Customization.....	320
Custom Authentication Development	321
Understanding The Security Filter Flows.....	321
Understanding The Authentication Framework.....	322
Authentication Extension	324
Customize DFC Session Manager	325
When To Customize.....	326
Customize Authentication Filter, Provider, and Entry Point.....	327
Authentication Provider.....	327
Authentication Filter.....	329
Authentication Entry Point.....	331
Client Token Integration	332
Spring Security Java Configuration	332
Implement Web Security	333
AbstractWebSecurity	333
@AuthSchemeProfile.....	334
@Order	334
Client Token Integration	335
Externalize Configuration Parameters	336
Configure Runtime Properties	337
Spring Security XML Configuration.....	337
Write XML Configuration.....	338
Configure Security XML Extension	340
Configure Runtime Properties	340
Packaging Artifacts.....	340
Samples	341
Tutorial: Documentum REST Authentication Extensibility Development.....	341
Chapter 9 Advanced Security Configuration	347
Default Security Headers	347
CSRF Protection	349
Enabling CSRF Protection	350
Generating a Token.....	350
Generating a Client-side Token.....	350
Initial Authentication.....	350
Runtime Configuration	351
Errors for Incomplete Authentication.....	352
Generating a Server-side Token	352
Initial Authentication.....	352
Token Lifetime	352
Runtime Configuration	353
Token Validation	353
Errors for Validation	354
HTTP Inspection Method	354
Cross-Origin Resource Sharing (CORS) Support.....	354
Enable REST Server CORS Support.....	354
Request Sanitization	355
Third Party Libraries.....	356
Sanitize the Input Object Metadata	356
Sanitize the Input HTML Content.....	357

	Configurations	358
	Customize the <i>AntiSamy</i> Policy Files	358
Chapter 10	Explore Documentum Platform REST Services	361
	Prerequisites	361
	Common Tasks on Folders, Documents, and Contents	362
Chapter 11	Tutorial: Consume REST Services Programmatically	375
	Basic Navigation	375
	Service Entry Point	376
	Link Relations	376
	Feeds and Entries	377
	From the Home Document Resource to Documents	378
	Read Entries.....	378
	Filter, Sort, and Pagination	379
	Create Entries.....	379
	Update Entries	380
	Delete Entries.....	381
Appendix A	Link Relations	383
Appendix B	Resource Coding Index	389

List of Figures

Figure 1.	The Launch Script Process.....	153
Figure 2.	Lifecycle of Building Custom Resources.....	163
Figure 3.	Maven Archetype Project Structure.....	168
Figure 4.	Ant Project Structure	172
Figure 5.	Documentum Platform REST Services Architecture	173
Figure 6.	Documentum REST MVC	176
Figure 7.	Java Primitive Types and XML/JSON Data Types Mapping	201
Figure 8.	Scan Report.....	244
Figure 9.	Java Annotations and Documentum MVC.....	254
Figure 10.	Code Structure	301
Figure 11.	Structure of Model.....	302
Figure 12.	Structure of Persistence.....	304
Figure 13.	Structure of Controller	305
Figure 14.	YAML File Location.....	314

List of Tables

Table 1.	Documentum Property to JSON Data Type	29
Table 2.	Metadata of a Feed	29
Table 3.	Metadata of an Entry	30
Table 4.	Elements/Properties in a Single Resource	35
Table 5.	Comparison Operators.....	50
Table 6.	Terminologies in CAS authentication	95
Table 7.	SerializableType Attributes	180
Table 8.	@SerializableField Attributes	191
Table 9.	@SerializableField4XmlList Attributes.....	196
Table 10.	@SerializableField4XmlMap Attributes	196
Table 11.	Supported Primitive Types and Wrapper Classes.....	201
Table 12.	URI Template Elements.....	274
Table 13.	Root Service Registry	280
Table 14.	Resource Link Registry	284
Table 15.	Resource design of the alias set collection.....	299
Table 16.	Resource design of the alias set resource	300
Table 17.	Resource design of the alias set resource	311
Table 18.	Public Link Relations	383
Table 19.	Link Relations Introduced in Documentum Platform REST Services	384

Preface

This document is a guide to using and customizing EMC Documentum Platform REST Services, which interact with Documentum repositories.

Intended Audience

This document is intended for developers and architects who are building applications that consume Documentum Platform REST Services.

Related Documentation

The following documentation provides additional information:

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

Conventions

The following conventions are used in this document:

Font Type	Meaning
<i>italic</i>	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 Platform 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 <i>EMC Documentum Platform REST Services Resource Reference Guide</i> 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
November 2016	Initial publication.

Overview

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

Being developed in a purely RESTful style, Documentum Platform 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 Platform REST Services the optimal choice for Web 2.0 applications and mobile applications to interact with Documentum repositories.

Documentum Platform 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 Platform REST Services supports two formats for resource representation:

- **JSON**
JavaScript Object Notation is a lightweight data interchange format based on a subset of the JavaScript Programming Language standard. Documentum Platform REST Services uses JSON as the primary format for representing resources.
- **XML**
XML is the dominant data format in traditional SOAP based Web Services, yet as well as being widely used in RESTful Web Services. Documentum Platform 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 Platform REST Services represents collection-based resources in Atom feeds, and represents non-collection based resources in Documentum XML documents.

Understanding RESTful Programming

Documentum Platform REST Services delivers a deployable Java web archive (WAR) file that runs in the web container of a Java EE application server (refer to the release notes for system requirements). The WAR file exposes the interface as network-accessible resources identified by URIs.

Documentum Platform REST Services is programming language independent. Therefore, you can consume its services using any programming language that has an HTTP client library, such as Java, .NET, Python, Ruby, and so on.

You can access the source code of Documentum Platform REST Services sample clients on [GitHub](#). The source code for these REST client samples is made available to the public through the Apache 2 license.

Relations with Other Documentum Platform APIs

Documentum Platform 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 Platform 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.

Deploy Documentum Platform REST Services

Documentum Platform 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

Documentum Platform REST Services release 7.3 and later can be deployed into a Docker container. For detailed information on how to deploy Documentum Platform REST Services, see the *EMC Documentum Platform and Platform Extensions Release Notes*.

General Deployment Configuration

On all application servers, the minimal and mandatory configuration is to set the docbroker information in location `<dctm-rest.war>\WEB-INF\classes\dfc.properties`. For more information on configuring `dfc.properties`, see the release notes.

For information on customizing the REST server, such as how to change an authentication scheme, use the REST runtime property file at `<dctm-rest.war>\WEB-INF\classes\rest-api-runtime.properties`. To see all available configuration parameters, see the template file located at `<dctm-rest.war>\WEB-INF\classes\rest-api-runtime.properties.template`.

For capturing logs and changing the logging level, use the `log4j` properties file at located at `<dctm-rest.war>\WEB-INF\classes\log4j.properties`.

General REST Definitions

Common Definition - HTTP Headers

Documentum Platform 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: <code>Authorization: Basic QWxhZGRpbjpwYXNkd29yZQ==</code> Or, Kerberos authentication header with the credential part encoded, for example: <code>Authorization: Negotiate YIIZG1hZG1pbjpwYXNkd29yZ...</code>
Accept	Acceptable media type for the Response	Request	See Supported MIME Types, page 25
Content-Type	MIME type of the Request body or response body	Request /Response	See Supported MIME Types, page 25 The REST server ignores the <code>charset</code> parameter in the Content-Type header. Therefore, the error <code>E_INPUT_MESSAGE_NOT_READABLE</code> 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
Set-Cookie	Sets an HTTP cookie Response	Response	Used by client token

HTTP Header Name	Description	In Request or Response?	Value Range
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 Platform 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	<ul style="list-style-type: none"> • <code>true</code> - return the object instance and embed the object instance into the entry's content element. • <code>false</code> - do not return object instance. 	false Note: Although the default value is <code>false</code> , we still recommend that you try setting this parameter to <code>true</code> 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 <code>false</code> 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 <code>items-per-page</code> to 200, and <code>page</code> 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 <code>inline</code> is set to <code>true</code> .	string	See Property View , page 52.	: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	<ul style="list-style-type: none"> • <code>true</code> - return the total number of objects. • <code>false</code> - do not return the total number of objects. 	false
sort	Specifies a set of the sort specifications in a returned collection.	string	<p>This parameter consists of multiple sort specifications, separated by comma (,).</p> <p>Each sort specification consists of a property (any non-repeating property) by which to sort the results and its sort order (<code>DESC</code> or <code>ASC</code>), separated by the whitespace character ().</p> <p>The sort order is optional. if not specified, the default sort order is <code>ASC</code>.</p> <p>If any property with an invalid name is specified, an error is thrown.</p> <p>Example:</p> <pre>sort=r_modify_date desc,object_id asc,title</pre>	NA
links	<p>Determines whether or not to return link relations in the object representation</p> <p>This parameter works only when <code>inline</code> is set to <code>true</code>.</p>	boolean	<ul style="list-style-type: none"> • <code>true</code> - return link relations. • <code>false</code> - 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	<ul style="list-style-type: none">• <code>true</code> - return all indirect children recursively when a request tries to get the children of an object.• <code>false</code> - Only return direct children when a request tries to get the children of an object.	false
filter	Filter expression	string	Filter Expression, page 43	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 Platform 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 at 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 Platform REST Services supports XML and JSON representations with the following MIME types:

MIME Type	Usage
application/vnd.emc.documentum+json (default MIME type)	JSON representation
application/atom+xml	XML representation for feeds (collections)
application/vnd.emc.documentum+xml	XML representation for a single object
application/json	JSON representation for compatible viewing
application/xml	XML representation for compatible viewing
<p>Note:</p> <ul style="list-style-type: none">• If the client does not specify the Accept header for the expecting response, the REST server uses default MIME type <code>application/vnd.emc.documentum+json</code>.• 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 Platform REST Services supports are not limited to the types in this table. For more information about other supported types, see Other Types, page 25.	

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

URL Extension

In Documentum Platform REST Services, you can use the URL extensions `.xml` and `.json` to negotiate the content type of the Response. For more information, 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
<code>.json</code>	<code>application/json</code>
<code>.xml</code>	<code>application/xml</code>
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
<code>application/atom+xml</code>	<code>application/vnd.emc.documentum+xml</code>
<code>application/xml</code>	<code>application/xml</code>

Feed Content Type	Entry Content Type
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 Platform REST Services, URIs are the only resource identifiers.

In Documentum Platform 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 Platform REST Services, see the following RFC document:

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

HTTP Methods

Documentum Platform 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 Platform REST Services does not support the `OPTIONS` method except in Cross-Origin Resource Sharing preflight requests. For more information, see the section titled *Cross-Origin Resource Sharing (CORS) Support*. 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 Platform REST Services supports web client caching on the content media resource using an HTTP ETag token.

Representation

Documentum Platform 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
String	String
ID	String
Time	String
Repeating	Array

XML is the other format for resource representation in Documentum Platform REST Services. Like the JSON format, collection-based resources are presented as feeds, defined by Atom. For more information about Atom, see RFC4287.

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

Collection Resource

In XML and JSON 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

Feed Metadata	Description
author	Feed author
self	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
content	<p>When the entry is not inline, it contains a <code>src</code> attribute whose value is the URI of the single item resource.</p> <p>When the entry is inline, it embeds the full representation of the single item resource. For more information, see Embedded Entry, page 32.</p> <p>When you set <code>inline</code> to <code>false</code>, <code>content-type</code> in JSON and <code>type</code> in XML indicate the MIME type of the resource.</p>
edit	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.

```
{
  id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets" ← Feed Id
  title: "Cabinets" ← Feed Title
  updated: "2013-06-19T15:14:39.679+08:00" ← Feed Updated
  -author: [1] ← Feed Authors
    -0: {
      name: "EMC Documentum"
    }
  -links: [1] ← Feed Links
    -0: {
      rel: "self"
      href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets"
    }
  -entries: [11] ← Feed Entries
    -0: {
      id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/objects/0c0004d280000104" ← Entry Id
      title: "acme01" ← Entry Title
      updated: "2012-10-15T15:27:30.000+08:00" ← Entry Updated
      summary: "dm_cabinet 0c0004d280000104"
      -author: [1] ← Entry Authors
        -0: {
          name: "acme01"
          uri: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/users/acme01"
        }
      -content: { ← Entry Content, pointing to the JSON format of cabinet resource. Alternately,
        content-type: "application/json" full representation of the cabinet can be embedded within entry content
        src: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
      }
      -links: [1] ← Entry Links
        -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.

```

{
  id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets" ← Feed Id
  title: "Cabinets" ← Feed Title
  updated: "2013-06-19T15:14:39.679+08:00" ← Feed Updated
  -author: [1] ← Feed Authors
    -0: {
      name: "EMC Documentum"
    }
  -links: [1] ← Feed Links
    -0: {
      rel: "self"
      href: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets"
    }
  -entries: [11] ← Feed Entries
    -0: {
      id: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/objects/0c0004d280000104" ← Entry Id
      title: "acme01" ← Entry Title
      updated: "2012-10-15T15:27:30.000+08:00" ← Entry Updated
      summary: "dm_cabinet 0c0004d280000104"
      -author: [1] ← Entry Authors
        -0: {
          name: "acme01"
          uri: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/users/acme01"
        }
      -content: { ← Entry Content, pointing to the JSON format of cabinet resource. Alternately,
        content-type: "application/json" full representation of the cabinet can be embedded within entry content
        src: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
      }
      -links: [1] ← Entry Links
        -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://localhost:8080/dctm-rest/repositories/acme01/users/61636d653031"
      }
    ],
    "content": {
      "name": "cabinet",
      "type": "dm_cabinet",
      "definition": "http://localhost: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://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
    },
    {
      "rel": "edit",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
    },
    {
      "rel": "http://identifiers.emc.com/documentum/linkrel/delete",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000104"
    },
    {
      "rel": "http://identifiers.emc.com/documentum/linkrel/folders",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0c0004d280000104/folders"
    },
    {
      "rel": "http://identifiers.emc.com/documentum/linkrel/documents",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0c0004d280000104/documents"
    },
    {
      "rel": "http://identifiers.emc.com/documentum/linkrel/objects",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0c0004d280000104/objects"
    },
    {
      "rel": "http://identifiers.emc.com/documentum/linkrel/child-links",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0c0004d280000104/child-links"
    },
    {
      "rel": "http://identifiers.emc.com/documentum/linkrel/relations",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/relations?related-object-id=0c0004d280000104&related-object-role=any"
    }
  ]
},

```

```
    "links": [
      {
        "rel": "edit",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01/
cabinets/0c0004d280000104"
      }
    ],
    "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://localhost: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://localhost: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://localhost:8080/dctm-rest/
      repositories/acme01/cabinets/0c0004d280000104" />
      <dm:link rel="edit" href= "http://localhost:8080/dctm-rest/
      repositories/acme01/cabinets/0c0004d280000104" />
      <dm:link rel="http://identifiers.emc.com/documentum/linkrel/delete"
```

```

        href= "http://localhost:8080/dctm-rest/repositories/acme01/
        cabinets/0c0004d280000104" />
<dm:link rel="http://identifiers.emc.com/documentum/linkrel/folders"
        href= "http://localhost:8080/dctm-rest/repositories/acme01/
        folders/0c0004d280000104/folders" />
<dm:link rel="http://identifiers.emc.com/documentum/linkrel/documents"
        href= "http://localhost:8080/dctm-rest/repositories/
        acme01/folders/0c0004d280000104/documents" />
<dm:link rel="http://identifiers.emc.com/documentum/linkrel/objects"
        href= "http://localhost:8080/dctm-rest/repositories/
        acme01/folders/0c0004d280000104/objects" />
<dm:link rel="http://identifiers.emc.com/documentum/linkrel/child-links"
        href= "http://localhost:8080/dctm-rest/repositories/
        acme01/folders/0c0004d280000104/child-links" />
<dm:link rel="http://identifiers.emc.com/documentum/linkrel/relations"
        href= "http://localhost:8080/dctm-rest/repositories/acme01/
        relations?related-object-id=0c0004d280000104&related-object-role=any" />
</dm:links>
</dm:cabinet>
</content>
<link rel="edit" href= "http://localhost: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.
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" ← name property as the resource category
  type: "dm_cabinet" ← Object Type
  definition: "http://core-rs-demo.lss.emc.com:8080/dctm-rest/repositories/acme01/types/dm_cabinet" ← Object Type Resource
  -properties: { ← Object Properties
    r_object_id: "0c0004d280000104" ← Single Property
    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] ← Repeating Property
      0: "/acme01"
  }
  -links: [8] ← Links
    -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" ? Object Type
< 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"> Object Properties
    < r_object_id>0c0004d280000104</r_object_id>
    < object_name>acme01</object_name> Single Property
    < 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> Repeating Property
      < item>/acme01</item>
    </r_folder_path>
  </properties>
  < links> 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> root element as the resource category
```

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:object xmlns:dm="http://identifiers.emc.com/documentum">
  <properties>
    <dm:object_name>hello</dm:object_name>
  </properties>
</dm:object>
--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
- REST Error Id (mandatory) - a unique identifier of the error in the log

Documentum Platform 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>xxx</status>
  <code>xxx</code>
  <message>xxx</message>
  <details>xxx</details>
  <id>xxx</id>
</error>
```

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

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<error>
  <status>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>
  <id>2275d26b-9761-43c9-9ca0-cd3a006d6c41</id>
</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>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>
<id>9dd9d03c-78fb-4c86-881c-d06cf2b5884b</id>
</error>

```

Example 3-6. JSON Error Representation

```

{
  "status":xxx,
  "code":"xxx",
  "message":"xxx",
  "details":"xxx",
  "id":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;",
  "id":"2275d26b-9761-43c9-9ca0-cd3a006c4541"
}

```

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;",
  "id":"2275d26b-9761-43c9-9ca0-cd3a006d6e64"
}

```

Transaction Support

Documentum Platform 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 Platform 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`.

Runtime Profile

You can customize the runtime profile by using the `rest.runtime.profile= runtime` property.

There are two runtime profiles that control how error messages are shown. The two runtime profiles are *development* and *production*. This property specifies which runtime profile is used for error messages.

The default runtime profile is *production*, which returns a wrapped exception message so that clients do not see server implementation specific information, which could be a security issue making the REST server vulnerable to attack.

The other runtime profile is *development*, and it returns implementation specific messages (the original error messages) including all server information. This runtime profile is meant to be used for development purposes only.

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 Platform 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*.

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. For instance, this can be used on the Cabinets Resource, Folder Child Documents Resource and so on. For example, GET requests to this URL returns all cabinets owned by user `dmadmin`: `/repositories/REPO/cabinets?inline=true&filter=owner_name='dmadmin'`.

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

- [Literals, page 43](#), which describes the literal formats in filter expressions.
- [Functions, page 47](#), which describes the functions in filter expressions.
- [Logical Operators, page 50](#), which describes the logical operators supported by filter expressions.
- [Comparison Operators, page 50](#), which describes the comparison operators supported by filter expressions.

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

Note: The attributes that can be used in a filter expression are determined by the implementation of the individual collection resources. For example, on the Cabinets Resource, all `dm_cabinet` type attributes can be used in a filter expression. On the Folder Child Objects Resource, all `dm_sysobject` attributes can be used.

When you want to reduce collection items to make them a sub type of the collection, you can use the `filter` and `object-type` query parameters together. In this case, the attributes used in the filter expression can come from the type specified by `object-type` parameter. For example, GET requests to this URL returns all cabinets of sub type `custom_cabinet` with a `custom_region` sub type attribute that equals `apj`: `/repositories/REPO/cabinets?inline=true&object-type&filter=custom_region='apj'`

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 44](#)
- [String Literal, page 44](#)
- [Boolean Literal, page 45](#)
- [Datetime Literal, page 45](#)

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 data type, 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 syntaxes:

- `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, page 47](#)
- [contains, page 47](#)
- [between, page 48](#)
- [type, page 48](#)
- [nilled, page 49](#)

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
<code>eq [1]</code>	<code>=</code>	Equal	<code>object_name = "REST"</code> <code>object_name eq REST</code>
<code>ne</code>	<code>!=</code>	Not equal	<code>object_name != "REST"</code> <code>object_name ne "REST"</code>
<code>lt</code>	<code><</code>	Less than	<code>r_modify_date < "2013-01-16"</code> <code>r_modify_date lt "2013-01-16"</code>
<code>le</code>	<code><=</code>	Less than or equal to	<code>r_full_content_size <= 2000</code> <code>r_full_content_size le 2000</code>
<code>gt</code>	<code>></code>	Greater than	<code>r_full_content_size > 2000</code> <code>r_full_content_size gt 2000</code>

Value comparison operator	General comparison operator	Description	Example
ge	>=	Greater than or equal to	<pre>r_modify_date >= "2013-01-16"</pre> <pre>r_modify_date ge "2013-01-16"</pre>
<p>[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 51.</p>			

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 Platform 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 <code>view</code> 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.

When a property name that is specified in the expression violates any of the naming rules shown above, the view expression is considered to be invalid and is therefore rejected. In this case, the server returns an HTTP 400 Bad Request error code status.

View Expression on Collections

Except for the rules above, in versions prior to version 7.3 you cannot specify a custom property in a custom view expression when trying to get a collection resource. Otherwise, an HTTP 400 Bad Request error code status is returned.

Example 4-13. URLs Regarded as Bad Requests

For example, GET requests to a URL that resembles the following examples 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 in versions before 7.3, you can set the `view` parameter to `:all`. Documentum Platform REST Services release 7.3 and later support tolerant

view expressions on most collection resources. The *view* parameter's usage on singular resources remains the same.

Note: When the *view* parameter is used on a single object resource that supports this parameter (such as the Cabinet resource, Document resource, etc.), the *view* attribute names must match the definition of the object data type. A *Bad request* (400) error is returned when any unknown attributes for the object data type are specified in the *view* parameter.

When the *view* parameter is used on a collection based resource that supports this parameter (such as the Cabinets resource, Folder Child Documents resource, etc.), a comma separated list of *view* attributes can contain both base type attribute names and arbitrary extended attribute names.

For example, let's assume that a folder X has a number of documents in it with the object type *dm_document* and document sub types *doc_ext1*, *doc_ext2*, and so on. When a REST client retrieves the folder children, it can specify the *view* attributes from both *dm_document* and its sub types.

Here is a code sample that shows you the Request:

```
// Get a folder child documents feed.
// In this sample, attributes 'r_object_id' and 'object_name' come from dm_document.
// 'doc_ext1_attr' comes from doc_ext1, and 'doc_ext2_attr' comes from dmc_ext2.
GET /documents?inline=true&
    view=r_object_id,object_name,doc_ext1_attr,doc_ext2_attr HTTP/1.1
```

Please also note the following *view* usage on a collection based resource:

- Aspect type attributes can also be put into the *view* when the aspect type is attachable to the collection base type, such as *dm_document* is in the above sample.
- Collection items return the specified attributes only when the attributes exist in those objects.
- Unknown attributes for collection items are ignored.

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 4-14. 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 4-15. 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 Platform 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 Data type	JSON/XML Data type	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 4-16. NULL value for Datetime in XML

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

Example 4-17. NULL value for Datetime in JSON

```
"property_name":null
```

Example 4-18. NULL value for String in XML

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

Example 4-19. NULL value for String in JSON

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

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

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

Example 4-21. 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 Platform REST Services preserve XML space for both input and output XML messages, so the client must normalize the Request before sending. For details, see the following examples:

Example 4-22. Preserve Whitespace Within the Request

```
<document>
  <properties>
    <object_name>  my      document  </object_name>
  </properties>
</document>
```

Example 4-23. Preserve Whitespace in the Response

```
<document xmlns="..." xsi:type="dm_document" definition="...">
  <properties xsi:type="dm_document-properties">
    <r_object_id>0900000180010d8e</r_object_id>
    <object_name>  my      document  </object_name>
    ...
  </properties>
</document>
```


Note: Even if all whitespace characters are processed by Documentum Platform 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
<code>thumbnail</code>	<p>Specifies whether or not to return the thumbnail link for each entry.</p> <ul style="list-style-type: none"> • <code>true</code> - Return the thumbnail link for each entry in the collection resource. • <code>false</code> - Do not return the thumbnail link for each entry in the collection resource. 	boolean	Value of the <code>rest.entry.thumbnail.default</code> parameter in the <code>rest-api-runtime.properties</code> file, which defaults to <code>false</code> .

Example 4-24. 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"/>
<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"/>
  <link rel="edit"
href="/repositories/acme01/documents/090004d280005123"/>
  <link rel="icon" href="/thumbsrv/getThumbnail?path=000004d2\80
\00\00\5f.jpg&store=thumbnail_store_01"/>
</entry>
<entry>...</entry>
</feed>
```

Example 4-25. 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"
    }
  ],
  "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"
      },
      "links": [
        {
          "rel": "edit",
          "href": "/repositories/acme01/documents/090004d280005123"
        },
        {
```

```

        "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 Platform REST Services utilizes the following query parameters for feed pagination (For detailed information, see [Common Definition - HTTP Headers, page 19](#)):

- `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:

- `first`
This link relation is always available.
- `last`
This 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 that parentheses, which can be used in Search, are not supported in the parameter `q` when being appended to the collection resources above.

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 Platform REST Services supports full-text search by facets, which enables you to explore a collection of search results categorized into specific dimensions called 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 the use of facets on other properties, you must modify your xPlore configuration and re-index.

When you perform a facet search, a facet section appears in the Response feed of the search result at the same level as the `entry` element.

Facet Search with URL Parameters

More information on how to use facet requests with URL parameters can be found in the *Documentum Platform REST Services Reference Guide*. Here is a code sample that shows you facet results:

```
<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?
        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?
          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>
  </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 <code>facet-id</code> , a property is prefixed with <code>facet_</code> . In the example, the <code>r_modify_date</code> 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 <code>r_modify_date</code> is <code>Modify Date</code> .
facet-value	Each <code>facet-value</code> section contains data of the results belonging to one group. In the example, two <code>facet-value</code> sections appear, one for instances whose <code>r_modify_date</code> falls in last year's date range and the other for instances whose <code>r_modify_date</code> 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
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 <code>r_modify_date</code> falls in last year's date range by accessing this link: <pre>http://localhost:8080/dctm-rest/repositories /acme01/search? 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</pre>

When 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:

- You use the repeating property `keywords` as the facet.
- The result set contains three objects:
 - `Object1`, which contains the keywords `foobar` and `V1`.
 - `Object2`, which contains the keywords `foobar` and `V2`.
 - `Object3`, which 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, when 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 because their keywords contain both `foobar` and `V1`. Similarly, the link `http://host:port /dctm-rest/repositories/documentum1/search?q`

=Helloworld&facet=keywords&facet-value-constraints= V1%2BV2 returns two results, Object1 and Object3.

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: These '+' and '|' operators have the same precedence in facet-value-constraints, and the expression is evaluated from right to left. For example, the expression `a+b|c+d` is treated as `a+(b|(c+d))`.

Facet Search with AQL

Here is a code sample that shows the facet request that is used when you want to navigate into the facet groups:

Example 4-26. Navigate into a Facet Group

```
<search>
  <expression-set>
    <expressions>
      <property-list name="owner_name" operator="IN">
        <values>
          <value>dmadmin</value>
        </values>
      </property-list>
    </expressions>
  </expression-set>
  <facet-definitions>
    <facet-definition id="facet_type">
      <attributes>
        <attribute>r_object_type</attribute>
      </attributes>
    </facet-definition>
  </facet-definitions>
</search>
```

Below are the facet results. The elements in this set of facet results are similar to the results from the [Facet Search with URL Parameters, page 63](#) section. However, the URL for the link relation search has a new element called *facet-id-constraints* that contains the key value pairs for each facet id and its constraint. The id should correspond to the facet id in the AQL. Here is a code sample that shows you facet results and the *facet-id-constraints* parameter:

Example 4-27. Facet Results and Constraints

```
<feed>
  <entry>
    ...
  </entry>
  <dm:facet
    xmlns:dm="http://identifiers.emc.com/vocab/documentum">
    <dm:facet-id>facet_type</dm:facet-id>
    <dm:facet-label>Type</dm:facet-label>
    <dm:facet-value>
      <dm:facet-id>facet_type</dm:facet-id>
      <dm:facet-value-id>dm_document</dm:facet-value-id>
      <dm:facet-value-count>30</dm:facet-value-count>
      <dm:facet-value-constraint>dm_document</dm:facet-value-constraint>
      <link rel="search" href="http://localhost:8080/dctm-rest/repositories/
```

```
        REPO/search?facet-id-constraints=facet_type%3Ddm_document"/>
    </dm:facet-value>
    <dm:facet-value>
        <dm:facet-id>facet_type</dm:facet-id>
        <dm:facet-value-id>dm_cabinet</dm:facet-value-id>
        <dm:facet-value-count>5</dm:facet-value-count>
        <dm:facet-value-constraint>dm_cabinet</dm:facet-value-constraint>
        <link rel="search" href="http://localhost:8080/dctm-rest/repositories/
            REPO/search?facet-id-constraints=facet_type%3Ddm_cabinet"/>
    </dm:facet-value>
</dm:facet>
</feed>
```

As you can see in the above code sample, the cabinet type has five results. You can navigate into this facet group by using the POST HTTP method with the original request to the URL in the facet result `http://localhost:8080/dctm-rest/repositories/REPO/search.xml?facet-id-constraints=facet_type%3Ddm_cabinet`.

For more information on using Facets requests with AQL, see the *Documentum Platform REST Services Reference Guide* for version 7.3 or later.

Lightweight and Shareable Objects

A lightweight object type is optimized to reduce the storage space needed in a database. When a lightweight SysObject is created, it references a shareable supertype object. Additional lightweight SysObjects that are created also reference the same shareable object.

The shareable supertype object is called the lightweight SysObject parent. The lightweight SysObject that references the supertype is called the child.

Each lightweight SysObject shares the information of its shareable parent object. Instead of having multiple, nearly identical, rows in the SysObject database tables to support all instances of the lightweight type, a single parent object exists for multiple lightweight SysObjects.

Shareable Type

Shareable type instances can share their property values with lightweight types. Multiple lightweight objects can share the property values of one shareable object.

Lightweight Type

Lightweight object type is a child of shareable object type. It is used to reduce the space required to store it in a database.

Get Lightweight and Shareable Type(s)

Lightweight and Shareable types can be retrieved by an existing Type(s) resource. Follow this procedure to retrieve a Lightweight and Shareable type:

1. Use the filter attribute `type_category` on the Types Resource to get Lightweight object types. Here is an example of the URI:

```
/repositories/{repositoryName}/types?filter=type_category=4
```

Note: You must include your repository name in the URI. Also notice the parameter *filter=type_category=4*, which is used for Lightweight object types.

2. Use the filter attribute *type_category* on the Types Resource to get Sharable object types. Here is an example of the URI:

```
/repositories/{repositoryName}/types?filter=type_category=2
```

Note: You must include your repository name in the URI. Also notice the parameter *filter=type_category=2*, which is used for Shareable object types.

3. The Shareable object types have a *lightweight-types* link relation that points to their relative child Lightweight object type.

Similarly, the lightweight object types have a link relation *parent-shareable-type* that points to their parent Shareable object type. The link relation *parent* points to the super type.

4. The *category* property has been added to the Type Resource version 7.3 and later to show the type of the category, and the *shared-parent* property has also been added to the Type Resource version 7.3 and later to show the *shared-parent* of the lightweight type. Here's a code sample that shows you how to use these two properties:

```
<type label="MyLightweight" name="my_lightweight"
  parent="http://localhost:8080/dctm-rest/repositories/REPO/types/my_shareable"
  shared-parent="http://localhost:8080/dctm-rest/repositories/REPO/types/my_shareable"
  category="shareable"
  xmlns="http://identifiers.emc.com/vocab/documentum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <properties></properties>
  <links>
    <link
      href="http://localhost:8080/dctm-rest/repositories/REPO/types/my_lightweight"
      rel="self"/>
    <link
      href="http://localhost:8080/dctm-rest/repositories/REPO/types?
parent-shareable-type=my_shareable" rel="http://identifiers.emc.com/linkrel/types",
    <link
      href="http://localhost:8080/dctm-rest/repositories/REPO/types/my_shareable"
      rel="parent"/>
  </links>
</type>
```

Generating Link Relation in DQL Results

To facilitate content consumption, Documentum Platform 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 *editlink* relation pointing to that resource.

Example 4-28. 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 4-29. 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 Platform 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

When the query result contains the `thumbnail_url` attribute, the system uses the value of this attribute to generate a link to the [thumbnail](#).

Example 4-30. Query Results with Thumbnail Links Generated in XML

Sample Request:

```
GET http://localhost:8080/dctm-rest/repositories/REPO?
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?
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?
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?
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?
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" rel="edit"/>
<link
href="http://localhost:8081/thumbsrv/getThumbnail?"
```

```
object_type=dm_cryptographic_key&format=&is_vdm=false&repository=8320"
rel="icon"/>
</entry>
</feed>
```

Example 4-31. Query Results with Thumbnail Links Generated in JSON

Sample Request:

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

Sample Response:

```
{
  id: "http://localhost: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://localhost:8080/dctm-rest/repositories/documentum1?dql=
      select r_object_id,object_name,r_object_type,
      thumbnail_url from dm_sysobject"
  }

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

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

  entries: [
    {
      id: "http://localhost: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://localhost: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://localhost:8081/thumbsrv/getThumbnail?object_type=
dm_cryptographic_key&format=&is_vdm=false&repository=1000001"
  },
  links: [...]

}

links: [

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

  {
    rel: "icon"
    href: "http://localhost:8081/thumbsrv/getThumbnail?object_type=
dm_cryptographic_key&format=&is_vdm=false&repository=1000001"
  }
]
},

{
  id: "http://localhost: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://localhost: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://localhost:8081/thumbsrv/getThumbnail?object_type=
dm_public_key_certificate&format=&is_vdm=false&repository=1000001"
    }
  }

  links: [...]

}

links: [

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

  {
    rel: "icon"
    href: "http://localhost:8081/thumbsrv/getThumbnail?object_type=dm_public_key_certificate&
format=&is_vdm=false&repository=1000001"
  }
]
]]

```

```
}
```

Location of Persistent Data

Some resources found in Documentum Platform REST Services release 7.3 and later, such as Saved Search Resource and User Preference Resource, must create internal persistent data in the repository. The correct configuration of the persistent data folder is required to prevent data access errors from occurring.

REST Services decides where to store persistent data depending on your runtime environment setting, the user accessing the persistent data, and the resource generating the data. This section discusses how REST Services resolves locations of persistent data.

The path of the folder storing persistent data consists of the two parts, a parent folder path followed by a sub folder path.

- Parent folder path: The parent folder can be a global location specified in the `rest-api-runtime.properties` file, the user's default folder, or the temp folder. For more information, see [Resolving Parent Folder](#).
- Sub folder path: The path of a sub folder follows this pattern: `${user_name}_${resource_name}`. For more information, see [Resolving Sub Folder](#).

Examples (Parent folder paths are presented in *italic*):

- */System/Applications/rest-persistence/admin_saved_search*
- */testuser_default/testuser_saved_search*
- */temp/myuser_myresource*

Resolving Parent Folder

When resolving the parent folder of persistence data, REST Services has the following options. These options are listed in the order of precedence, with the highest-precedence at the top.

- Global location specified in `rest-api-runtime.properties`
- User's default folder
- `/Temp`

If none of the options is enabled, the system returns a 403 error code to the client that attempts to generate persistent data.

Global location

The global location is the root folder for all persistence data. With this location specified, operations generating persistent data create sub folders under the root to store persistence.

The runtime property `rest.persistence.folder` specifies the global location of persistence data. For example:

```
rest.persistence.folder=/System/Applications/rest-persistence
```


If you specify an invalid path, the system returns a 403 error code. In this case, the system will not try the two lower precedence options (user default folder or /Temp folder).

Make sure that you grant users Write permission on this folder as they have to create sub folders there. To do this, set the basic permission of `dm_world` to 6 for the global location folder as shown in the following snippet.

```
<permission-set
  xmlns="http://identifiers.emc.com/vocab/documentum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <permitted>
    <permission accessor="dm_world" basic-permission="6"/>
    <permission accessor="dm_owner" basic-permission="7"
      extend-permissions="EXECUTE_PROC,CHANGE_LOCATION"/>
  </permitted>
  <links>
    <link rel="self" href="http://10.32.94.31:8080/">
    <link rel="edit" href="http://10.32.94.31:8080/">
    <link rel="http://identifiers.emc.com/linkrel/acl"
      href="http://10.32.94.31:8080/repositories/REPO/acls/45024c518000110c"/>
  </links>
</permission-set>
```

For security reasons, you may want to hide implementation details. Marking the global location folder hidden serves the purpose:

```
<folder
  xmlns="http://identifiers.emc.com/vocab/documentum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:type="dm_folder"
  definition="http://10.32.94.31:8080/repositories/REPO/types/dm_folder">
  <properties>
    <object_name>rest-persistence</object_name>
    <r_object_type>dm_folder</r_object_type>
    ...
    <a_is_hidden>true</a_is_hidden>
    ...
  </properties>
</folder>
```

User's default folder

If a global location for persistent data does not exist, REST Services tries the user' default folder (specified by `default_folder`) as an alternative.

Temp folder

Non-admin users may not have a default folder. In this case, REST services tries the /Temp folder. Because the /Temp folder can be used as the storage for persistent data, be cautious when you delete this folder or remove data from the folder.

Resolving Sub-folder

Once the parent folder is resolved, REST Services checks whether the parent folder has a sub-folder `${user_name}_${resource_name}` to save persistent data. If the sub folder exists, REST Services returns the path `paranet_folder\sub_folder` to the client for persistent data storage. Note that the user initiating the operation must be granted Write permission on the sub folder. If the sub folder

does not exist, the client creates the sub folder on behalf of the user. If the user does not have the writer permission on the parent folder, the system throws an exception.

Global Location Change

When you change the runtime property `rest.persistence.folder`, only persistent data generated after the change is stored in the new location. Old persistent data stays in the original global location folder. However, when you update a resource, such as the `Saved search` or the `User preference` resource that generates persistent data after a global location change, the data store location does not change.

Authentication

Documentum Platform REST Services supports the following authentication schemes:

- SAML 2.0 Web Single Sign-on authentication for LDAP users
- Pre-authenticated authentication for Web Access Management solutions
- 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

Note: In addition to these out-of-the-box authentication schemes, *Documentum Platform REST Services* also allows you to create your own custom authentication schemes using the extensibility library.

For more information, see [Chapter 8, Authentication Extensibility](#).

HTTP Basic Authentication

HTTP Basic authentication sends usernames and passwords to the REST server for authentication.

Note: We strongly recommend that you use HTTPS, which requires a secure connection, when using HTTP Basic authentication because HTTP Basic authentication transmits the username 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})
```

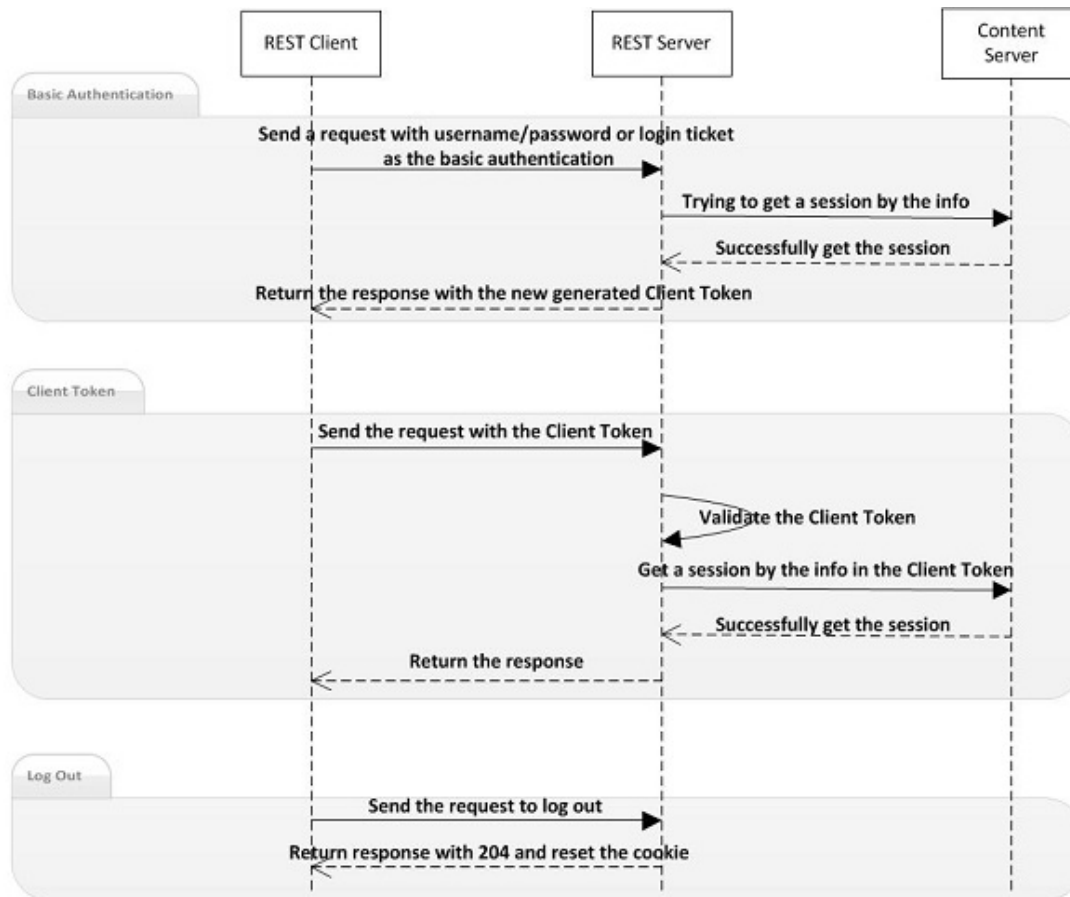
Example 5-1. Username/password Pair for dmadmin/password

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

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

Documentum Platform REST Services release 7.2 and later allow you to improve the efficiency of HTTP Basic authentication by using [client tokens](#). This improvement allows an authenticated REST client to access the REST server without having 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 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 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
```

For more information, see [Explicit Logoff](#).

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

Enabling HTTP Basic Authentication

HTTP Basic is the default authentication scheme in Documentum Platform 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 (':').

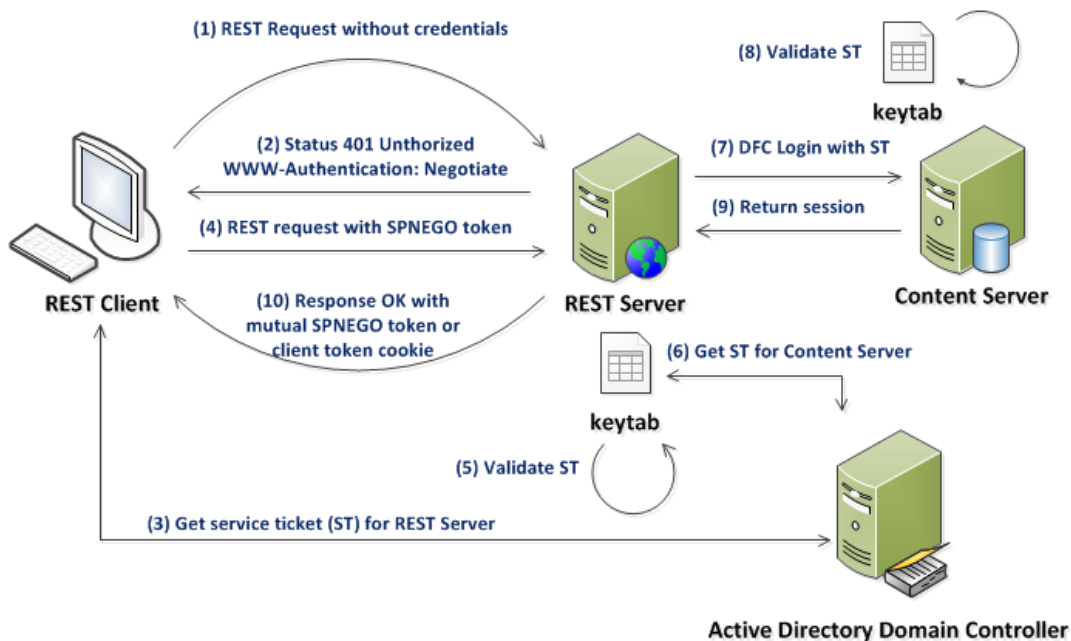
Kerberos Authentication

Documentum Platform REST Services implements SPNEGO-based Kerberos authentication to authenticate a request when Documentum Platform REST Services is registered as a Kerberos service. EMC recommends 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 Platform REST Services:



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 142](#). 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 must 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 re-sends 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 CXXCBAbjpwYXBSS90B.....
```

Note: A Kerberos service ticket is meant to be used only once. A REST client cannot resubmit the same SPNEGO token multiple times for authentication. When 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

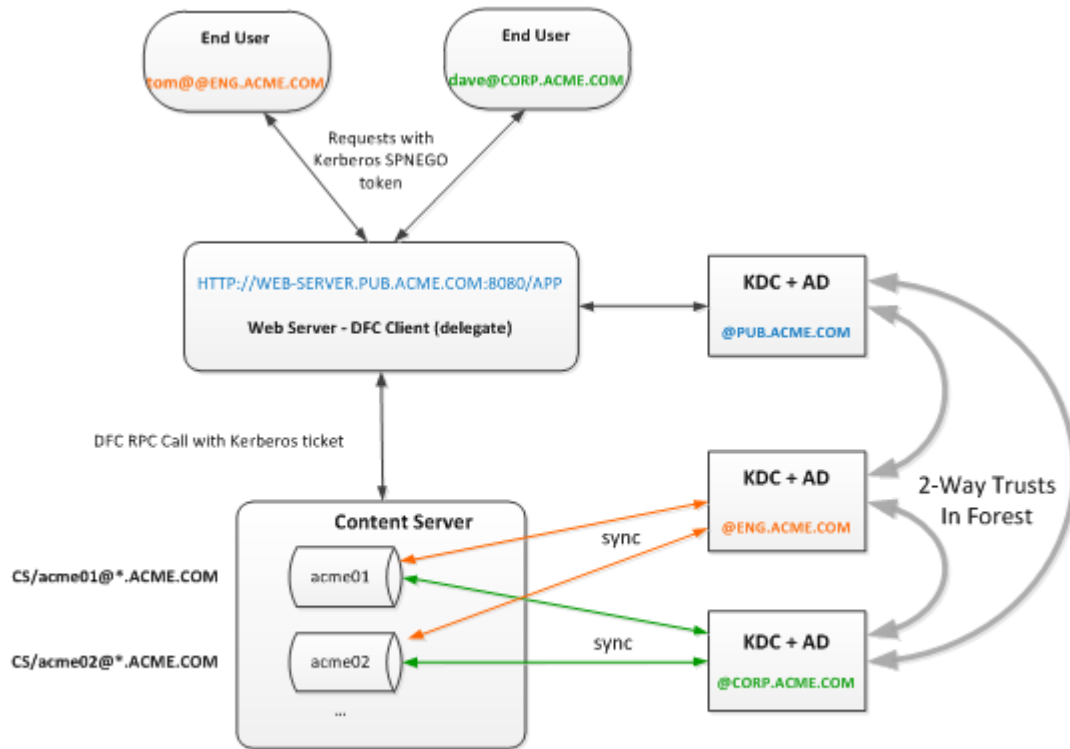
In Documentum Platform REST Services, Kerberos authentication can be implemented across multiple domains when one of the following conditions is true:

- The domains are in the same forest.
- Two-way trusts are enabled across the 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 has supported 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:



Caution: Domain names must be entered all in uppercase letters. Otherwise the operation will fail during the authentication process. This is a requirement of Content Server.

- The Documentum Platform 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 Platform REST Services, the following configurations are required:

- [Content Server Configuration for SPNEGO-based Kerberos, page 83](#)
- [REST Server Configuration for SPNEGO-based Kerberos, page 83](#)
- [Recommendations on Web Browser Configuration for SPNEGO-based Kerberos, page 90](#)

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 83](#)
- [JAAS configuration, page 85](#)
- [Configuring Runtime Properties, page 88](#)
- Startup 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 your 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. Run the following command to create a keytab file for the SPN and map the SPN to the service account:

```
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 [Ktpass Syntax](#).

Example 5-2. A Ktpass Command

```
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 Platform 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.



Caution: The **Delegation** tab does not exist until you run the ktpass command to map the principal.

Note: Kerberos constrained delegation is supported in version 7.3 and later. By setting the REST runtime property `rest.security.kerberos.constrained.delegation` to `true`, Documentum Platform REST Services principal can be delegated to a limited set of Content Server repository service principals.

In the *Active Domain Computers and Users Management* console, select "Trust this user for delegation to specific services only", then select "Use Kerberos only" and add service principals for the Content Server repositories.

Constrained Delegation cannot be configured across multiple domains and can only be used in a single domain scenario. This is a limitation of the MS KDC configuration.

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`. However, in WebSphere Application Server, the JAAS configuration file is named `wsjaas.conf` under the following directory:

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

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 is located in the following directory:

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

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 JBoss 6.3 the deployment must go in the `jboss-eap-6.3\standalone\deployments` directory.

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 `standalone.xml` file. Any configuration settings that are made to any other files are ignored by the Jboss 6.3 Application Server.

Element Id	Description
<code><security-domain></code>	<p>Corresponds to the Documentum REST web application's SPN. You replace separator characters with hyphen characters and omit the <code>@REALM</code> segment in the SPN.</p> <p>For example, the following Security domain is derived from the corresponding SPN:</p> <ul style="list-style-type: none"> • security-domain: HTTP-myhost-mydomain-com-8080 • SPN: HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM <p>Note: Make sure that the SPN in the JAAS configuration matches the SPN defined in <code>rest-api-runtime.properties</code>.</p>

Element Id	Description
<LoginModule>	<p>For both single and multi domain, the REST services always uses the Quest login module.</p> <ul style="list-style-type: none"> com.dstc.security.kerberos .jaas .KerberosLoginModule <p>Note: When you want to enable ticket cache, perform one of the following operations:</p> <ul style="list-style-type: none"> Enable createTicketCache: <pre>useTicketCache=true createTicketCache=true</pre> Enable createTicketCache and specify a cache path: <pre>useTicketCache=true createTicketCache=true ticketCache=<cache_path> 32</pre> <p>Otherwise, disable ticket cache by setting useTicketCache to false.</p>
<SPN>	The Documentum REST web application's SPN. 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
<REST_user_keytab_path>>	The path to the user account's *.keytab file on the Documentum REST web application. For example: C:\restuser.keytab

Example 5-3. 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/restuser.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="notTGT" value="true"/>
      <module-option name="realm" value=" MYDOMAIN.MYCORP.COM "/>
    </login-module>
  </authentication>
</security-domain>
```

Example 5-4. 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="RETKDC.IIG.EMC.COM"
    refreshKrb5Config=true
    noTGT=true
    useKeyTab=true
    storeKey=true
    doNotPrompt=true
    useTicketCache=false
    isInitiator=false
    keyTab=<REST_user_keytab_path>;
};
```

<loginContext>	<p>Corresponds to the Documentum Platform 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:</p> <ul style="list-style-type: none"> • LoginContext: <pre>HTTP-myhost-mydomain-com-8080</pre> • SPN: <pre>HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM</pre> <p>Note: Make sure that the SPN in the JAAS configuration matches the SPN defined in <code>rest-api-runtime.properties</code>.</p>
<LoginModule>	<p>Specify the Kerberos login module to be used to perform user authentication.</p> <p>Referring to QUEST Libraries for both Single domain and Multi-Domain: <code>com.dstc.security.kerberos.jaas.KerberosLoginModule</code></p> <p>Note: If you want to enable ticket cache, perform one of the following operations. Otherwise, disable ticket cache by setting <code>useTicketCache</code> to <code>false</code>.</p> <ul style="list-style-type: none"> • Enable createTicketCache: <pre>useTicketCache=true createTicketCache=true</pre> • Enable createTicketCache and specify a cache path: <pre>useTicketCache=true createTicketCache=true ticketCache=<cache_path></pre>

<SPN>	<p>For QUEST login modules, the SPN does not contain the @ character and the string after that. For example:</p> <pre>HTTP/myhost.mydomain.com:8080</pre> <p>If Documentum Platform REST Services is deployed on WebLogic, append the realm name to the SPN. For example:</p> <pre>HTTP/myhost.mydomain.com:8080@MYDOMAIN.MYCORP.COM</pre>
<REALM>	<p>(Multi-domain support only) The realm name. For example:</p> <pre>@MYDOMAIN.MYCORP.COM</pre>
<REST_user_keytab_path>	<p>The path to the user account's *.keytab file on the REST server. For example: c:\RESTuser.keytab</p>

Configuring Runtime Properties

The `rest-api-runtime.properties` file contains the runtime properties for Documentum Platform 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 89](#))
- 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 Platform 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
```

Recommendations on Web Browser Configuration for SPNEGO-based Kerberos

You may want to build a web browser application with access to Documentum Platform 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 Platform 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 Platform REST Services is: `http://restserver.acme.com:8080/dctm-rest/`.
- You have created a Kerberos SPN `HTTP/RESTSERVER.ACME.COM` for Documentum Platform 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 Platform 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 Platform 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 Platform 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 5-5. 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 Platform 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`

When 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.
 - You must be able to forward the service ticket to issue on the client.

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 to forward the tickets it produces, any ticket that the Kerberos TGT produces (including the service ticket) is forwarded, this causes the Kerberos authentication to fail.

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>. Please 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 to allow it to be forwarded.

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>. Please 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 to allow it to be forwarded.

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

CAS Authentication

Overview

Documentum Platform 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 Platform 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

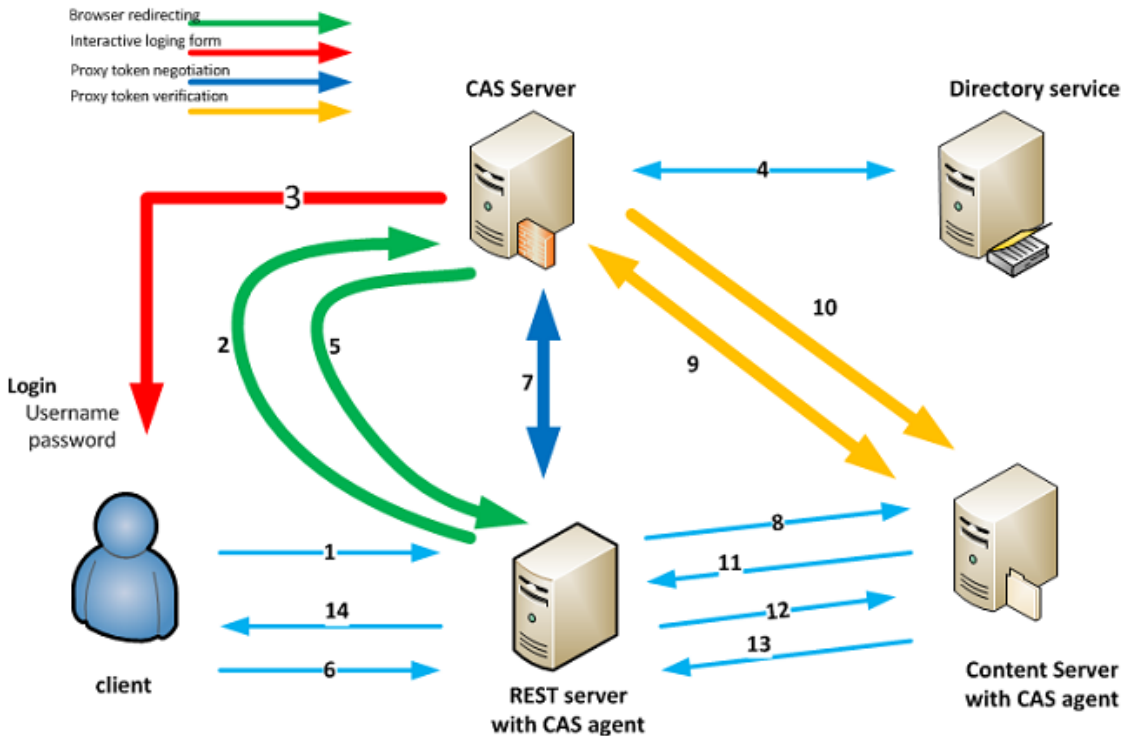
Term	Description
Client Token (CT)	Authentication token that the REST server provides for clients. This token contains an encrypted Documentum ticket and additional metadata used for token expiration. For more information about how a client token works, see Client Token, page 142 .
Ticket Granting Ticket (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

CAS is an open-source Java server component. For more information about CAS, see [Central Authentication Service](#).

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 status code redirection 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 status code redirection 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-mHMDsK0A9sAhie2Tldep-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 can only be used one time.

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 100](#).
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
```

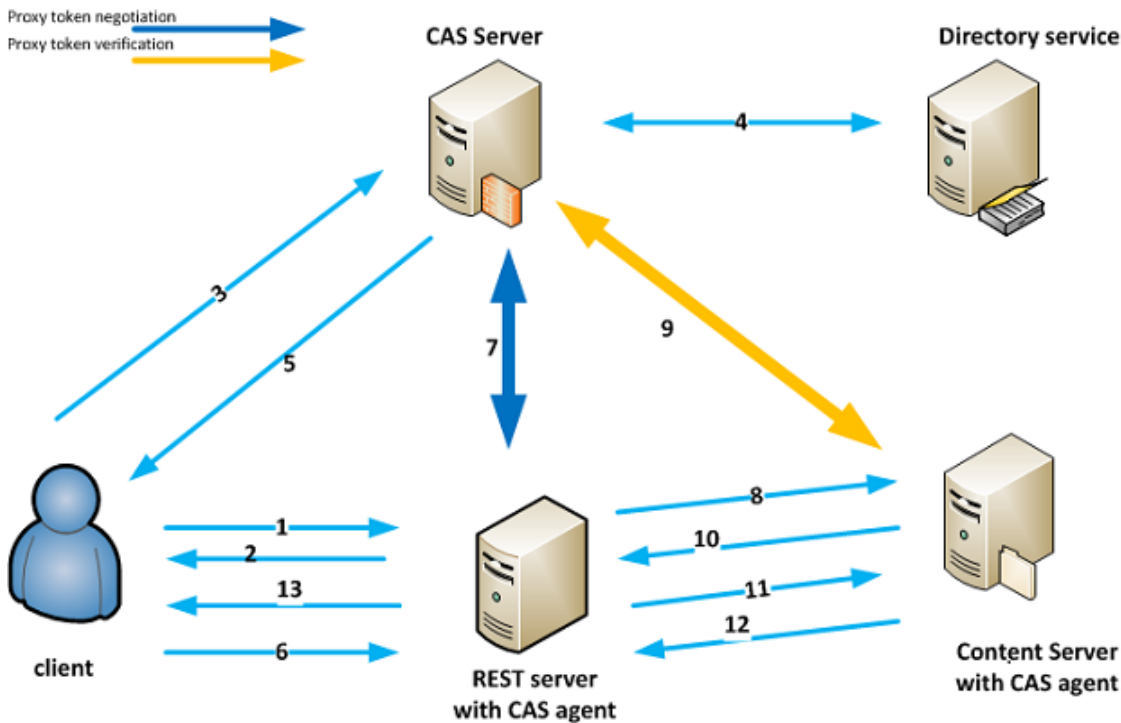
10. The CAS server validates the PT. If the PT is valid, the CAS server responds Content Server with the proxy address.

```
<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 142](#)) 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="Ym9ibGVlOkRX1RJQ0tFVD1UMEpL...==";
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.

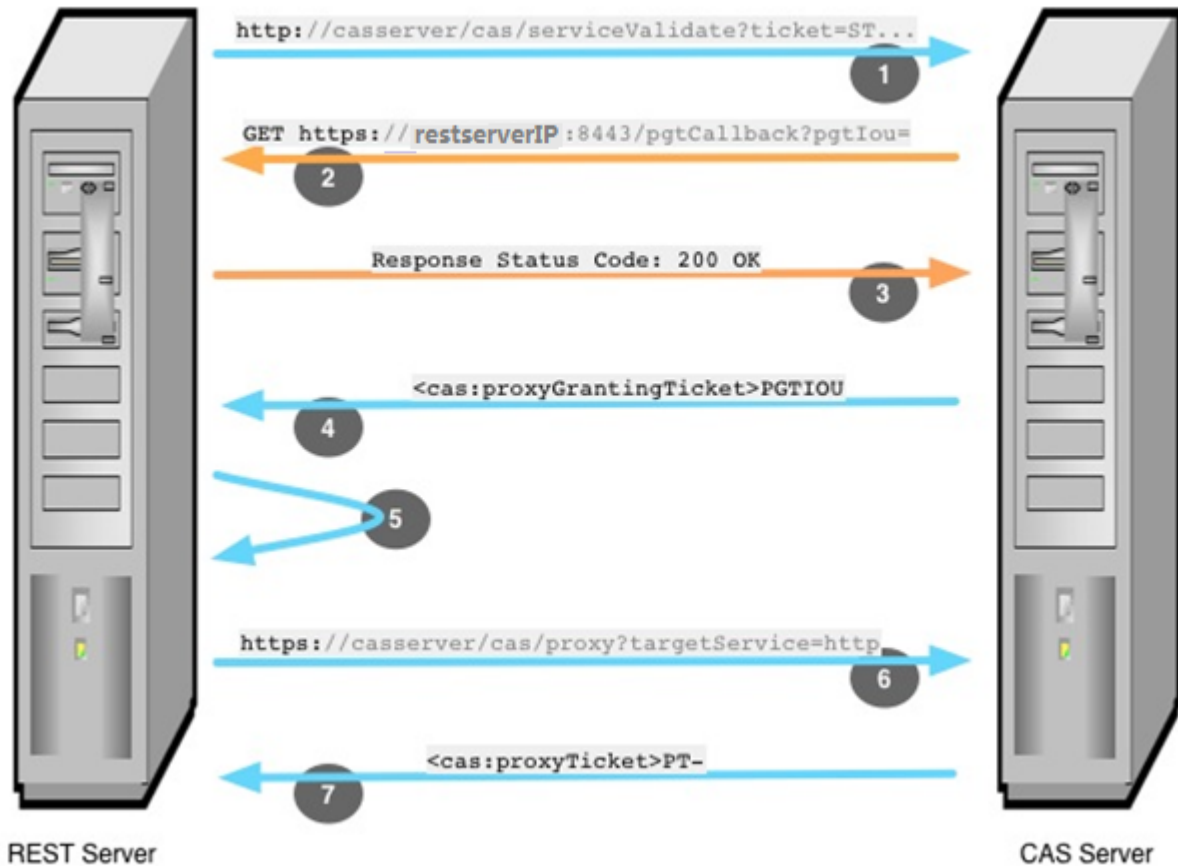
7. The client sends a POST request to the REST server to consume the resource, with the ST appended in the ticket parameter.

===== 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 96](#).

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:



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
CookieDOCUMENTUM-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-85-8PFx8qipjkWYDbuBbNJlroVu4yeb9WJIRdngg7fzl523Eti2td
</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 Platform REST Services requires configurations on the following servers:

- [Content Server, page 103](#)
- [CAS Server, page 104](#)
- [REST Server, page 107](#)
- [Reverse Proxy Server, page 108](#)

Content Server

To Enable CAS authentication for Documentum Platform 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_LOADED` info.

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 Platform 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:username="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://domainctrlr.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 Platform 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 Platform 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 108](#).

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 Platform 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 141](#)

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 Platform 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 100](#)). 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 utilize Ehcache to perform the replication of PGT IOU/PGT mappings across the cluster.

To enable Ehcache for PGT IOU/PGT mappings in Documentum Platform 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`.

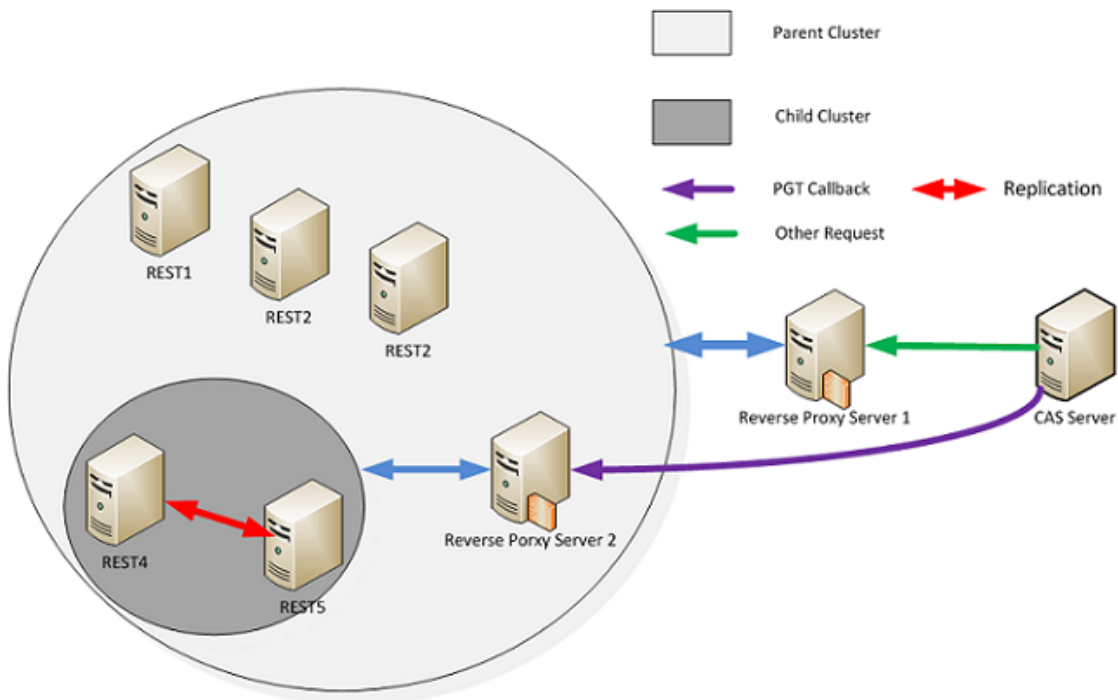
Additionally, you must follow the instructions on the following web site to configure Ehcache in `<dctm-rest>/WEB-INF/classes/rest-api-common-ehcache.xml`.

<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`:

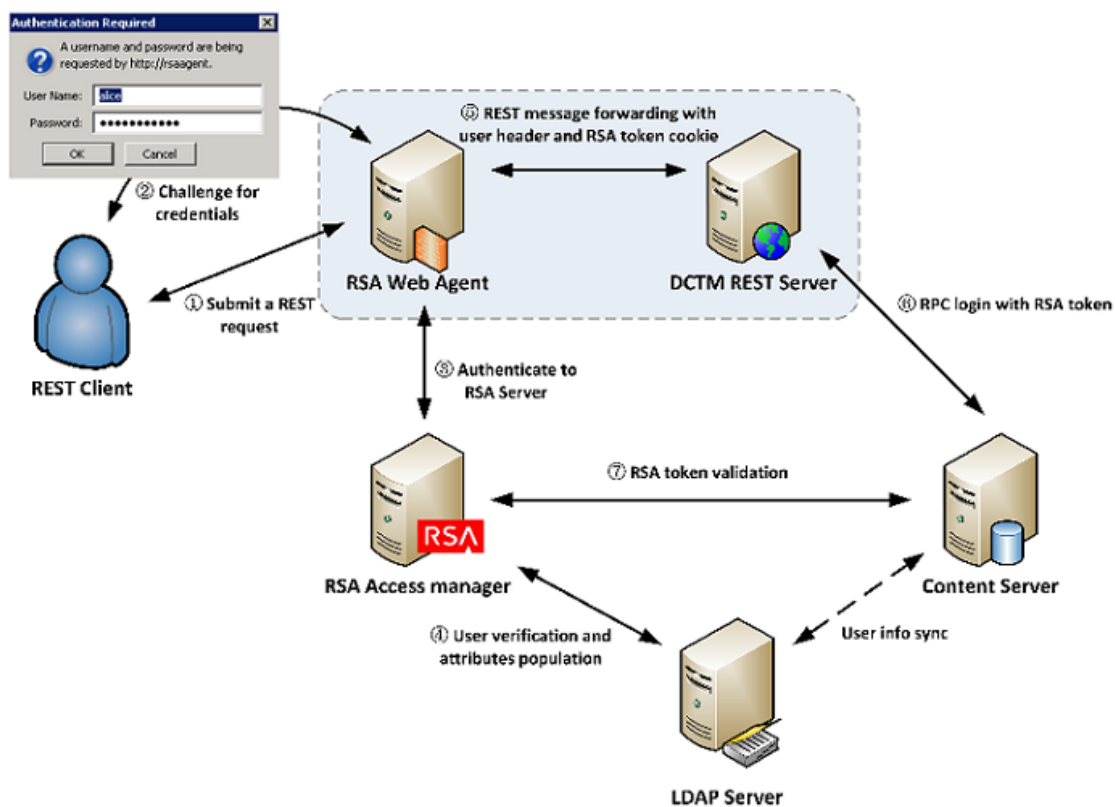
```
rest.security.auth.cas.client.pgt.storage=ehcache
```
- You must configure peer discovery for all REST servers in the child cluster and all of these servers must be set as peers.

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 Platform 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.



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 username header is required. The REST server uses this header to log in to Content Server.
6. The REST server uses the RSA token and username 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 Platform REST Services requires configurations on the following servers:

- [Configuration on the REST Server, page 113](#)
- [Configuration on Content Server, page 113](#)
- [Configuration on the RSA Web Agent, page 114](#)
- [Configuration on the RSA Administrative Console, page 114](#)

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 141](#) 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**.

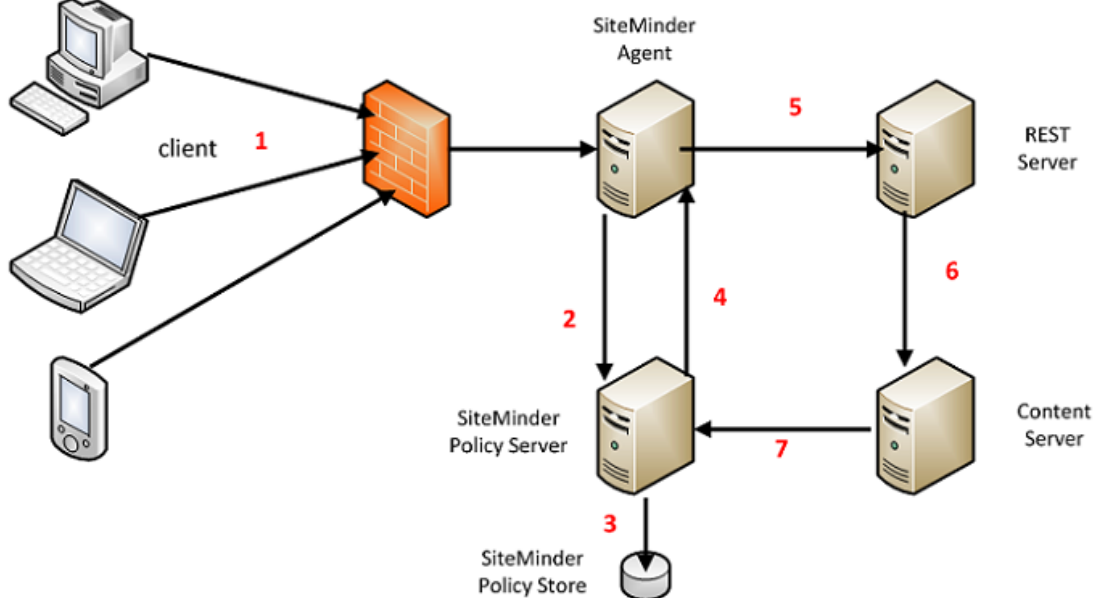
2. Create a new application, for example **CoreREST**,
3. Add a new resource to the application you created in step 2 with the resource type **URL** and set URL to **dctm-rest/repositories/***
4. Navigate to the Entitlements settings of the user or groups that the user belongs to, and select **Allow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

SiteMinder Authentication

Documentum Platform 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 username 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 117](#)
- [Configuration on Content Server, page 117](#)

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 141](#).

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 Platform REST Services does not have any restriction on characters used in URLs. However settings of the `BadUrlChars` and `BadQueryChars` 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 Platform 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.

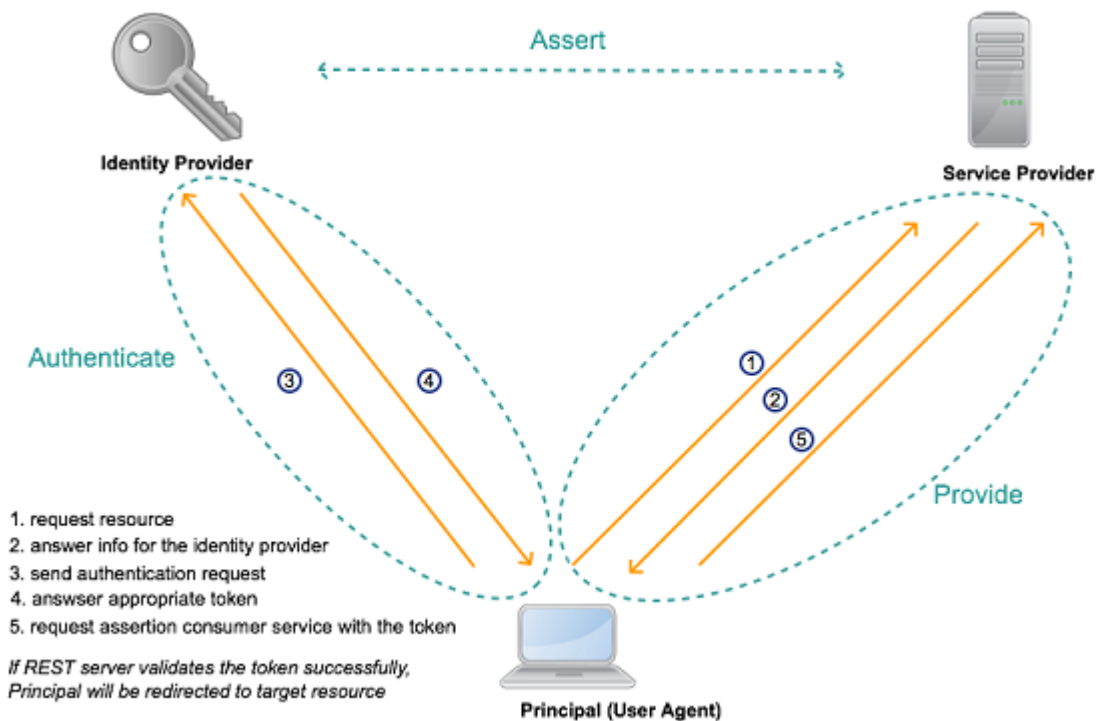
SAML 2.0 Authentication

Security Assertion Markup Language 2.0 (SAML 2.0) is an XML-based protocol in version 2.0 of the SAML standard. The SAML standard is used to exchange authentication and authorization data between secure domains making Single Sign-on (SSO) using web-enabled authentication and authorization possible. Documentum Platform REST Services supports SAML 2.0 based SSO authentication.

In a typical SAML SSO scenario, there are three participants:

1. Principal - A principal is an entity that is acting on behalf of a user
2. Service Provider- A service provider is an entity that provides a web-based service, application, or resource
3. Identity Provider - An identity provider is an entity that authenticates principals and produces assertions of authentication and attribute information in accordance with the SAML Assertions and Protocols specification

The figure below illustrates a typical workflow of SAML based SSO.



For more information, see [SAML 2.0](#).

With Documentum REST SAML SSO, the Principal is the REST client; the Service Provider works as a two component entity: The Documentum REST server is the front end, and the Content Server is the back-end. The Identity Provider can be any SAML based authentication products, such as ADFS 2.0.

Documentum REST SAML based SSO Authentication Workflow

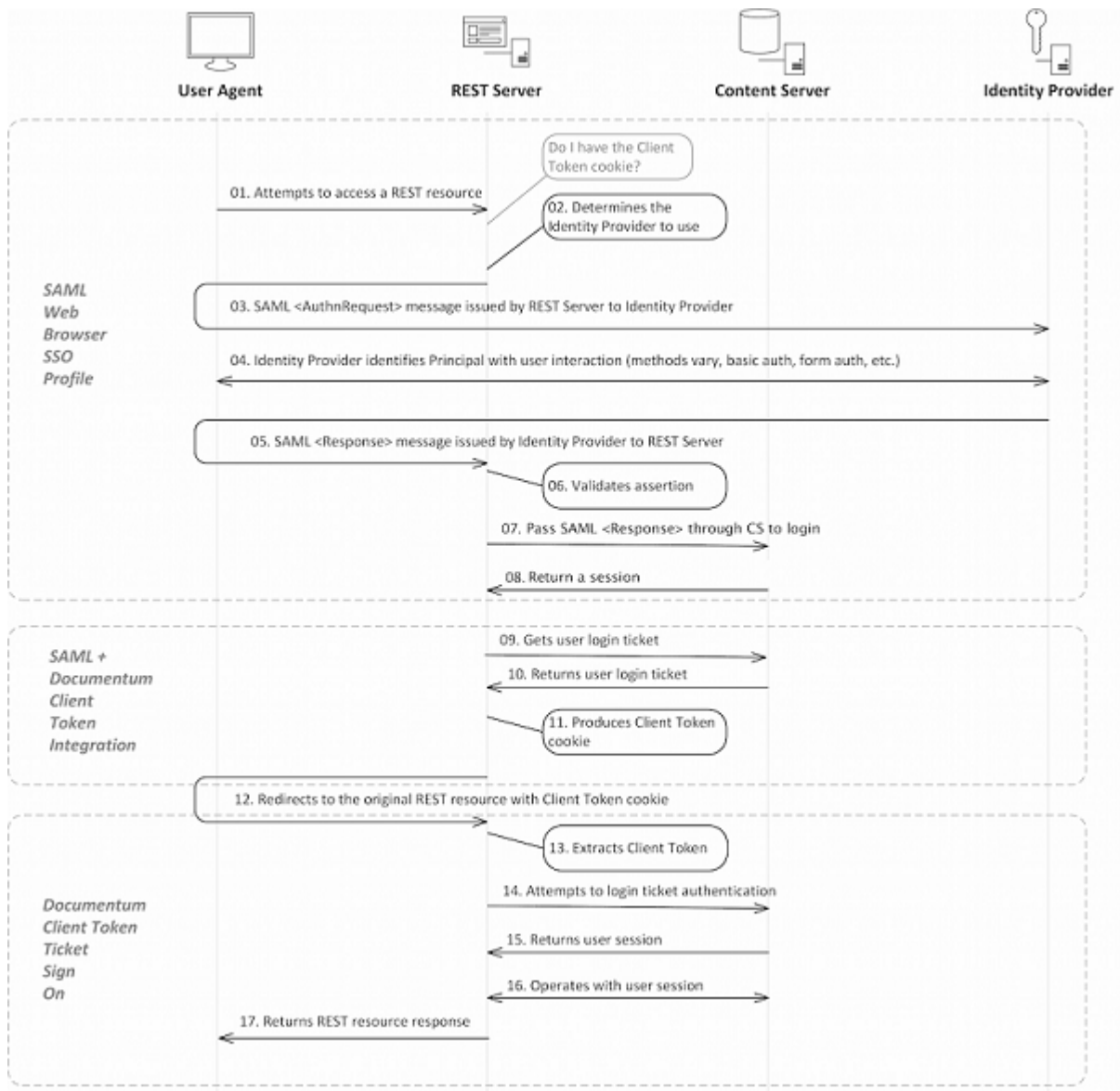
Comparing to the standard SAML SSO process, the scenario of Documentum REST is different.

- Besides Principal (REST client), Identity Provider and Service Provider (Documentum REST), a new participant is involved - Content Server. After the Documentum REST server verifies the Response token successfully, it passes the SAML assertion token to the Content Server for a DFC session. When the token is validated on the Content Server, the Documentum REST server gets a session, which can be used to execute various operations.
- The SAML SSO procedure involves multiple parties to complete the authentication, and is slower than a single pass-through login. To avoid using the SAML login for each request, Documentum REST for SAML SSO has been integrated with Documentum Client Token. After the SAML login succeeds, Documentum REST Server requires the Documentum Login Ticket from the Content Server with the SAML DFC session. Documentum REST Server uses these elements to produce the Client Token and set the Client Token in the REST client cookie. All the subsequent REST requests are based on Client Token authentication, with the exception of the SAML login.

During the deployment phase, the REST Server can have one or multiple Identity Provider metadata files. When more than one Identity Provider is available, the Identity Provider used to authenticate is determined during the SAML SSO initialization, also known as the Identity Provider Discovery.

SAML Based SSO With One Identity Provider

When only one Identity Provider metadata file is configured in the REST Server, this Identity Provider is the default provider. No negotiation between the REST Server and client is done to determine the Identity Provider. The following figure describes the detailed steps of SAML SSO in a Documentum REST scenario with one Identity Provider configured.



The steps illustrated in the diagram are as follows:

1. Attempts to request a REST resource

Typically, the repository is the first resource that requires authentication.

```
//REQUEST - initial repository resource request
GET https://restsp:8443/dctm-rest/repositories/REPO HTTP/1.1
```

In this step, the User Agent does not know which Identity Provider to use for authentication.

2. Determines which Identity Provider to use

Since the REST Server has only one Identity Provider's metadata file, this is the Identity Provider authentication service that is used.

When multiple Identity Provider metadata files are configured on the REST Server, there is an additional discovery phase so that the User Agent and server can negotiate which Identity Provider to use. The multiple Identity Provider case will be described later in this document.

3. The REST Server initializes SAML <AuthnRequest> and sends it to the Identity Provider

The REST Server assembles a SAML <AuthnRequest> and redirects the User Agent to the Identity Provider.

A cookie DOCUMENTUM-SAML2-REQUEST-TOKEN is sent back to the client and is used to validate the SAML Response against the original SAML Request.

```
//RESPONSE - Redirect the User Agent to the Identity Provider
//              with SAML <AuthnRequest>
HTTP/1.1 302 Found
Set Cookie DOCUMENTUM-SAML2-REQUEST-
TOKEN-a59abj125gb93th3447ae97ie8b1a7d; Secure; HttpOnly
Location https://idp.rest.org/idp.rest.org/idp/profile/SAML2/Redirect/SSO?
SAMLRequest-jVLLbs...sb4h8%3D&RelayState-%2Frepositories%2FREPO&SigAlg
http%3A%2F%2Fwww.w3.org%2F2000%2Fxmldsig%23rsa-sha1&Signature=
HSW...%3D%3D
```

```
//REQUEST - Sending the SAML <AuthnRequest> to the Identity Provider
GET http://idp.rest.org/idp/profile/SAML2/Redirect/SSO?
SAMLRequest-jVLLbs...sb4h8%3D&RelayState-%2Frepositories%2FREPO&SigAlg
http%3A%2F%2Fwww.w3.org%2F2000%2Fxmldsig%23rsa-sha1&Signature=
HSWv...%3D%3D HTTP/1.1
```

4. The Identity Provider authenticates the Principal

The Identity Provider communicates with the User Agent for the credential. The process is different for various kinds of Identity Providers.

```
//RESPONSE - Redirect the User Agent to the authorization engine
HTTP/1.1 302 Found
Set-Cookie JSESSIONID=60E013C4F9B6AE0F7A9CE1FF8EF92CBA; Path=/idp/; Secure; HttpOnly
Set-Cookie _idp_authn_lc_key
=8fa4ae6aa43d337ec48fbf355dd21bf1b81d8e3524787f2a6b675872c19f86ea;
Location https://idp.rest.org:443/idp/AuthnEngine
```

```
//REQUEST - Send Request to authorization engine
GET /idp/AuthnEngine HTTP/1.1
Host: idp.rest.org
Cookie: JSESSIONID=60E013C4F9B6AE0F7A9CE1FF8EF92CBA;
_idp_authn_lc_key
=8fa4ae6aa43d337ec48fbf355dd21bf1b81d8e3524787f2a6b675872c19f86ea
```

```
//RESPONSE - Redirect the User Agent to login page
HTTP/1.1 302 Found
Location https://idp.rest.org:443/idp/Authn/UserPassword
```

```
//REQUEST - Request the login page
GET /idp/Authn/UserPassword HTTP/1.1
Host: idp.rest.org
Cookie: JSESSIONID=60E013C4F9B6AE0F7A9CE1FF8EF92CBA;
_idp_authn_lc_key
=8fa4ae6aa43d337ec48fbf355dd21bf1b81d8e3524787f2a6b675872c19f86ea
```

```
//RESPONSE - Return to the login page
HTTP/1.1 200 OK
<!DOCTYPE html>
<html>
...
<body>
...
<form id="login" action="/idp/Authn/UserPassword" method="post">
```

```

<legend>
  Log in to restsp
</legend>

<section>
  <label for="username">Username</label>
  <input class="form-element form-field" name="j_username" type="text"
    value="">
</section>

<section>
  <label for="password">Password</label>
  <input class="form-element form-field" name="j_password" type="password"
    value="">
</section>

<section>
  <button class="form-element form-button"
    type="submit">Login</button>
</section>
</form>
...
</body>
</html>

//REQUEST - Submit principal credential
POST /idp/Authn/UserPassword HTTP/1.1
Host: idp.rest.org
Referer: https://idp.rest.org/idp/Authn/UserPassword
Cookie: JSESSIONID=60E013C4F9B6AE0F7A9CE1FF8EF92CBA; _idp_authn_lc_key
      =8fa4ae6aa43d337ec48fbf355dd21bf1b81d8e3524787f2a6b675872c19f86ea

j_username=joe&password=passw0rd

```

5. The Identity Provider issues SAML < Response> to the REST Server

When the input credential of the Principal is verified by the Identity Prover, a SAML < Response> that is sent by the Identity Provider is redirected to the REST Server by the User Agent.

The cookie DOCUMENTUM-SAML2-REQUEST-TOKEN is used to verify that the SAML < Response> is for the original Request.

```

//RESPONSE - Redirect the User Agent to the IdP SSO service
HTTP/1.1 302 Found
Set-Cookie _idp_session=MTAuMzIuOTQuMjE%3D%7CLQ%3D%3D%7CMjg1NWE5MmZiMWQyMjBjNGIxM2FiNzRhM
      mZiMTY1ZWQ1ZjI4NzQ4MTg5NjgwMDg1ZTk2OTU1YmQwNzM3MGJmZA%3D%3D%7CBN8tDDMNfj
      soPRouakNi6a%2Bg910%3D;
      Version=1;
      Path=/idp;
      Secure Location https://idp.rest.org:443/idp/profile/SAML2/Redirect/SSO

//REQUEST - Send request to the IdP SSo service
GET /idp/profile/SAML2/Redirect/SSO HTTP/1.1
Host: idp.rest.org
Referer: https://idp.rest.org/idp/Authn/UserPassword
Cookie: JSESSIONID=60E013C4F9B6AE0F7A9CE1FF8EF92CBA;
      _idp_authn_lc_key=8fa4ae6aa43d337ec48fbf355dd21bf1b81d8e3524787f2a6b675872c19f86ea;
      _idp_session=MTAuMzIuOTQuMjE%3D%7CLQ%3D%3D%7CMjg1NWE5MmZiMWQyMjBjNGIxM2FiNzRhMmZiMT
      Y1ZWQ1ZjI4NzQ4MTg5NjgwMDg1ZTk2OTU1YmQwNzM3MGJmZA%3D%3D%7CBN8tDDMNfjsoPRouakNi
      6a%2Bg910%3D

```

```

1 //RESPONSE
2 HTTP/1.1 200 OK
3 Set-
  Cookie: _idp_authn_lc_key=8fa4ae6aa43d337ec48fbf355dd21bf1b81d8e3524787f2a6b675872c19f8
  6ea; Version=1; Max-Age=0;
4 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
5   <head>
6     </head>
7   <body onload="document.forms[0].submit()">
8     <noscript>
9       <p>
10         <strong>Note:</strong> Since your browser does not support JavaScript,
11         you must press the Continue button once to proceed.
12       </p>
13     </noscript>
14
15     <form action="https&#x3a;&#x2f;&#x2f;restsp&#x3a;8443&#x2f;dctm-
  rest&#x2f;saml&#x2f;SSO" method="post">
16       <div>
17         <input type="hidden" name="RelayState" value="&#x2f;repositories&#x2f;REPO
  "/>
18
19         <input type="hidden" name="SAMLResponse" value="PD94bWwgdmVyc...zcG9uc2U+"
  />
20       </div>
21       <noscript>
22         <div>
23           <input type="submit" value="Continue"/>
24         </div>
25       </noscript>
26     </form>
27   </body>
28 </html>

```

```

1 //REQUEST
2 POST /dctm-rest/saml/SSO HTTP/1.1
3 Host: restsp:8443
4 Referer: https://idp.rest.org/idp/profile/SAML2/Redirect/SSO
5 Cookie: DOCUMENTUM-SAML2-REQUEST-TOKEN=a59abj125gb93ih3447ae97ie8b1a7d
6 RelayState=%2Frepositories%2FREPO&SAMLResponse=PD94bWwgdmVyc...UmVzcG9uc2U%2B

```

```

//REQUEST - send request to the REST Server SSO service
POST /dctm-rest/saml/SSO HTTP/1.1
Host: restsp:8443
Referer: https://idp.rest.org/idp/profile/SAML2/Redirect/SSO
Cookie: DOCUMENTUM-SAML2-REQUEST-TOKEN=a59abj125gb93ih3447ae97ie8b1a7d
RelayState=%2Frepositories%2FREPO&SAMLResponse=PD94bWwgdmVyc...UmVzcG9uc2U%2B

```

6. Validates the SAML < Response>

The REST Server validates the assertion. Once the validation is done, the principal name is extracted according to the attribute name specified in the `rest-api-runtime.properties` file.

7. Attempt to SAML Login Against the Content Server

The REST Server sends the principal name along with the SAML < Response> to the Content Server for SAML user login.

8. Returns SAML Login Session

When the principal name and the SAML < Response> are verified, Content Server returns a session for the Principal. At this point, the SAML authentication has been successfully completed.

9. Gets User Login Ticket

The Client Token is used after SAML authentication succeeds. At this point, the REST Server requests a login ticket for the authenticated user by using the SAML login session.

10. Returns User Login Ticket

The Content Server returns a user login ticket to the REST Server.

11. Produces a Client Token Cookie

Using the user login ticket, the REST Server produces a Client Token cookie for the User Agent.

12. Redirects to the Original REST Resource with the Client Token Cookie

When the User Agent has the Client Token cookie set, it can request the original target resource using Client Token authentication. This process clears the cookie DOCUMENTUM-SAML2-REQUEST-TOKEN as the SAML authentication completes.

```
//RESPONSE - the REST Server redirects the the User Agent to the original REST resource
HTTP/1.1 302 Found
Set-Cookie: DOCUMENTUM-SAML2-REQUEST-TOKEN=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT;
Path=/dctm-rest
Set-Cookie: DOCUMENTUM-CLIENT-TOKEN="3fVt7mVQ+ivzpYtp8CV70JLCWJfKiRfhy6w34NUYhaJfZWoxAG1M
sbY2LrkBkwp5sGykIT3MhSiGM9gdOfr8GeUjFvxDC1Zlah2gQHlz2KkQs3i6lcf7KQIcAr0xrl6as+2s+U8lFP
DaNbuA6Hp8P0ly/ODHaAdkLQJwQ7Ev0ozqdtDdBXyJxUyVZatsak5NzaIDgclBShk2xLeTbifuM/SL2T4zEtOv1
rsn4hQDkJJi9HmXCLQFuEIBP9b78/WAEAkPn9RsMQP9xn02NZBd0mckylwJxGLj9XWVASjDG6xjGvdWp6fMCvnt
732CyaIZmNCdD2vT9/GnK29bMspMImKVHrpRQ0lmhnzrrpbYXrXQB6g+3ZWHow1crnkNXtovKMCv7+6g68YWG
JjKQ4gjCz/SGLwMH1zRmEhorNsJBETZjglxEFYehXQUisdUkaZ2WaiM3hD1B96+UIWxL736JRXIA7bAwQ+JtP
9TS5g="; Version=1; Max-Age=10000; Expires=Fri, 04-Mar-2016 04:54:31 GMT;
Path=/dctm-rest; Secure; HttpOnly
Location: https://restsp:8443/dctm-rest/repositories/REPO

//REQUEST - request the original REST resource
GET /dctm-rest/repositories/REPO HTTP/1.1
Host: restsp:8443
Referer: https://idp.rest.org/idp/profile/SAML2/Redirect/SSO
Cookie: DOCUMENTUM-CLIENT-TOKEN="3fVt7mVQ+ivzpYtp...JRXIA7bAwQ+JtP9TS5g="
```

13 to 17

Steps 13 through 17 are typical Client Token ticket sign-on steps. The User Agent reuses the existing Client Token cookie to execute subsequent REST requests, until the cookie has expired.

```
//RESPONSE - return the REST resource
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<?xml version='1.0' encoding='UTF-8'?>
<repository
  xmlns="http://identifiers.emc.com/vocab/documentum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <id>1</id>
  <name>REPO</name>
```

```
<description></description>
<server>
  <name>REPO</name>
  <host>AQEDFSSQL20-0-1</host>
  <version xml:space="preserve">7.3.0000.0063   Win64.SQLServer</version>
  <docbroker>AQEDFSSQL20-0-1</docbroker>
</server>
<links>
  <link rel="self" href="https://restsp:8443/dctm-rest/repositories/REPO"/>...
  <link rel="http://identifiers.emc.com/linkrel/batches"
    href="https://restsp:8443/dctm-rest/repositories/REPO/batches"/>
  <link rel="http://identifiers.emc.com/linkrel/batch-capabilities"
    href="https://restsp:8443/dctm-rest/repositories/REPO/batch-capabilities"/>
  <link rel="http://identifiers.emc.com/linkrel/logoff"
    href="https://restsp:8443/dctm-rest/logout"/>
</links>
</repository>
```

Disable Redirecting for SAML SSO Initialization

In some cases, the User Agent does not want to be redirected to an Identity Provider after it makes a requests for a REST resource.

In step 1 a Request with the HTTP header `DOCUMENTUM-NO-SAML2-REDIRECT=true` can be used to disable redirection. By default, redirecting is enabled when the HTTP header `DOCUMENTUM-NO-SAML2-REDIRECT` is not in the Request.

Note: The HTTP header `DOCUMENTUM-NO-SAML2-REDIRECT` only works when there is a single Identity Provider metadata file configured.

1. Attempts to Request a REST Resource

Typically the repository resource is first to require authentication. At this point, the User Agent does not know which Identity Provider will be used to authenticate.

```
//REQUEST - initial repository resource request
GET https://restsp:8443/dctm-rest/repositories/REPO HTTP/1.1
DOCUMENTUM-NO-SAML2-REDIRECT: true
```

```
//RESPONSE - return Unauthorized HTTP status code and header Location with the URL for sending
SAML authentication request HTTP/1.1 401 Unauthorized
Set-Cookie: Path=/dctm-rest/; Secure; HttpOnly
WWW-Authenticate: SAML2 realm="com.emc.documentum.rest"
Location: https://idp.rest.org/idp/profile/SAML2/Redirect/SSO?SAMLRequest=jVLLbs...
sb4h8%3D&RelayState=%2Frepositories%2FREPO&SigAlg=http%3A%2F%2Fwww.w3.org%2
F2000%2F09%2Fxmldsig%23rsa-sha1&Signature=HSVv...%3D%3D
```

Once the User Agent gets the Response with a 401 status code, it can decide whether to continue the SAML authentication process. If yes, the User Agent can follow the URL in the *Location* HTTP header then all subsequent steps are same as the typical case shown.

SAML Based SSO with Multiple Identity Providers

When multiple Identity Provider metadata files are configured in the REST Server, the first two steps shown in the figure extend into several additional steps. The negotiation for the Identity Providers are also known as Identity Provider Discovery.

1. Attempts to Request a REST Resource

The repository resource is first to require authentication. At this point, the User Agent does not know which Identity Provider will be used for authentication.

```
//REQUEST - initial repository resource request
GET https://restsp:8443/dctm-rest/repositories/REPO HTTP/1.1
```

2. Determines Which Identity Providers to Use

The REST Server and client negotiate to determine which Identity Provider to use.

- a. First, the REST server redirects the User Agent to the SAML discovery URL.

The original request URL must be put into the *Referer* HTTP header during redirection. Otherwise, the URL of the resource that was originally requested is lost during the discovery process.

```
//RESPONSE - redirect the User Agent to the discovery service
HTTP/1.1 302 Found
Location https://restsp:8443/dctm-rest/saml/discovery?entityID=https%3A%2F%2Frestsp%3A8443%2Frest%2Fsaml%2Fmetadata&returnIDParam=idp
```

```
//REQUEST - send request to the discovery service with the HTTP header Referer;
the original URL is set in this header
GET /dctm-rest/saml/discovery?entityID=https%3A%2F%2Frestsp%3A8443%2Fdctm-rest%2Fsaml%2Fmetadata&returnIDParam=idp HTTP/1.1
Referer: https://restsp:8443/dctm-rest/repositories/REPO
```

- b. The REST Server returns the URLs of configured Identity Providers in the *Location* HTTP header for the User Agent to select.

```
//REQUEST - request the original REST resource
GET /dctm-rest/repositories/REPO HTTP/1.1
Host: restsp:8443
Referer: https://idp.rest.org/idp/profile/SAML2/Redirect/SSO
Cookie: DOCUMENTUM-CLIENT-TOKEN="3fVt7mVQ+ivzpYtp...JRXIA7bAwQ+JtP9TS5g="

//RESPONSE - return the available IdPs' URLs for SAML authentication
HTTP/1.1 200 OK
Location https://restsp:8443/dctm-rest/saml/login?disco=true&idp=http%3A%2F%2FAQEDFSWIN20-0-1.oauth.emc.com%2Fadfs%2Fservices%2Ftrust&original-url=https%3A%2F%2Frestsp%3A8443%2Fdctm-rest%2Frepositories%2FREPO
Location https://restsp:8443/dctm-rest/saml/login?disco=true&idp=https%3A%2F%2Fidp.rest.org%2Fidp%2Fshibboleth&original-url=https%3A%2F%2Frestsp%3A8443%2Fdctm-rest
```

```
%2Frepositories%2FREPO
```

- c. In this step, the User Agent selects one URL. Then the subsequent steps are same as those shown in the one Identity Provider example.

```
//REQUEST - send request to the selected IdP
GET /dctm-rest/saml/login?disco=true&idp=https%3A%2F%2Fidp.rest.org%2Fidp%2Fshibboleth&original-url=https%3A%2F%2Frestsp%3A8443%2Fdctm-rest%2Frepositories%2FREPO
```

Note: The DOCUMENTUM-NO-SAML2-REDIRECT HTTP header does not work for configuration with multiple Identity Provider metadata files.

Skip Identity Providers Discovery

When the User Agent does not want to trigger another discovery because it has already determined the Identity Provider, the User Agent can skip the discovery by appending a URL parameter `idp=idp-entity-id` to the URL of the REST resource requested. For example, the URL parameter `idp=https://idp.rest.org/idp/shibboleth` is added to skip the discovery process.

The entity ID must be encoded.

```
https://restsp:8443/dctm-rest/repositories/REPO?idp=https%3A%2F%2Fidp.rest.org%2Fidp%2Fshibboleth
```

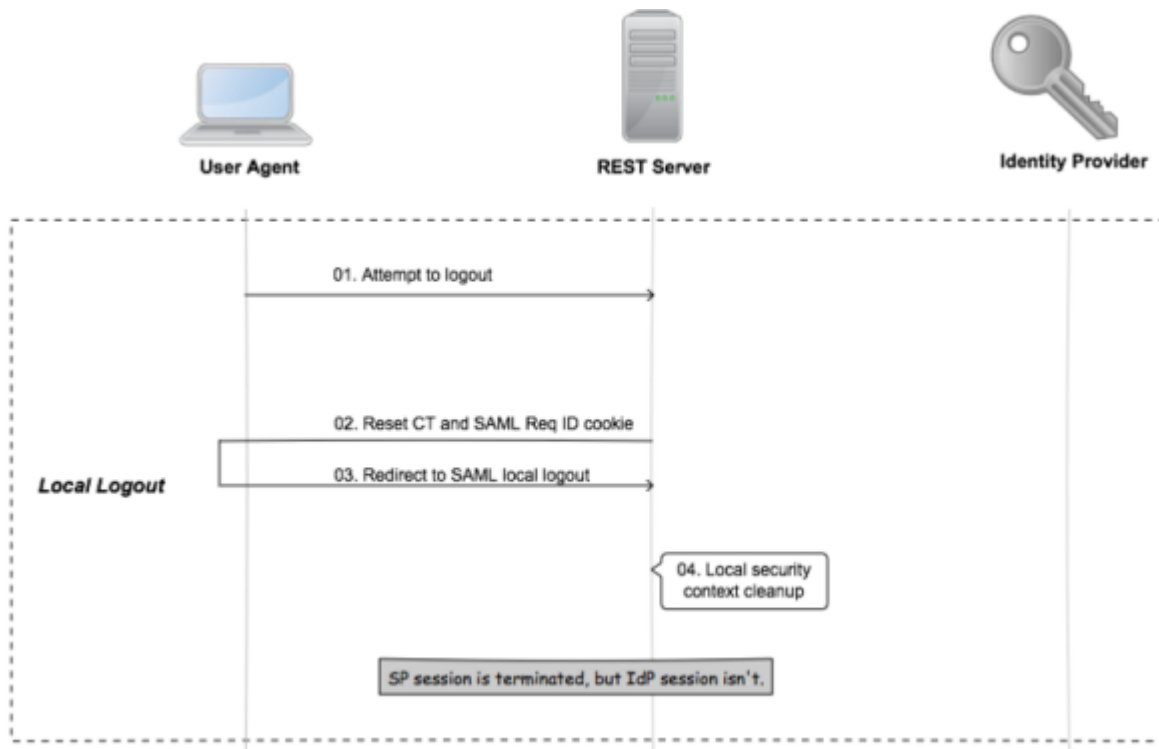
Documentum REST SAML Logout Workflow

Documentum REST SAML logout includes two steps in its cleanup operations:

1. It cleans up the Client Token cookie on User Agent
2. It terminates the SAML session on the REST Server initializing the logout (the sessions on the Identity Provider and other REST Servers are not affected).

This type of logout is known as a *local logout* because only the original REST Server has its session terminated.

The following figure shows the workflow:



The steps shown in the diagram are as follows:

1. Attempts to Logout

The User Agent sends a logout request to the REST Server. The logout URI is also defined by the common runtime property `rest.security.logout.url`.

```
//REQUEST - initiate the logout
GET /dctm-rest/logout HTTP/1.1
Host    restsp:8443
```

2. **a.** Resets the cookies and redirects to SAML local logout. At this point, the Client Token and SAML Request Token cookie are reset

```
//RESPONSE - clear the cookies and redirect to SAML logout URL
HTTP/1.1 302 Found
Set-Cookie    DOCUMENTUM-CLIENT-TOKEN=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT;
Path=/dctm-rest
Set-Cookie    DOCUMENTUM-SAML2-REQUEST-TOKEN=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT;
Path=/dctm-rest
Location      https://restsp:8443/dctm-rest/saml/logout?local=true
```

- b.** Redirects to the SAML logout URL

```
//REQUEST - send request to the SAML logout URL
GET /dctm-rest/saml/logout?local=true HTTP/1.1
Host: restsp:8443
```

3. Local Security Context Cleanup

At this point, the security context is cleaned up and the User Agent is redirected to the successful logout URL

```
//RESPONSE - finish logout and redirect to successful logout URL
HTTP/1.1 302 Found
```

```
Location: https://restsp:8443/dctm-rest/services

//REQUEST - send request the successful logout URL
GET /dctm-rest/services HTTP/1.1
Host: restsp:8443

//RESPONSE - return the content of the successful logout URL
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
<?xml version='1.0' encoding='UTF-8'?>
  <resources xmlns="http://identifiers.emc.com/vocab/documentum"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <resource rel="http://identifiers.emc.com/linkrel/repositories">
      <link href="https://restsp:8443/dctm-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>
    <resource rel="about">
      <link href="https://restsp:8443/dctm-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>
  </resources>
```

Configuration

Documentum REST SAML based SSO requires configuration on the following servers:

- REST Server
- Identity Provider Server
- Content Server

REST Server

1. Create a REST Server certificate and import it into the server JRE trust store. In the following sample, it is shown as `path-of-the-saml-cerficate-file.jks`.
2. Update `server.xml` of the application server where Documentum Platform REST Services is deployed to enable *HTTPS*. The following sample shows you how to enable *HTTPS* on the Tomcat server.

```
<Connector port="8443"
    protocol="org.apache.coyote.http11.Http11Protocol"
    maxThreads="150"
    SSLEnabled="true"
    scheme="https"
    secure="true"
    clientAuth="false"
    keystoreFile="path-of-the-saml-certificate-file.jks"
    keystorePass="password"
    sslProtocol="TLS"/>
```

3. Edit `rest-api-runtime.properties` in the `dctm-rest.war` as shown . The `rest-api-runtime.properties.template` file, which is in the same location, has all the details of the configuration parameters that are available. The metadata file of the Identity Provider Server should be downloaded to the local path of the REST server installation after the Identity Provider Server is configured. In the following sample, the path of this metadata file is shown as `path-of-the-idp-metadata-file`.

Note: The runtime property `rest.security.saml2.user.attributes` specifies the user principal attributes in the SAML assertion, This runtime property is where Documentum REST Server gets the principal name. The availability of these attributes is configured by Identity Provider's attribute release policy.

In the following sample, the Identity Provider's attribute release policy is configured by the ADFS 2.0 attribute `http://schemas.microsoft.com/ws/2008/06/identit.../windowsaccountname` Multiple attributes can be specified in the runtime property. Since SAML SSO is integrated with Documentum Client Token, you may have to customize the Client Token configuration. You can find the corresponding section for the Client Token configuration in the same runtime properties file.

```
#enable Client Token and SAML2 authentication schema
rest.security.auth.mode=ct-saml2

#specify the java key store file
rest.security.saml2.ks.file==path-of-the-saml-certificate-file.jks

#specify the password of the java key store
rest.security.saml2.ks.password=password

#specify the alias of key entry used by the SAML Service Provider to sign the SAML message
rest.security.saml2.ks.entry.alias=alias

#specify the password of the key entry used by the SAML Service Provider to sign the SAML message
rest.security.saml2.ks.entry.password=password

#specify the HTTP method used to send SAML request to the Identity Provider
rest.security.saml2.request.binding==HTTP-Redirect

#specify the metadata files of the Identity Providers
rest.security.saml2.idp.metadata.files=path-of-the-idp-metadata-file

#specify the attributes used to extract principal names from the SAML response
rest.security.saml2.user.attributes==http://schemas.microsoft.com/ws/2008/06/identity/claims/windowsaccountname

#specify the cookie timeout of SAML request token
rest.security.client.saml2.timeout=300
```

4. Edit `dfc.properties` in the `dctm-rest.war` according to the Content Server configuration.

5. Deploy the `dctm-rest.war`
6. Get the metadata file of the REST Server. The REST server metadata file is also used to setup the Identity Provider trust.

```
GET https://<tomcat-server-hostname>:8443/dctm-rest/saml/metadata
```

Note: To simplify the configuration, service communication and token-signing use the same certificate. Alternatively, two different certificates can be used.

Identity Provider Server

Here, *Microsoft ADFS 2.0* is used as the Identity Provider Server

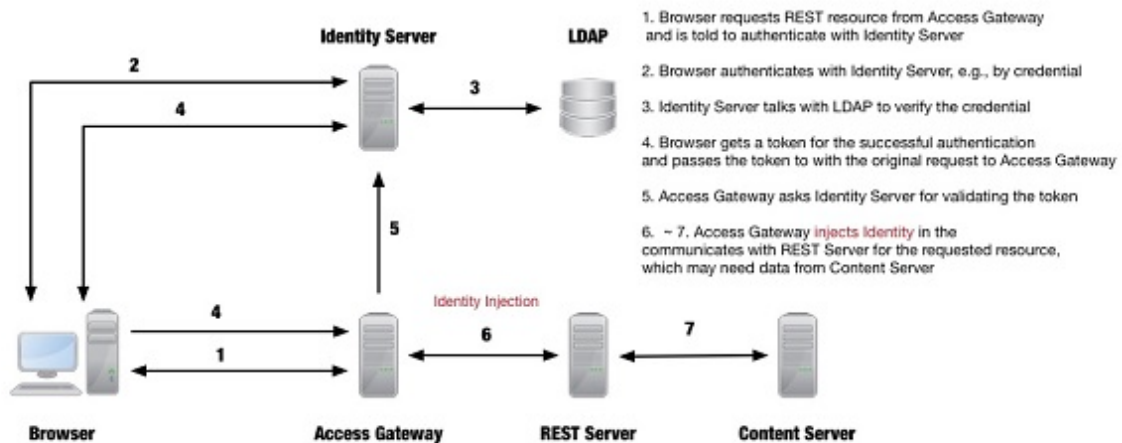
For information on ADFS 2.0, see the documentation provided by [Microsoft](#).

Pre-authenticated Authentication

You may have deployed your own claim-based authentication infrastructure prior to deploying Documentum REST. The customer authentication scheme is not supported by REST or Content Server out of the box. The customer authentication scheme performs the authentication ahead of Documentum REST and returns an HTTP header or cookie along with the HTTP Request to the REST servlet. The HTTP header or cookie serves as the plain-text principal (without password) authorization of the authenticated user. The Pre-authenticated authentication scheme can be used to integrate Documentum Platform REST Services with Tivoli WebSEAL, NetIQ Access Manager or other similar Web Access Management (WAM) systems.

This security requirement can be fulfilled by using a WAM solution, which protects networks, applications, and data while simultaneously providing quick and efficient access to an increasing number of authorized users.

Below is a typical topology of a WAM architecture, which describes a typical scenario.



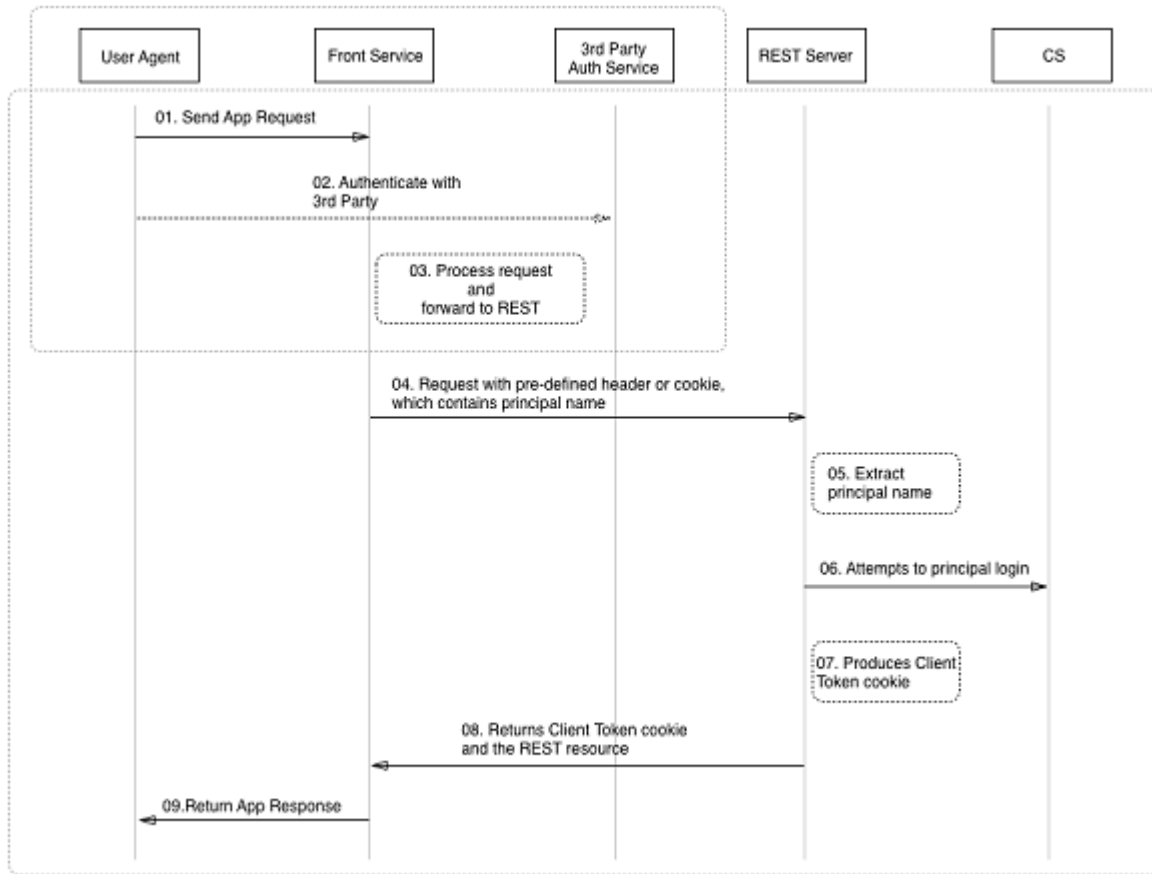
Documentum Platform REST Services can provide you with a pre-authenticated authentication scheme, where customers only need to specify the header name and the regular-expression to parse the header value. Documentum Platform REST Services uses privileged DFC (principal mode) or *trust.properties* to get Content Server sessions for the user operation.

Note: Details of the privileged DFC instance and configuration of *trust.properties* can be found in the *Content Server Fundamentals Guide*.

Workflow

Below is a diagram that shows the workflow of pre-authenticated login. The following are constraints for this authentication extension:

1. The User Agent cannot communicate with the REST server. Instead it only communicates with the Front Service.
2. The Front Service authenticates the User Agent using a 3rd Party Authentication Service. The Front Service does not authenticate the User Agent with the REST server or the Content Server (CS).
3. The Front Service communicates with the REST server according to a predefined schema, such as http headers or a cookie.
4. The REST server extracts the principal name based on a predefined pattern, and uses the principal name to attempt a principal login to Content Server (CS).



Feature Highlights

Runtime Configuration for Pre-authenticated Login

```

#####
##          6. General Security Configuration          ##
#####
# .....
# .....

# preauth
#   - Support pre-authentication without Client Token cookie supported
# ct-preauth
#   - Support pre-authentication with Client Token cookie supported
# The default mode is basic.
# For more information, see the Documentum Platform REST Services Reference Guide.
#
rest.security.auth.mode=

#####
##          Security Configuration for Pre-authenticated Login          ##
#####
#
# This section contains the security configuration for Pre-authenticated Login to parse
# the principal from the Request by the front service. Configuration in this section work

```

```
# only when preauth scheme is set in the General Security Configuration section.
# More information is found in this guide, including details on the configuration steps.
#
#
# Specifies the names of headers which contain the principal name sent by the front service.
# The value must be a set of comma separated header names without spaces. The value is case
# sensitive. The REST server iterates through the headers of the Request and retrieves the
# first one that is contained in the set of cookie names defined here. For example, assuming
# [FrontServiceToken1,FrontServiceToken2] is specified here. The HTTP Request has header
# "FrontServiceToken1=principal_name". Therefore "principal_name" is extracted to attempt
# the DFC principal login. When the value is not specified, the REST server won't extract the
# principal name from the headers. With regards to extracting the principal name, the priority
# of headers is higher than the priority of settings within cookies.

rest.security.preauth.header.names=
# Specifies the names of cookies which contain the principal name sent by the front service.
# The value must be a set of comma separated cookie names without spaces. The value is case sensitive.
# The REST server iterates through the cookies of the Request and retrieves the first one
# that is contained in the set of cookie names defined here. For example,
# [FrontServiceToken1,FrontServiceToken2] is specified here. HTTP request has cookie
# "FrontServiceToken2=principal_name". The "principal_name" is extracted to attempt DFC
# principal login. When the value is not specified, the REST server won't extract principal
# name from cookie. With regards to extracting the principal name, the priority
# of headers is higher than the priority of cookies.
#
rest.security.preauth.cookie.names=
# Specified the regular expression patterns to extract principal name from the distinguished
# name by the front service. The value must be dual-pound (##) separated regular expressions
# without spaces. The value is case sensitive.
# For example, "uid=(.*)?,ou=ecd,dc=emc,dc=com##(.*)" is specified here. If distinguished
# name by the front service is "uid=joe,ou=ecd,dc=emc,dc=com", first regular expression will
# work and "joe" is parsed out. When the distinguished name by the front service is "joe", the
# second regular expression works and "joe" is extracted.
# By default, the regular expression is "(.*)" to match the whole string.
#
rest.security.preauth.principal.patterns=
```

Set Documentum Client Token Cookie or Not

The Documentum (DCTM) Client Token is one of the authentication schemes that are supported by Documentum Platform REST Services. When the DCTM Client Token cookie is set, the REST Server can serve the Front Service by using the Client Token authentication schema. When the DCTM Client Token is not sent back, the REST server has to get the session by using the principal login for all subsequent requests.

REST API Runtime Properties

```
#Client Token won't be sent back
rest.security.auth.mode=preauth
```

Principal in Headers or Cookie

The Front Service sends the distinguished name that contains the principal by header or cookie. The REST Server processes the header first, then it processes the cookie. The names are case sensitive.

Principal Pattern

Documentum Platform REST Services applies the patterns specified to the distinguished name sent by the Front Service and extracts the principal. Here are the two patterns:

```
# Two patterns separated by double pound ## characters
rest.security.preauth.principal.patterns=uid=(.*)?,
    ou=ecd,dc=emc,dc=com##cn=(.*)?,ou=pc,dc=dell,dc=com
```

Multiple Authentication Schemes

Documentum Platform 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`.

The following authentication schemes combinations 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`)
- 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 succeeds, 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

If 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 5-6. 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 5-7. 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 5-8. Matching basic Authorization header

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

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

Example 5-9. Matching negotiate Authorization header

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

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

Example 5-10. 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 5-11. 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 5-12. 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 5-13. 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 5-14. Matching basic Authorization header

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

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

Example 5-15. 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 5-16. 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 doabase space01.\""
}
```

Example 5-17. Matching CAS ticket with bad credential (no redirect)

```
// Request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ++bad++credential HTTP/1.1
DOCUMENTUM-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 5-18. Matching CAS ticket with bad credential (redirect)

```
// Request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ++bad++credential HTTP/1.1
DOCUMENTUM-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 <protocol>
```

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 <protocol>
```

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 5-19. 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 5-20. Reverse Proxy Configuration on Apache to improve concurrent access performance

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

Client Token

A Client Token (Documentum Client Token) is an encrypted proof token that represents the identity of an authenticated user who can access Documentum REST Services. It is not a standalone authentication scheme, it is integrated with other authentication schemes and used together with those authentication schemes to authenticate REST requests. Typically, a Client Token is transferred between the REST client and the REST server as HTTP cookies. These HTTP cookies have the cookie name *DOCUMENTUM-CLIENT-TOKEN*.

When Client Token is enabled, the REST server generates a Client Token cookie after a successful authentication, and sends it back to the REST client in the Response. The purpose of this client token is to provide an authenticated REST client with a temporary token, that expires, and can be used to access the REST server without having to validate user credentials, or negotiate SSO tokens each and every time. The Client Token cookie has no session state stored on the REST server. Therefore, it can work in clustered environments.

A client token cookie is encrypted and validated by the REST server. The REST client is not responsible for saving or decrypting 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 must be repeated.

In the current release, the Client Token can be used with the following existing authentication schemes:

- HTTP Basic authentication
- Kerberos authentication
- CAS authentication

The use of a client token in this case is mandatory.

- SAML 2.0 authentication

The use of a client token in this case is mandatory.

- Pre-authentication authentication

For more information on how to develop your own authentication scheme and integrate it with the Client Token, see .

The following example describes the authentication negotiation workflow between the REST 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 with the Kerberos domain controller 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

To see if a Client Token can be enabled in an authentication scheme, please refer to your chosen authentication scheme, or to see <dictm-rest.war>/WEB-INF/classes/rest-api-runtime.properties .template. For example, the security configuration in Documentum Platform REST Services disables

client tokens for Kerberos and CAS authentication by default. 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 Platform REST Services supports the following expiration policies for the client token cookie:

Policy	Description
<code>com.emc.documentum.rest.security.ticket.impl.HardTimeoutExpirationPolicy</code>	<p>The client token expires after a specified duration.</p> <p>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.</p>
<code>com.emc.documentum.rest.security.ticket.impl.TolerantTimeoutExpirationPolicy</code> (default)	<p>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:</p> <ul style="list-style-type: none"> • 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.
<code>com.emc.documentum.rest.security.ticket.impl.TouchedTimeoutExpirationPolicy</code>	<p>The client token expires after a specified duration. The REST server issues new client tokens under certain conditions. For details, see the following:</p> <ul style="list-style-type: none"> • 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.

Policy	Description
	<ul style="list-style-type: none"> If the REST client sends a request after the duration, the REST server rejects the client token, and the client has to authenticate again.

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 Platform REST Services is deployed to achieve better performance:

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

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.

Logoff Success Handling

By default, the sign-out process returns a 204 response without a response body. In a real world case, the sign out usually must redirect the user to another page. In 7.3, we will expose another runtime property for users to configure the logout success URL. The redirection happens only when the logout succeeds. The runtime property below allows to customize logout success URL.

```
# Specifies the URL to redirect to after the successful logout.
# This property can be used when a Client Token cookie is used# for authentication.
# It can be a complete URL or a relative URL. When it is a relative URL, the server
# inserts the Request base and context path to the URL path.
# When it is not specified, the server returns no content for a successful logout.
# Examples:
# - http://another-server/goodbye
# - /services
rest.security.logout.success.url=
```

By convention, only the POST HTTP method is typically supported for signing out of most web applications. Allowing for the integration with other authentication schemes, like SAML, by using GET to communicate with REST Services for the process of logout. Another runtime property is used to specify which HTTP methods are supported for sing out. When the Request method is not supported, the status code 405 is returned with an error message in the Response. The runtime property below allows you to customize logout HTTP method.

```
#Specifies supported HTTP methods for sign out. Multiple values separated by comma,
# are supported as below.
```

```
#Examples:
#rest.security.logout.supported.methods=GET,POST
#Note: the HTTP methods specified in String format are not case sensitive,
# meaning 'get,POST', 'GET,post' or 'Get,Post' all make sense.
#when not specified, the default values GET,POST would be applied.
rest.security.logout.supported.methods=
```

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 cryptographic algorithms that are used for 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 after modifying the `rest.security.crypto.provider`.
- When you deploy Documentum Platform 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 the `rest-api-runtime.properties` file:

- `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 the security providers in Documentum Platform 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 Platform REST Services when the two providers share the same name. The default value is `false`, meaning that Documentum Platform REST Services uses the security provider registered in the web application. The default setting is recommended.
- Documentum Platform REST Services client token encryption is supported by the **Java 7** cryptographic cipher.

For more information, see [Class Cipher](#).

- Documentum Platform REST Services also supports third party encryption providers such as **RSA BSAFE**, **Crypto-J**, and **Bouncy Castle**.

For more information, see [RSA BSAFE Version Spaces](#) and [The Legion of the Bouncy Castle](#).

Algorithms with a Key Size Larger than 128 bits

The default version of Java Cryptography Extension (JCE) policy files bundled in the JDK 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 Browser Login Forms 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 some web browsers to display a login dialog box prompt for re-authentication. However, if you design your own login screen, it could be blocked by the one the web browser prompts. To bypass a login dialog box, the Documentum REST server can append a suffix to the authentication scheme in the *WWW-Authenticate* header. This makes web browsers not recognize the scheme and so that 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. Optionally, on the REST server, you can customize the *WWW-Authenticate* response header for HTTP status 401 with the `rest.api.unauthorized.response.scheme.suffix` in the `rest-api-runtime.properties` runtime property setting.

Example 5-21. Request

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

Example 5-22. Runtime Property Setting

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

Example 5-23. Response

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

Deploying to Docker Containers

Traditionally, Web services software is delivered by vendors to customers as WAR files. The customer has to prepare their development and production environment to make the web service deployment work as expected. This preparation is time consuming and complex. Often, many unexpected issues arise, which break the deployment. These deployment issues must be resolved before the deployment can resume. Updating a web service deployment is also difficult for both vendors and customers, because development and production environments are similar but rarely the same. More time consuming work must be done to get a successful update without causing new issues. Docker solves these problems and more.

Overview

Docker is an open source containerization platform that's used for developing, deploying, testing, and running your applications. Packaging your application along with all of its dependencies in a Docker Image is known as **Dockerizing** your application. Since the Docker image contains all of your application's dependencies, Dockerizing your applications allows you to separate your applications from their dependencies on hardware or software, which allows you to treat your machine environment like a managed application. Docker helps simplify the process of deploying code, running or testing code, and shipping code.

All REST services are stateless, which makes them ideal candidates for Docker Containers. Using Docker Containers allows you to safely develop and test your REST services application in a predictable and stable way.

Docker Containers make updating or upgrading **Documentum Platform REST Services** easier and more efficient than working with traditional WAR file updates or upgrades. You can use Docker to patch your application image, along with all of its required dependencies. The same is true for when you want to deliver a new version of your REST services application to customers. Running a container from a Docker application image is usually all that's necessary to update or upgrade your REST services application.

In addition to the above advantages, using the **Docker-Compose** and **Swarm** tools make it easier to set up application clusters, and deploy other sophisticated use scenarios. When your environment is ready, it can be easily scaled, migrated, and shipped. Docker employs native security, simple sharing capabilities, integration with clouds services, such as AWS or Azure, and it supports many orchestration frameworks too.

Your REST web service application, along with all of its dependencies, can be packaged into a single deployable Docker Image. The same Docker Container that uses this image can be run on various

operating systems and environments, and because everything that your application requires is included in your lightweight Docker Container, your application runs exactly the same on each environment.

For more information, see [Docker](#).

Docker Setup

The Linux operating system is natively supported by Docker. However, you can use the Docker Toolbox to install and setup a Docker environment on Windows or Mac OS X.

For more information, see [Docker Engine](#).

Docker Image of Documentum Platform REST Services

To begin using Docker, you must have **Documentum Platform REST Services** running with Content Server (CS) as its backend. The Documentum Foundation Classes (DFC) client within **Documentum Platform REST Services** uses the settings in the `dfc.properties` file to communicate with CS. Your deployment of CS can be on a physical machine, a virtual machine, or a Docker Container. As long as **Documentum Platform REST Services** can communicate with CS, any of the above deployments will work. For example, when the **Documentum Platform REST Services** and CS are in two Docker Containers on the same Docker host, you can use the internal IP of the Docker network or the hostname specific to the Docker network to access the CS.

To Dockerize your web application, you must include all of its dependencies when you create the Docker image. The following table shows information about the Docker image, including the typical dependencies (OS, Web Container, and JRE), its configuration, and its features.

OS	Web Container	JRE	Ports	Logs Persistence Support	Support for SSL
CentOS 7 / Ubuntu 14.04 and 16.04	Apache Tomcat 8	JRE of OpenJDK 1.8	8080(http) / 8443(https) / 8009(ajp)	YES	YES

Dockerizing your Documentum Platform REST Services Application

You must create a `Dockerfile`, which defines how to build a Docker Image. When the Docker image is ready for use, you can execute Docker commands against it to run a Docker Container from the image. You must use the `docker run` command to launch your Docker Container.

Note: The `entrypoint.sh` launch script only manipulates the REST related configuration files.

Below is an example of a Dockerfile. As you can see, this file creates the image with the JRE, Tomcat, REST, volume and port definitions, and then it defines the script to start the REST services. Pay particular attention to the following configurable environment variables: TOMCAT_MAJOR, TOMCAT_VERSION, and DCTM_REST_URL.

Example 6-1. Launch script

The launch script manipulates two things:

1. The REST configuration files, such as dfc.properties, rest-api.properties, and others as needed.
2. The server.xml file for Tomcat.

```
#!/bin/bash
# Improve tomcat startup performance, http://openjdk.java.net/jeps/123
java_security=$(find /usr/lib/jvm | grep jre | grep java.security$)
if grep -q securerandom.source=file:/dev/random ${java_security}; then
    sed -i -e "s/securerandom.source=file:/dev/random/securerandom
        .source=file:/dev/\.\.\./urandom/" ${java_security}
elif grep -q securerandom.source=file:/dev/urandom ${java_security}; then
    sed -i -e "s/securerandom.source=file:/dev/urandom/securerandom
        .source=file:/dev/\.\.\./urandom/" ${java_security}
fi

# Apply configuration files from volume
cp -R ${CONFIG_DIR}/* ${DCTM_REST_HOME}/WEB-INF/classes/

# Add Tomcat SSL configuration in server
if [ -n "${COMM_KEYSTORE_FILE}" ] && [ -n "${COMM_KEYSTORE_PWD}" ]; then
    echo "Setting Tomcat SSL/TLS connector: key store file and password..."
    sed -i '/$!;!--$/N;/Connector port="8443"/{N;N;N;s|(. *$!;!--\n)\|(.*)\|(\|>\n.*-->)\|
        \|1\2 keystoreFile="'${COMM_KEYSTORE_FILE}'" keystorePass="'${COMM_KEYSTORE_PWD}'"\3|}'
        ${CATALINA_HOME}/conf/server
else
    echo "Communication keystore file or password is not specified..."
fi

if [ -n "${COMM_KEY_ALIAS}" ] && [ -n "${COMM_KEY_PWD}" ]; then
    echo "Setting Tomcat SSL/TLS connector: key alias and password..."
    sed -i '/$!;!--$/N;/Connector port="8443"/{N;N;N;s|(. *$!;!--\n)\|(.*)\|(\|>\n.*-->)\|
        \|1\2 keyAlias="'${COMM_KEY_ALIAS}'" keyPass="'${COMM_KEY_PWD}'"\3|}'
        ${CATALINA_HOME}/conf/server
else
    echo "Communication key alias or password is not specified..."
fi

if [ -n "${COMM_KEY_STORE_TYPE}" ]; then
    echo "Setting Tomcat SSL/TLS connector: keystore type..."
    sed -i '/$!;!--$/N;/Connector port="8443"/{N;N;N;s|(. *$!;!--\n)\|(.*)\|(\|>\n.*-->)\|
        \|1\2 keystoreType="'${COMM_KEY_STORE_TYPE}'"\3|}' ${CATALINA_HOME}/conf/server
else
    echo "Communication keystore type is not specified..."
fi

# Uncomment the Tomcat SSL configuration to make it effect
if [ -n "${COMM_KEYSTORE_FILE}" ] || [ -n "${COMM_KEYSTORE_PWD}" ] ||
    [ -n "${COMM_KEY_ALIAS}" ] || [ -n "${COMM_KEY_PWD}" ] || [ -n
        "${COMM_KEY_STORE_TYPE}" ]; then
    echo "Enable Tomcat SSL/TLS configuration..."
    sed -i '/$!;!--$/N;/Connector port="8443"/{N;N;N;s|. *$!;!--\n\|(.*)\|>\n.*-->|\1|}'
        ${CATALINA_HOME}/conf/server
else
    echo "Tomcat SSL/TLS configuration is not enabled..."
fi
```

```
fi

# Start Tomcat
${CATALINA_HOME}/bin/catalina.sh run
```

Building the Docker Image

There are only two files needed to build the REST image:

- Dockerfile
- entrypoint.sh

Build the Docker Image

1. Make a build folder and put the building files in this folder.

```
mkdir ~/building
cd ~/building
cp <...>/Dockerfile ./
cp <...>/entrypoint.sh ./
```

2. The building process automatically retrieves the binaries for **Tomcat**, **JRE**, and **REST**. The configurable variables are listed in the following table:

Configurable Variable	Description
TOMCAT_MAJOR	Tomcat major version, used to create TOMCAT_TGZ_URL
TOMCAT_VERSION	Tomcat version, used to create TOMCAT_TGZ_URL
DCTM_REST_URL	Download URL of Documentum Platform REST WAR file

3. Run the Docker command to build the image:

```
docker build -t dctm-rest .
```

Runtime Configuration

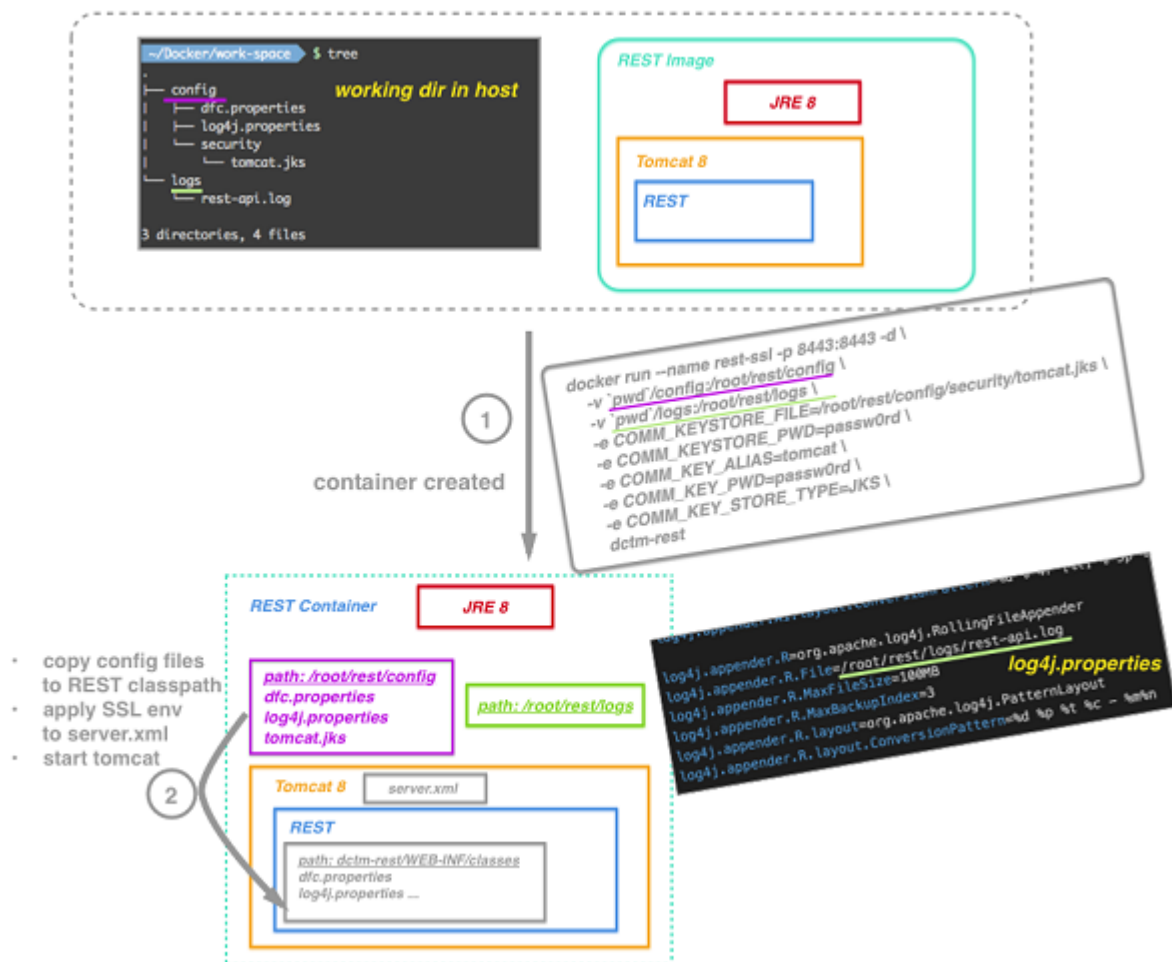
It is possible to apply configuration setting by using container environment variables, however this is not recommended for **Documentum Platform REST Services**. The configuration files to use include `rest-api-runtime.properties` and `dfc.properties`, which allow you set many options. You can mount a volume with the configuration file from the host to the specific path of the

Docker Container. When you execute the `docker run` command, the following two phases start within the REST container:

- The Docker Container is created. Volumes are mounted and environment variables are put into affect.
- The launch script runs and does 3 things:
 1. It copies your application configuration files to the `CLASSPATH` of Documentum Platform REST Services, which for Tomcat is `TOMCAT_ROOT/webapps/dctm-rest/WEB-INF/classes`.
 2. Applies the configuration settings according to environment variables.
 3. Launches the Tomcat server.

The following image illustrates the launch script process:

Figure 1. The Launch Script Process



A host folder can be mounted to a specific container folder so that the REST log files can be saved to the host, regardless of whether the container has been stopped or removed. The `log4j.properties` file directs the log file to this folder, in the above image you can see its path as `/root/rest/logs`.

To enable support for SSL, a keystore file can be mounted from the host to the container. Several environment variables define the keystore location, its type, password, and the certificate name

/password key value pair. The launch script applies these variables to the Tomcat server for SSL configuration.

Run the Container

The image can be run in different ways to accommodate various use scenarios. A basic use scenario is where the `dfc.properties` file is only configured to communicate with the Content Server. Here are the steps used in this scenario:

1. Make a working directory.

```
mkdir -p ~/rest/config
cd ~/rest
```

2. Prepare the `dfc.configuration` file to use *CS1*.

```
vi ~/rest/config/dfc.properties
```

3. Start the **Documentum Platform REST Services** container.

Within the command, the host folder `~/rest/config` is mapped to the container folder `/root/rest/config`. The launch script, `entrypoint.sh`, moves the configuration files from `/root/rest/config` to the class path location of the web container.

```
docker run --name rest -p 8080:8080 -d \
  -v `pwd`/config:/root/rest/config -v
  `pwd`/logs:/root/rest/logs dctm-rest
```

Here's a listing of the parameters used:

Parameter	Description
<code>--name rest</code>	The container name, which is rest
<code>-p 8080:8080</code>	Maps the host port 8080 to the container port 8080
<code>-d</code>	Run the container as daemon
<code>-v `pwd`/config:/root/rest/config</code>	Map the host folder <code>`pwd`/config</code> to the container folder <code>/root/rest/config</code>
<code>-v `pwd`/logs:/root/rest/logs</code>	Map the host folder <code>`pwd`/logs</code> to the container folder <code>/root/rest/logs</code> so that the REST log files are saved on the host

4. Verify the REST services.

```
curl http://dmadmin:password@localhost:8080/dctm-rest/repositories/REPO
{
  "id":1,"name":"REPO","description":"","servers":[{"name":"REPO","host":"sam1cs",
"version":"7.3.0000.0135 Win64.SQLServer","docbroker":"sam1cs"}],
"links":[{"rel":"self","href":"http://localhost:8080/dctm-rest/repositories/REPO"},
{"rel":"http://identifiers.emc.com/linkrel/cabinets",
"href":"http://localhost:8080/dctm-rest/repositories/REPO/cabinets"},
{"rel":"http://identifiers.emc.com/linkrel/checked-out-objects",
"href":"http://localhost:8080/dctm-rest/repositories/REPO/checked-out-objects"},
{"rel":"http://identifiers.emc.com/linkrel/current-user",
"href":"http://localhost:8080/dctm-rest/repositories/REPO/currentuser"}],
```

```

    {"rel": "http://identifiers.emc.com/linkrel/current-user-preferences",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/currentuser-preferences"},
    {"rel": "http://identifiers.emc.com/linkrel/users",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/users"},
    {"rel": "http://identifiers.emc.com/linkrel/groups",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/groups"},
    {"rel": "http://identifiers.emc.com/linkrel/formats",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/formats"},
    {"rel": "http://identifiers.emc.com/linkrel/network-locations",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/network-locations"},
    {"rel": "http://identifiers.emc.com/linkrel/relations",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/relations"},
    {"rel": "http://identifiers.emc.com/linkrel/relation-types",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/relation-types"},
    {"rel": "http://identifiers.emc.com/linkrel/types",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/types"},
    {"rel": "http://identifiers.emc.com/linkrel/aspect-types",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/aspect-types"},
    {"rel": "http://identifiers.emc.com/linkrel/dql",
     "hreftemplate": "http://localhost:8080/dctm-rest/repositories/REPO/{?dql}"},
    {"rel": "http://identifiers.emc.com/linkrel/search",
     "hreftemplate": "http://localhost:8080/dctm-rest/repositories/REPO/search{?collections,
       facet, include-total, inline, items-per-page, locations, object-type, page, q, sort,
       timezone, view}"},
    {"rel": "http://identifiers.emc.com/linkrel/saved-searches",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/saved-searches"},
    {"rel": "http://identifiers.emc.com/linkrel/search-templates",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/search-templates"},
    {"rel": "http://identifiers.emc.com/linkrel/acls",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/acls"},
    {"rel": "http://identifiers.emc.com/linkrel/batches",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/batches"},
    {"rel": "http://identifiers.emc.com/linkrel/batch-capabilities",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/batch-capabilities"}]
  }

```

5. Modify the configuration file `dfc.properties` to use `CS2`. You can modify the file in the container or the host, either will work.

```
// Modify the file in container
docker exec -it rest vi /root/rest/config/dfc.properties
```

OR

```
// Modify the file in host
vi ~/rest/config/dfc.properties
```

6. Restart the container.

```
docker restart rest
```

```

curl http://dmadmin:password@localhost:8080/dctm-rest/repositories/REPO
{
  "id": 5, "name": "REPO", "description": "", "servers": [{"name": "REPO", "host": "RESTCS72GA",
    "version": "7.2.0000.0155 Win64.SQLServer", "docbroker": "RESTCS72GA"}],
  "links": [{"rel": "self", "href": "http://localhost:8080/dctm-rest/repositories/REPO"},
    {"rel": "http://identifiers.emc.com/linkrel/cabinets",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/cabinets"},
    {"rel": "http://identifiers.emc.com/linkrel/checked-out-objects",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/checked-out-objects"},
    {"rel": "http://identifiers.emc.com/linkrel/current-user",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/currentuser"},
    {"rel": "http://identifiers.emc.com/linkrel/current-user-preferences",
     "href": "http://localhost:8080/dctm-rest/repositories/REPO/currentuser-preferences"},
    {"rel": "http://identifiers.emc.com/linkrel/users",

```

```

    "href": "http://localhost:8080/dctm-rest/repositories/REPO/users"},
    {"rel": "http://identifiers.emc.com/linkrel/groups",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/groups"},
    {"rel": "http://identifiers.emc.com/linkrel/formats",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/formats"},
    {"rel": "http://identifiers.emc.com/linkrel/network-locations",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/network-locations"},
    {"rel": "http://identifiers.emc.com/linkrel/relations",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/relations"},
    {"rel": "http://identifiers.emc.com/linkrel/relation-types",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/relation-types"},
    {"rel": "http://identifiers.emc.com/linkrel/types",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/types"},
    {"rel": "http://identifiers.emc.com/linkrel/aspect-types",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/aspect-types"},
    {"rel": "http://identifiers.emc.com/linkrel/dql",
    "hreftemplate": "http://localhost:8080/dctm-rest/repositories/REPO{?dql}"},
    {"rel": "http://identifiers.emc.com/linkrel/search",
    "hreftemplate": "http://localhost:8080/dctm-rest/repositories/REPO/search{?collections,
        facet, include-total, inline, items-per-page, locations, object-type, page, q, sort,
        timezone, view}"},
    {"rel": "http://identifiers.emc.com/linkrel/saved-searches",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/saved-searches"},
    {"rel": "http://identifiers.emc.com/linkrel/search-templates",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/search-templates"},
    {"rel": "http://identifiers.emc.com/linkrel/acls",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/acls"},
    {"rel": "http://identifiers.emc.com/linkrel/batches",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/batches"},
    {"rel": "http://identifiers.emc.com/linkrel/batch-capabilities",
    "href": "http://localhost:8080/dctm-rest/repositories/REPO/batch-capabilities"}}
}

```

7. Stop the container.

```
docker stop rest
```

Note: All the configuration files and directories in the host folder `~/rest/config/` are copied to the container folder `<TOMCAT_ROOT>/webapps/dctm-rest/WEB-INF/classes/` when the container is started. This means that users have full control over the REST configuration.

SSL Configuration

The image supports SSL HTTPS communication. There are five additional environment variables that are used in SSL communication in addition to those used in the basic scenario. You can use a self-signed certificate or a certificate from CA. You must import your certificate to a keystore so you can use it for HTTPS communication. After the certificate has been imported, you can start the container with the following command:

```

docker run --name rest -p 8080:8080 -p 8443:8443 -d \
    -v `pwd`/config:/root/rest/config \
    -e COMM_KEYSTORE_FILE=/root/rest/config/security/tc.jks \
    -e COMM_KEYSTORE_PWD=<KEYSTORE_PWD> \
    -e COMM_KEY_ALIAS=<KEY_ALIAS> \
    -e COMM_KEY_PWD=<KEY_PWD> \
    -e COMM_KEY_STORE_TYPE=<STORE_TYPE> \
    dctm-rest

```

The following table displays the attributes that are configurable in the `server.xml` file of the Tomcat 8 server.

Parameter	Description	Default Value
-e COMM_KEYSTORE_FILE =<KEYSTORE_FILE>	The path of the keystore file in the container, which is located in /root/rest/config	N/A
-e COMM_KEYSTORE_PWD =<KEYSTORE_PWD>	The password of the keystore	N/A
-e COMM_KEY_ALIAS=<KEY_ALIAS>	The alias of the entry to secure the communication	When there is only one entry in the keystore and the entry password is same as keystore, it's not necessary to define it.
-e COMM_KEY_PWD=<KEY_PWD>	The password of the entry	Same as COMM_KEY_ALIAS
-e COMM_KEY_STORE_TYPE =<STORE_TYPE>	The type of the keystore	JKS

Verify the REST SSL Communication

```
curl -k https://dmin:password@localhost:8443/dctm-rest/repositories/REPO
```

```
{
  "id": 5,
  "name": "REPO",
  "description": "",
  "servers": [
    {
      "name": "REPO",
      "host": "RESTCS72GA",
      "version": "7.2.0000.0155 Win64.SQLServer",
      "docbroker": "RESTCS72GA"
    }
  ],
  "links": [
    {
      "rel": "self",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/cabinets",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/cabinets"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/checked-out-objects",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/checked-out-objects"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/current-user",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/currentuser"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/current-user-preferences",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/currentuser-preferences"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/users",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/users"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/groups",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/groups"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/formats",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/formats"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/network-locations",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/network-locations"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/relations",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/relations"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/relation-types",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/relation-types"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/types",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/types"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/aspect-types",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/aspect-types"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/dql",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/{?dql}"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/search",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/search{?collections, facet, include-total, inline, items-per-page, locations, object-type, page, q, sort, timezone, view}"
    },
    {
      "rel": "http://identifiers.emc.com/linkrel/saved-searches",
      "href": "https://localhost:8443/dctm-rest/repositories/REPO/saved-searches"
    }
  ]
}
```

```
{
  "rel": "http://identifiers.emc.com/linkrel/search-templates",
  "href": "https://localhost:8443/dctm-rest/repositories/REPO/search-templates"},
  {"rel": "http://identifiers.emc.com/linkrel/acls",
  "href": "https://localhost:8443/dctm-rest/repositories/REPO/acls"},
  {"rel": "http://identifiers.emc.com/linkrel/batches",
  "href": "https://localhost:8443/dctm-rest/repositories/REPO/batches"},
  {"rel": "http://identifiers.emc.com/linkrel/batch-capabilities",
  "href": "https://localhost:8443/dctm-rest/repositories/REPO/batch-capabilities"]}
}
```

Logging

You can check the console logs by using this docker command:

```
docker logs rest
```

You can save your REST log files in the host in two ways:

- By default, the log files are saved in the container folder located in `/root/rest/logs`.

Run the `docker inspect rest` docker command to see which host folder is mounted and mapped to the container folder. Here's an example of the path for the host folder: `/var/lib/docker/volumes/2bf0e2bfc08bde54d187bc74b43fd4466ff94c3c3de81152250ecb25ce2bdf56/_data`. The log file `rest-api.log` is in this host folder.

Example 6-2. Saving REST Log File in the Host Folder

```
"Mounts": [
  {
    "Source": "/home/<your-username>/rest/all-in-one/config",
    "Destination": "/root/rest/config",
    "Mode": "",
    "RW": true,
    "Propagation": "rprivate"
  },
  {
    "Name": "2bf0e2bfc08bde54d187bc74b43fd4466ff94c3c3de81152250ecb25ce2bdf56",
    "Source": "/var/lib/docker/volumes/2bf0e2bfc08bde54d187bc74b43fd4466ff94c3c3de81152250ecb25ce2bdf56/_data",
    "Destination": "/root/rest/logs",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
],
```

- The second option is to run the REST container with the volume parameter. Using the `-v 'pwd' /logs/root/rest/logs` host folder `'pwd' /logs` maps to the container folder `/root/rest/logs`.

Example 6-3. Running the REST Container with the Volume Parameter

When the container is started by the command shown below, the logging file is in the `~/rest/logs` host folder.

```
docker run --name rest -p 8080:8080 -d \
  -v `pwd`/config:/root/rest/config \
  -v `pwd`/logs:/root/rest/logs \
```

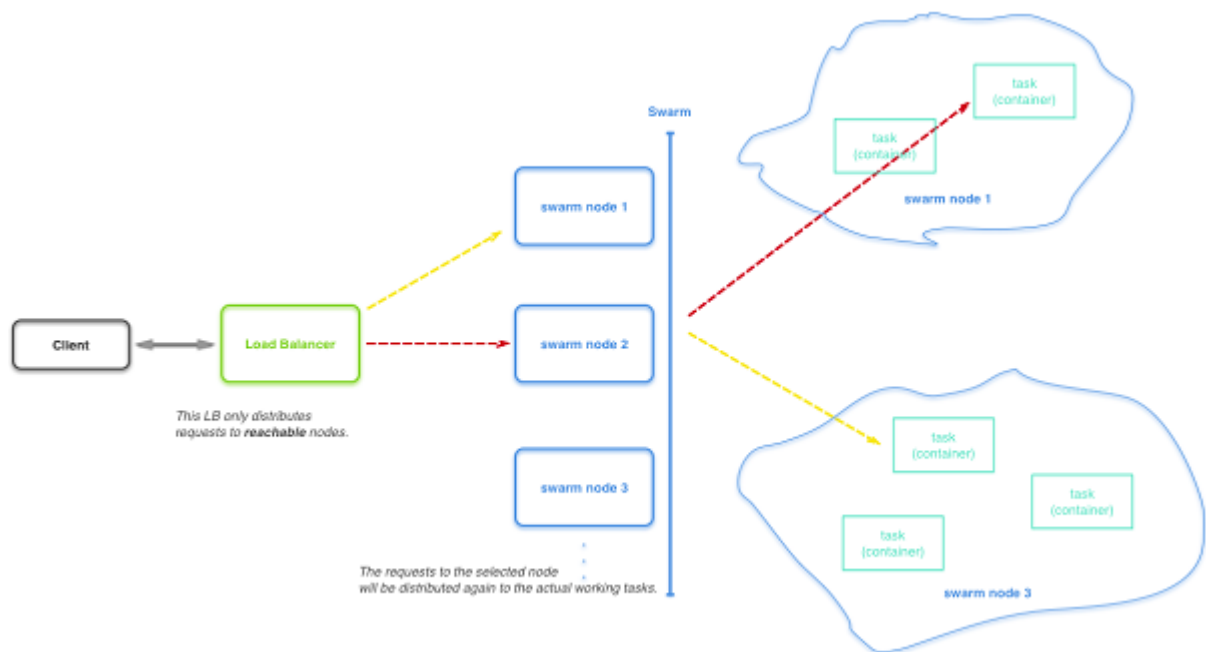
dctm-rest

Scalability and High Availability

The Docker Engine 1.12 includes **swarm mode** that can be used to natively manage a cluster of Docker Engines called a **swarm**. You can use the Docker Command Line Interface to create a swarm, deploy application services to a swarm, and manage swarm behavior. This feature allows you to use a purely Docker solution for load balancing, scalability, and seamless upgrading.

The figure below describe the topology of the Documentum Platform REST Services in swarm mode. The Load Balancer, shown below in green, is a native Docker functionality that routes Requests from the Client to various swarm nodes. Each swarm node has many running tasks, and each task can serve as specific job, such as Documentum Platform REST Services.

Once a REST Request arrives at the Load Balancer, it can be routed to any node on which Documentum Platform REST Services are running. When there are multiple tasks serving Documentum Platform REST Services on the selected node, one of the tasks is automatically selected to handle the incoming Request.



Follow these steps to deploy Documentum Platform REST Services in swarm mode:

- **Setup a swarm.** You must create a swarm and add nodes to it. For more information, see [Create a swarm](#) and [Add nodes to the swarm](#).
- **Deploy Documentum Platform REST Services.** You must use configuration files to run your REST services task. However, in a clustered environment the local configuration files that are in one swarm node are not available to other swarm nodes. One technique that allows you to use your configuration files across swarm nodes is to include the configuration files in the REST image.

Follow these steps to include your configuration files in the REST image:

1. **Create the REST node image.** Prepare the Dockerfile used to create the REST node image.

The Dockerfile sample below copies the configuration files (such as `dfc.properties` or `log4j.properties`) into the `config` building context folder that is in the REST node image.

```
FROM <REST_BASE_IMAGE>
#config directory containing the necessary configuration files
COPY config/ ${CONFIG_DIR}/
```

2. **Build the REST node image.** Use the following command to build the REST node image:

```
docker build -t <REST_NODE_IMAGE>
```

Note: After the REST node image is ready, we recommend that you push it to a registry so that other swarm nodes can retrieve and use it.

3. **Start the Documentum Platform REST Services as a Docker swarm service.** The `--replicas` argument specifies two instances of the Documentum Platform REST Services that will be set to run in the swarm.

The two instances may be on the same or different swarm nodes:

```
docker service create --replicas 2 --name rest-multinodes -p 8080:8080
<REST_NODE_IMAGE>
```

- **Scale Documentum Platform REST Services.** An existing Docker service can be scaled using the command shown in the following sample. In this case, Docker launches another 3 instances in the swarm.

```
docker service scale rest-multinodes=5
```

At this moment, a cluster of Documentum Platform REST Services is built according to the Docker swarm mode.

In such a deployment, there are two kinds of high availability that are supported:

1. One container running the Documentum Platform REST Services task is broken down.
The Swarm initializes another task to substitute the broken one. In our sample, there are still 5 tasks for the Documentum Platform REST Services.
2. One swarm node is broken down.
The Swarm migrates the tasks of the broken down node to other living nodes.

No matter which containers or swarm nodes are broken down, the swarm engine continues to maintain the server availability. It does by launching substitution tasks for broken containers or by migrating tasks from each broken down node to another living node.

Upgrading a Dockerized REST Service

REST services are stateless, and all the configuration files are mapped from the host to the container.

Upgrade a Dockerized REST Service

Let's assume that you want to upgrade your REST service from version 7.3 to version 7.3 P01. Follow these steps:

1. Pull the 7.3 P01 image from registry
2. Stop the 7.3 REST container
3. Start 7.3 P01 container with all of the existing configuration files of the 7.3 container

This upgrading process seems similar to the normal upgrading process using a `WAR` file. However, all existing troubleshooting steps can also be applied to the dockerized REST service, which will run exactly the same in different environments regardless of machine infrastructure.

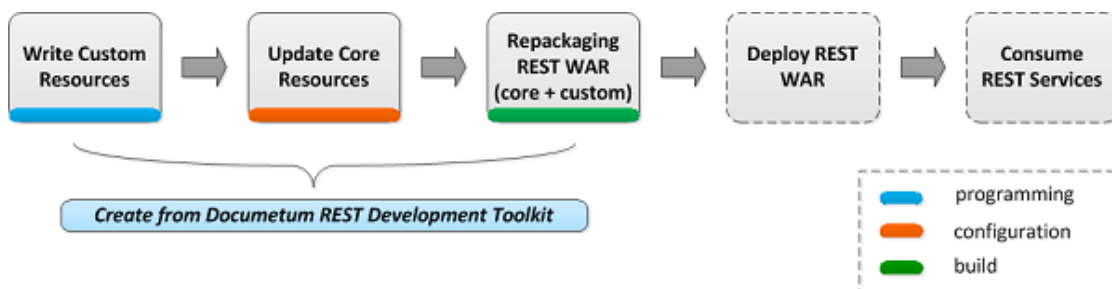
Resource Extensibility

Overview

Resource extensibility is an API infrastructure that enables you to extend Documentum Platform 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 by using HATEOS relations. By wielding the power of resource extensibility, you can tailor Documentum Platform REST Services to your needs with limited amount of coding.

The following diagram illustrates the lifecycle of custom resource development.

Figure 2. Lifecycle of Building Custom Resources



1. Design and implement custom resources.

Documentum Platform REST Services extensibility features all you to add custom resources to your REST applications. Your custom resources can be used with the out-of-box Core resources to help you develop a customized web application. The Documentum Platform REST Services SDK has a set of toolkits and libraries that you can use to development custom resources.

Here are the toolkits and libraries that are available in the Documentum Platform REST Services SDK:

- A set of Core REST Java libraries for custom resource development that enhances the REST services development of the Core REST services. Java docs and code samples for the shared library are also provided.
- A [Marshalling Framework](#) to handle XML or JSON message marshalling and unmarshalling of Core REST resources.
- A [Maven-based toolkit](#) to manage the build process of custom resource projects. An archetype is included and you can use it 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 custom resource development are available.

2. Customize Core REST resources.

Typically, the newly developed resources need to interact with Core resources via link relations or other representations. Documentum Platform 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 must give a link relation to the new resource.
3. Repackage the REST WAR file.
Documentum Platform 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 Platform REST Services, as long as the custom REST services does not bring any library conflict to the application servers.
 5. Consume the extended Documentum Platform 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.

Java API Deprecations

The following items list JAVA API deprecations:

- The `com.emc.documentum.rest.wire.xml.AnnotatedXmlMessageWriter.setSuggestedNamespace()` method has been deprecated. Manually suggesting namespaces does not produce a well-formatted XML representation. Moving forward, the `AnnotatedXmlMessageWriter` will automatically examine the XML root type to determine the default namespace
- The `com.emc.documentum.rest.binding.SerializableField$xmlListItemName` has been deprecated. It has been replaced with a new annotation called `com.emc.documentum.rest.binding.SerializableField4XmlList` that defines a Java List or Array field for custom marshaling and unmarshalling
- The `com.emc.documentum.rest.binding.SerializableField$xmlListUnwrap` has been deprecated. It has been replaced by a new annotation called `com.emc.documentum.rest.binding.SerializableField4XmlList` that defines a Java List or Array field for custom marshaling and unmarshalling
- The `com.emc.documentum.rest.binding.SerializableEntry` has been deprecated. This annotation was used to marshal to and unmarshal from a key-value pair. Since `java.util.Map` is supported as a single serializable field `com.emc.documentum.rest.binding.SerializableEntry` is no longer needed
- The `com.emc.documentum.http.UriFactory` has been deprecated. It has been replaced by `com.emc.documentum.rest.context.ResourceUriBuilder`, which provides a typical pattern that can be used to build a URI
- The `com.emc.documentum.rest.http.SupportedMediaTypes` has been deprecated
- The `com.emc.documentum.rest.view.FeedableView.getUriFactory()` has been deprecated
It has been replaced by `com.emc.documentum.rest.context.ResourceUriBuilder(String)`
- The `com.emc.documentum.rest.view.LinkableView` has been deprecated. It has been replaced by `com.emc.documentum.rest.context.ResourceUriBuilder`
- `LinkableView.getUriFactory()` has been deprecated and the method has been moved to a new parent class called `LegacyLinkableView`
- The `com.emc.documentum.rest.dfc.ContentManager.setPrimaryContent(String, InputStream, String, String, int, boolean, CheckinPolicy, String)` has been deprecated. New parameters have been added, please use `setPrimaryContent(String, InputStream, long, String, String, String, int, boolean, CheckinPolicy, String)` instead

- The `com.emc.documentum.rest.dfc.ContentManager.setRendition(String, InputStream, String, String, int, String, boolean)` has been deprecated. New parameters have been added, please use `setRendition(String, InputStream, String, String, String, int, String, boolean)` instead
- The `com.emc.documentum.rest.dfc.ContextSessionManager.executeWithinTheContextTran(Callable <T>)` has been deprecated. It has been replaced by `executeWithinTheContextTran(SessionCallable)`

Get Started With the Development Kit

Documentum Platform 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 Platform 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 JAR files

- A Maven archetype project for custom resource development

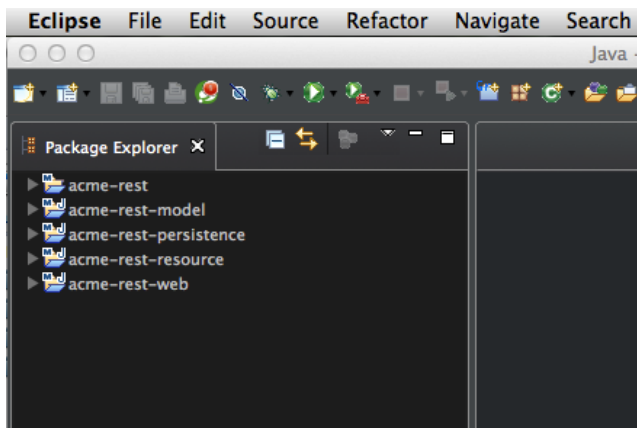
The archetype project can be used as a template project for custom resource development. This project can be installed in both local and remote repositories

- A script to install the Maven archetype and dependencies into a local Maven repository
- A script to create a sample project from the Maven archetype

Creating a Custom Resource Project from the Maven Archetype

Instead of requiring you to create a custom resource project from scratch, Documentum Platform 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 3. Maven Archetype Project Structure**System Requirements:**

- Java 7 or Java 8 must be installed, and the path location of the Java installation must be added to the `classpath` Environment 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 and dependencies to your local repository](#)
2. [Creating a project based on the archetype](#)

Installing the Maven Archetype and Dependencies

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`
3. View the following Readme files: `archetype-install-guide.md` and `dependencies-install-guide.md`. These two ReadMe files guide you through the installation process of the local Maven archetype and dependencies.
4. The SDK has a script that you can run to the Maven artifacts. To run this script, follow these instructions:
 - a. Open a command line interface

Run the command that applies to your operating system:

- **Windows**

```
dctm-maven-offline-install.bat
```

- **Linux or Mac**

```
bash dctm-maven-offline-install.sh
```

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`

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

The Eclipse IDE has support for Maven archetype projects. In this section, we use Eclipse 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 click **Finish**. Eclipse starts to build a project from the Maven archetype.

Creating a Project from the Maven Archetype in a Common-Line Prompt

If 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. Navigate to the `documentum-rest-sdk-version-number/maven-kit` directory
2. Run the script command below that applies to your operating system:

- **Windows**

```
dctm-rest-getstarted.bat
```

- **Linux or Mac**

```
bash dctm-rest-getstarted.sh
```

After the task completes, a new project is created in the directory that is generated.

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 and Platform Extensions 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 and Platform Extensions 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>
    </author>
```

```
    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-information/classes` folder and then edit the `dfc.properties` file according to your Content Server configuration.

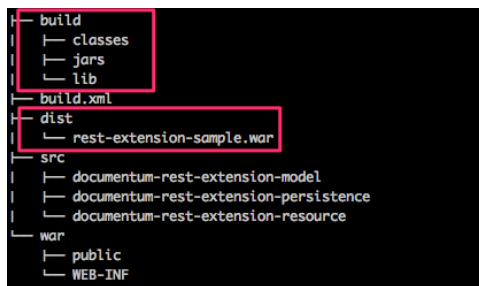
For more information about how to configure `dfc.properties`, see the DFC Configuration section of the *EMC Documentum Platform and Platform Extensions Release Notes*.

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 4. 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
```

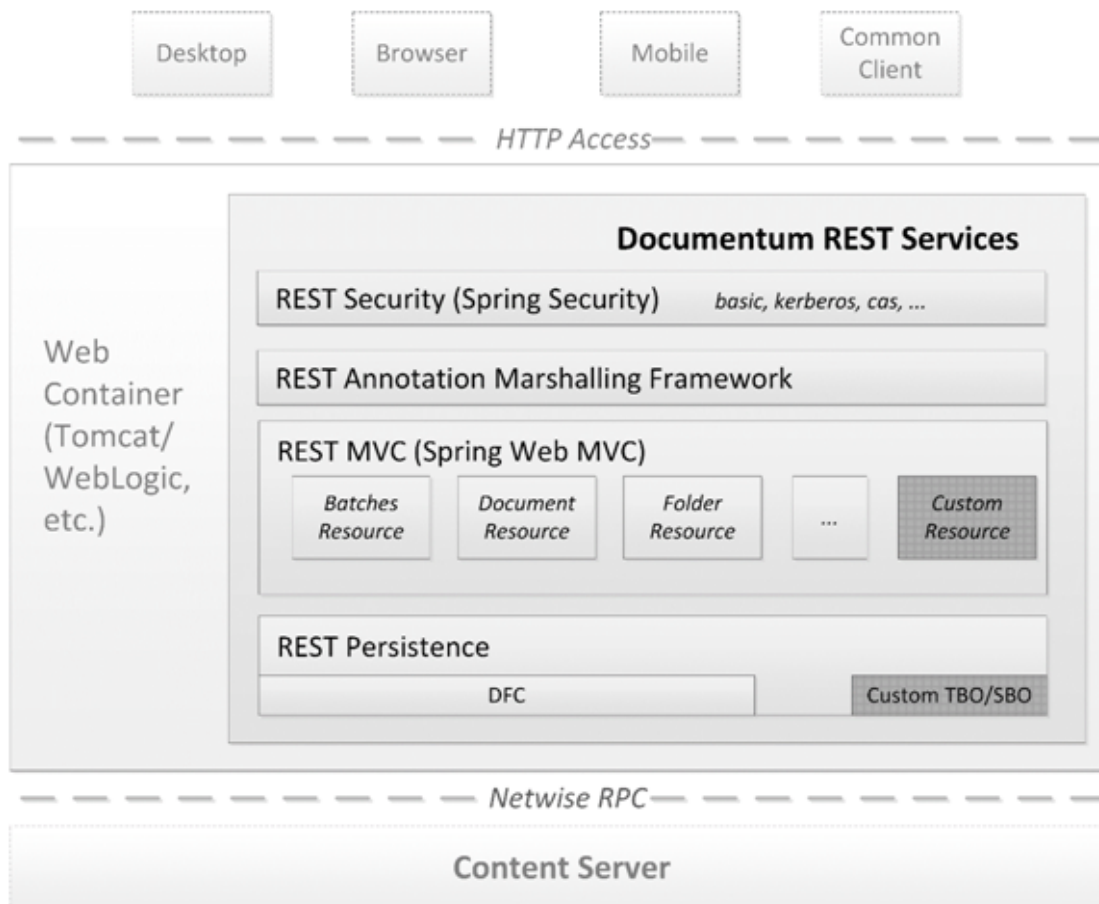
If both the build creation and WAR file deployment are successful, the operation returns a collection of alias sets in the repository.

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 Platform 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 Platform REST Services. The following diagram illustrates the overall architecture of Documentum Platform REST Services.

Figure 5. Documentum Platform REST Services Architecture



The whole Documentum Platform REST Services is built as a single WAR application, including both Core resources and custom resources.

Documentum Platform 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 Platform 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 Platform 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 Platform 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 Platform REST Services at the framework level.

Under the bottom of the Documentum REST MVC, persistence APIs communicate with Content Server repositories. Documentum Platform REST Services provides a lot of common APIs to manipulate persistent data in the Content Server repositories. Besides, Documentum Platform 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

Custom resources often use the same authentication scheme as Core resources within the same WAR deployment. More precisely, Documentum Platform 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 included in the specified authentication scheme.

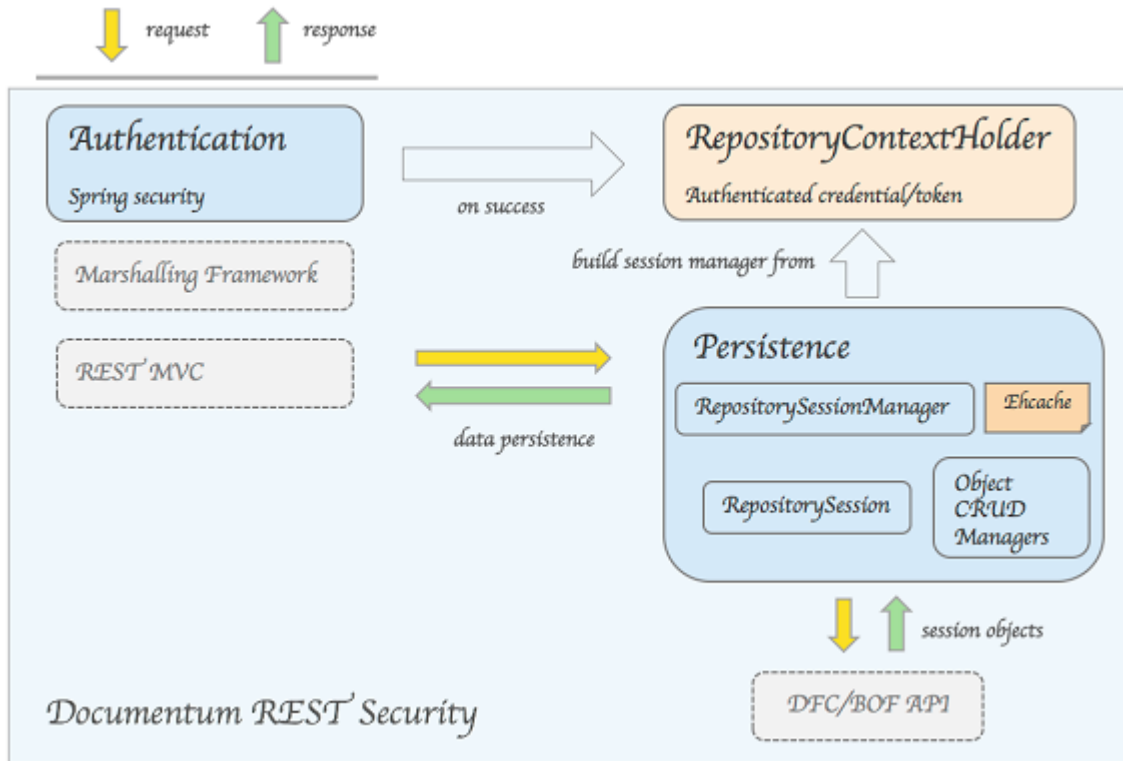
Documentum Platform REST Services applies many authentication schemes by using the resource URI pattern `/repositories/{repositoryName}` and all of its subsequent path segments. Therefore, when a custom resource's URI is created with the prefix `/repositories/{repositoryName}/`, such as `/repositories/{repositoryName}/alias-sets`, access to that resource is authenticated by the configured authentication scheme(s). When this is not the case, such as in `/help`, access to the resource is anonymous.

For more information, see [Custom Authentication Development](#).

The security filters actually perform the authentication for access to Content Server repositories or third party Single-sign-On entities. When the authentication procedure is successful, the Persistence layer of the resource implementation retrieves the authenticated Documentum Foundation Classes (DFC) session manager. It retrieves the 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 the `com.emc.documentum.rest.dfc.SessionAwareAbstractManager` API to get the repository session.

For more information, see [Persistence: Session Management](#).

The following diagram illustrates the relationship between REST authentication and the session related APIs:



Here is an explanation on what is happening in the diagram:

- When authentication succeeds, it stores the authenticated user credential or Single Sign-On token in the `com.emc.documentum.rest.config.RepositoryContextHolder` Core REST class.
- Resource controllers that are used in REST MVC call persistence object managers to perform any persistent object operations.
- Persistent object managers are auto-wired with a `com.emc.documentum.rest.dfc.RepositorySession` Java bean.
- The `RepositorySession` Java bean is auto-wired using a Singleton of the `com.emc.documentum.rest.dfc.RepositorySessionManager` Java bean.
- The Singleton of the `RepositorySessionManager` Java bean retrieves the DFC session manager according to the user principle from the `RepositoryContextHolder` Java bean for the current Request.
- Ehcache is used to store the DFC session manager, and the cache key is generated from the `RepositoryContextHolder` Java bean.
- The `RepositoryContextHolder` bean is cleared after each Request exits, however the Ehcache for the DFC session manager remains for a period of time.

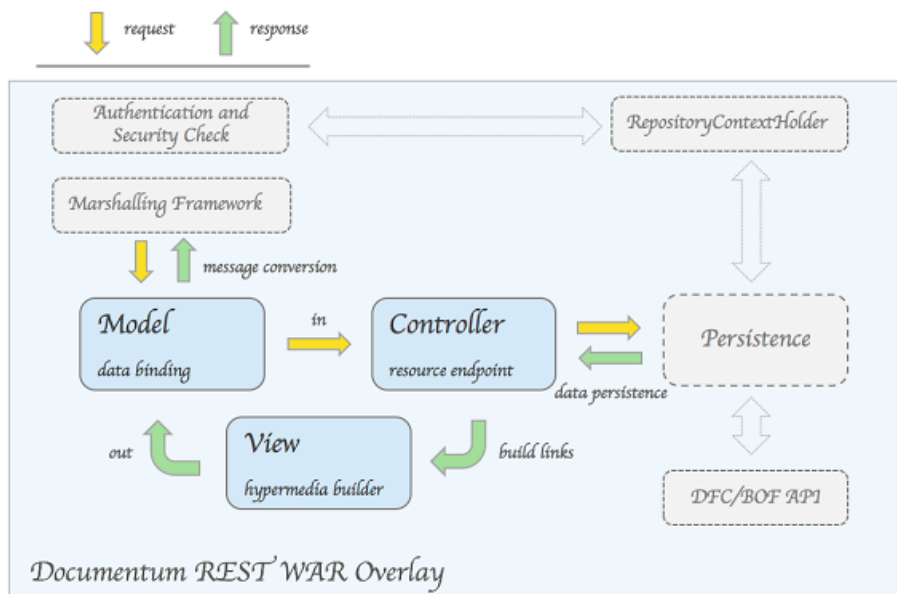
Documentum REST MVC

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 6. 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 Persistence

Documentum Platform 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 `com.emc.documentum.rest.dfc` Java package and its sub packages.

Custom resources may also need to write new persistence APIs or integrate their existing type-based objects (TBOs) or service-based objects (SBOs) into the REST implementation. The Custom persistence APIs must be written using the same pattern as other Core persistence APIs, and they must 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) {
        ...
    }
}
```

Note: For more information on the usage of DFC sessions in the object manager, see .

3. Implement it as a Java bean.

There are two ways to create a Java bean:

- Spring Java code configuration
- Using an XML namespace

Example 7-1. Create a Java Bean Using Spring Code Configuration

```
@Configuration
@ComponentScan(
    basePackages = { "com.acme" },
    excludeFilters = {
        @ComponentScan.Filter(
            type = FilterType.CUSTOM,
            classes = {
                com.emc.documentum.rest.context.ComponentScanExcludeFilter.class
            })
    }
)

public class CustomContextConfig {
    @Bean(name="customUserManager")
    public UserManager customUserManager() {
        return new UserManagerImpl();
    }
}
```

You must ensure that the package where you created your custom configuration class is specified in the `rest.context.config.location` property within the `rest-api-runtime.properties` file. For example, when the `CustomContextConfig` class is under the `com.acme.context.config` package, the `rest.context.config.location` property should be defined in runtime properties file as shown here:

```
rest.context.config.location=com.acme.context.config
```

When you use the Spring `@ComponentScan` annotation in your custom defined configuration class, the `com.emc.documentum.rest.context.ComponentScanExcludeFilter` exclude filter is mandatory. This exclude filter ensures that the Spring framework loads all of the resources that you have defined.

Example 7-2. Create a Java Bean Using Spring XML Namespace

Define the 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;
    ...
}
```

For more information, see [persistence programming](#) for the details of implementing a persistence API.

Documentum REST Marshalling Framework

Overview

Documentum Platform REST Services provides a simple yet powerful marshalling framework to facilitate object to object communication. The Documentum Platform REST Services marshalling framework introduces a set of Java annotations that bind models directly to your XML or JSON representations. This data binding eliminates the need to write message converters for custom data models.

By default, core and custom resources use the annotation marshalling framework to convert XML and JSON messages. However, because Documentum Platform REST Services also supports Spring message conversions, these messages can be altered by users who understand how to work with [Spring message conversions](#).

Features of the REST Marshalling Framework

The REST Marshalling Framework has the following features:

- Out-of-the-box message marshalling

The REST Marshalling Framework handles the serialization of Java objects into HTTP messages, and the deserialization of HTTP messages into Java object 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, when a Java class is annotated, both XML and JSON representation are supported.

The resource controller returns the annotated class instance directly, leaving all the message conversion work to [Documentum REST MVC](#) and the REST 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.
- `@SerializableField4XmlList`
Used on a Java class field to serialize the Java List type field to a sub element/property of a REST representation.
- `@SerializableField4XmlMap`
Used on a Java class field to serialize the Java Map type field to a sub element/property of a REST representation.

All annotations work for 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 and unmarshalling behavior.

Table 7. SerializableType Attributes

Attributes	Description	Supported Formats	Marshall / Unmarshall	Default
value	Specifies the serializable name	JSON/XML	in and out	not set
fieldVisibility	Specifies whether to serialize all or some fields according to the access modifiers at the class level: <ul style="list-style-type: none"> • ALL - all fields are intended to be serialized but static fields are excluded. Static fields are serializable only when they are explicitly annotated with the 	JSON/XML	in and out	ALL

Attributes	Description	Supported Formats	Marshall / Unmarshall	Default
	<p><code>@SerializableField</code> annotation</p> <ul style="list-style-type: none"> PUBLIC - public fields are intended to be serialized but public static fields are excluded. Public static fields are serializable only when they are explicitly annotated with the <code>@SerializableField</code> annotation NONE - no fields are intended to be serialized <p>Settings of the annotation <code>@SerializableField</code> take precedence over this attribute. For example, when <code>fieldVisibility</code> is set to <code>none</code>, properties with the annotation <code>@SerializableField</code> are still intended to be serialized.</p> <p>For more information, see Example A and Example B.</p>			
fieldOrder	<p>Specifies the order of fields in the REST representation for a serialized object.</p> <p>For fields that can be serialized but not appearing in this list, they are presented at the end of the REST representation.</p> <p>For fields appearing in the list, but not serialized, they are ignored.</p> <p>For more information, see Example C.</p>	JSON /XML	out	not set

Attributes	Description	Supported Formats	Marshall / Unmarshall	Default
ignoreNullFields	Specifies whether to ignore fields with null values when marshalling. For more information, see Example D.	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 <code>inlineField</code> to a certain field, all other fields in the class are not serialized. For more information, see Example E.	JSON /XML	out	not set
xmlValueField	Specifies a field from which the value will be taken as the XML element value of this type. For more information, see Example F.	XML	in and out	not set
jsonWriteRootAsField	For JSON marshalling, specifies whether to write the root name (specified by the <code>value</code> attribute) into a field specified by the annotation <code>jsonRootField</code> . For more information, see Example G. If <code>jsonWriteRootAsField</code> is <code>false</code> 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

Attributes	Description	Supported Formats	Marshall / Unmarshall	Default
xmlNS	Specifies the namespace for XML representation. For more information, see Example H .	XML	out	not set inherited from the parent class
xmlNSPrefix	Specifies the namespace prefix for XML representation. For more information, see Example H .	XML	out	not set inherited from the parent class
inheritValue	Specifies whether to reuse the serializable name of the super type. <ul style="list-style-type: none"> • 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
unknownFieldDeserializers	You can specify any <code>AnnotatedFieldDeserializer</code> implementation to process an unknown field. However, the <code>unknownFieldDeserializers</code> annotation field is only used when deserializing.	JSON /XML	out	Annotated Exceptional UnknownField Deserializer

Examples

This section lists several examples explaining the detailed usage of the attributes that are available in the `@SerializableType` annotation.

Example 7-3. Public Field Visibility

In the following example, the non-public field `id` is not serialized.

Java Definition	XML	JSON
<pre> @SerializableViewType (value = "business-object", fieldVisibility SerializableType.FieldVisibility.PUBLIC) public class BusinessObject { private String id; public Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } // field visibility for public fields public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6); } } </pre>	<pre> <?xml version='1.0' encoding='UTF- 8'?> <business-object> <vstamp>6</vstamp> </business-object> </pre>	<pre> { "vstamp":6 } </pre>

Example 7-4. None Field Visibility

In the following example, only the field `active` annotated by `@SerializableField` is serialized.

Java Definition	XML	JSON
<pre> @SerializableViewType (value = "business-object", fieldVisibility = SerializableType.FieldVisibility.NONE) public class BusinessObject { private String id; private Integer vstamp; @SerializableField private Boolean active; public BusinessObject(String id, Integer vstamp, Boolean active) { this.id = id; this.vstamp = vstamp; this.active = active; } // field visibility for none fields public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6, true); } } </pre>	<pre> <?xml version='1.0' encoding='UTF- 8'?> <business-object> <active>true</active> </business-object> </pre>	<pre> { "active":true } </pre>

Example 7-5. Field Order

In the following example, fields are serialized according to the order specified in `fieldOrder`.

Java Definition	XML	JSON
<pre> @SerializableViewType (value = "business-object", fieldOrder = {"vstamp", "id"}) public class BusinessObject { private String id; private Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } // field order public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6); } } </pre>	<pre> <?xml version='1.0' encoding='UTF-8'?> <business-object> <vstamp>6</vstamp> <id>09ae2d</id> </business-object> </pre>	<pre> { "vstamp":6, "id":"09ae2d" } </pre>

Example 7-6. Ignore Null Fields

In the following example, the null field `id` is not serialized.

Java Definition	XML	JSON
<pre> @SerializableType (value = "business-object", ignoreNullFields = true) public class BusinessObject { private String id; private Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } // ignore null fields public static void main(String[] args) { BusinessObject bo = new BusinessObject(null, 0); } } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <business-object> <vstamp>6</vstamp> </business-object> </pre>	<pre> { "vstamp":6 } </pre>

Example 7-7. 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.

Java Definition	XML	JSON
<pre> @SerializableType (value = "business-object", inlineField = "event") public class BusinessObject { private String id; private Integer vstamp; private Event event; public BusinessObject(String id, Integer vstamp, Event event) { this.id = id; this.vstamp = vstamp; this.event = event; } } @SerializableType public static class Event { private String eventId; private Boolean active; public Event (String eventId, Boolean active) { this.eventId = eventId; this.active = active; } } // inline public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6, new Event("775200", true)); } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <event> <eventId>775200</eventId> <active>true</active> </event> </pre>	<pre> { "eventId":"775200", "active":true } </pre>

Example 7-8. 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.

Java Definition	XML	JSON
<pre> @SerializableType (value = "business-object", xmlValueField = "vstamp") public class BusinessObject { @SerializableField(xmlAsAttribute = true) private String id; private Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } } // xml value field public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6); } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <business-object id="09ae2d">6</business-object> </pre>	<pre> { "id":"09ae2d", "vstamp":6 } </pre>

Example 7-9. 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`.

Java Definition	XML	JSON
<pre> @SerializableType (value = "business-object", jsonWriteRootAsField = true) public class BusinessObject { private String id; private Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } // json write root public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6); } } </pre>	<pre> <?xml version='1.0' encoding='UTF-8'?> <business-object> <id>09ae2d</id> <vstamp>6</vstamp> </business-object> </pre>	<pre> { "name": "business-object", "id": "09ae2d", "vstamp": 6 } </pre>

Example 7-10. Namespace

In the following example, XML namespaces and prefixes are added.

Java Definition	XML	JSON
<pre> @SerializableType (value = "business-object", xmlNSPrefix = "dm", xmlNS = "http://documentum.emc.com") public class BusinessObject { private String id; private Integer vstamp; private Event event; public BusinessObject(String id, Integer vstamp, Event event) { this.id = id; this.vstamp = vstamp; this.event = event; } } @SerializableType (value = "event", xmlNSPrefix = "acme", xmlNS = "http://acme.org") public static class Event { private Boolean active; public Event (Boolean active) { this.active = active; } } // xml namespace public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6, new Event((true))); } </pre>	<pre> <?xml version='1.0' encoding='UTF-8'?> <business-object xmlns="http://documentum.emc.com" xmlns:acme="http://acme.org"> <id>09ae2d</id> <vstamp>6</vstamp> <acme:Event> <acme:active>true</acme:active> </acme:Event> </business-object> </pre>	<pre> { "id": "09ae2d", "vstamp": 6, "event": { "active": true } } </pre>

Unmarshalling with Undefined Attributes

A new property called *unknownFieldDeserializers* has been added to the annotation *SerializableType*. You can specify any *AnnotatedFieldDeserializer* implementation to process an unknown field. However, the *unknownFieldDeserializers* annotation field is only used when deserializing. The default value of the *unknownFieldDeserializers* property is *AnnotatedExceptionalUnknownFieldDeserializer* implementation, which throws *AnnotationParseException* exception when finding any unknown element.

- For backward compatibility Unknown elements parsing (deserializing) will have the *AnnotationParseException* exception added in Documentum Platform REST Services version 7.3, which may be correctly parsed by 7.2.

The solutions for the above issue are as follows:

- Avoid sending elements that do not exist
- Customize the *unknownFieldDeserializers*
- When you use *unknownFieldDeserializers*, you will want to ignore all unknown elements

In some cases, unknown elements are useless, or don't need to be processed. Although in these cases, it is recommended to throw the exception to let the client know that the server can also ignore these unknown elements.

In addition to throwing an exception with *AnnotatedExceptionalUnknownFieldDeserializer*, there is another default implementation, *AnnotatedIgnoreableUnknownFieldDeserializer*, that is also provided. This implementation ignores the unknown elements directly. To use it, just set it to *unknownFieldDeserializers* of the *SerializableType*

Here's an example that shows you how to do this:

```
@SerializableType(value="model",
    unknownFieldDeserializers={AnnotatedIgnoreableUnknownFieldDeserializer.class})
public class SomeModel2 {
    private String fieldA;
    private String fieldB;
}
```

The unknown elements are ignored during deserialization process. Ignoring the unknown elements allows the existing elements to be correctly populated. Here's an example that illustrates this point:

```
<model>
  <fieldA>a</fieldA>
  <fieldB>b</fieldB>
  <unknownField>x</unknownField>
</model>

<model unknownAttribute="x">
  <fieldA>a</fieldA>
  <fieldB>b</fieldB>
</model>

{
  "fieldA": "a",
  "fieldB": "b",
  "unknownField": "x"
}
```

- Specify empty *unknownFieldDeserializers*

An empty *unknownFieldDeserializers* explicitly sets *unknownFieldDeserializers* to {} for a class. Here's an example that illustrates this technique:

```
@SerializableType(value="model", unknownFieldDeserializers={})
public class SomeModel3 {
    private String fieldA;
    private String fieldB;
}
```

Setting the empty *unknownFieldDeserializers* property causes the binding framework to decide whether to process the unknown field or ignore it. This is useful because some 'unknown' elements may be meaningful and may need to be processed by the system.

The typically used for inheritance. You declare the field as a parent class, but the field instance is actually a child of the parent. The server tries to deserialize the representation with the parent while the attributes belonging to the child are unknown to the parent. The following example helps to understand this use case:

```
@SerializableType(value="parent", unknownFieldDeserializers= {})
public abstract class AbstractParent {
    private String name;
}
```

```
@SerializableType(value="child-a", jsonWriteRootAsField=true, jsonRootField="parent")
public class ChildA extends AbstractParent {
    @SerializableField(xmlAsAttribute=true)
    private int age;
}

@SerializableType(value="child-b", jsonWriteRootAsField=true, jsonRootField="parent")
public class ChildB extends AbstractParent {
    private boolean graduated;
}

@SerializableType(value="group")
public class Group {
    List<AbstractParent> children;
}
```

There are three instances, one parent and two children. Each child has its own properties. The parent has the empty *unknownFieldDeserializers* property defined. The fourth instance has a List of *AbstractParent* data type. The output representation may be:

```
<?xml version='1.0' encoding='UTF-8'?>
<group>
  <children>
    <child-a age="10">
      <name>child a</name>
    </child-a>
    <child-b>
      <graduated>true</graduated>
      <name>child b</name>
    </child-b>
  </children>
</group>

{"children":[
  {"parent":"child-a","age":10,"name":"child a"},
  {"parent":"child-b","graduated":true,"name":"child b"}
]}
```

When deserializing the representation for the *Group* element, the system tries to use *AbstractParent* to parse the child representation. The child contains the 'unknown' elements 'age' and 'graduated' for *AbstractParent*. With empty *unknownFieldDeserializers*, the binding system processes it correctly.

The server throws an exception when you don't set *unknownFieldDeserializers* to empty and the default *AnnotatedExceptionalUnknownFieldDeserializer* is used.

When you set the *AnnotatedIgnoreableUnknownFieldDeserializer*, unknown elements are ignored. The server creates *AbstractParent* with its own elements. When *AbstractParent* is not abstract, the instance can be created. When it is abstract, which is the case here, the exception is thrown.

The difference between *AnnotatedIgnoreableUnknownFieldDeserializer* and empty *unknownFieldDeserializers* is that *AnnotatedIgnoreableUnknownFieldDeserializer* ignores the elements, whereas empty *unknownFieldDeserializers* lets the server decide how to deserialize it. The server may decide to deserialize it or ignored the unknown elements.

Note: When using inheritance, the *unknownFieldDeserializers* must be set to empty.

- Implement customized deserializer

The customized deserializer should implement *AnnotatedFieldDeserializer* directly. You can extend *AbstractFieldJsonDeserializer* or *AbstractFieldXmlDeserializer*.

```
@SerializableType(value="model", fieldVisibility=FieldVisibility.NONE,
    unknownFieldDeserializers= {MyFieldJsonDeserializer.class,
                                MyFieldXmlDeserializer.class})

public class SomeModel4 {
    @SerializableField
    private String name;
    private String myfoo;

    //get/set methods

    public static class MyFieldXmlDeserializer extends AbstractFieldXmlDeserializer {
        @Override
        public Object deserialize(Object parser, Object object, String name,
            SerializableFieldMeta fieldNode, Map<String, Object> infoMap) {
            if(!"foo".equals(name)) {
                throw new AnnotationParseException("E_UNDEFINED_FIELD_NAME", name);
            }
            if(parser instanceof StartElement) {
                return readAttribute((SomeModel4)object, (StartElement)parser, name);
            } else {
                return readElement((SomeModel4)object, asEventReader(parser), name);
            }
        }
        @Override
        public boolean deserializable(Object next, SerializableFieldMeta fieldNode) {
            return true;
        }
        private String readElement(SomeModel4 childD, XMLEventReader reader, String name) {
            try {
                XMLEvent event = null;
                while((event=reader.nextEvent())!=null) {
                    if(event.isCharacters()) {
                        childD.setMyfoo(event.asCharacters().getData());
                    } else if(event.isEndElement()) {
                        EndElement ee = event.asEndElement();
                        if(ee.getName().getLocalPart().equals(name)) {
                            break;
                        }
                    }
                }
            } catch (XMLStreamException e) {
                throw new AnnotationParseException(e, "E_PARSE_XML_FIELD_ERROR", name, this);
            }
            return null;
        }

        private String readAttribute(SomeModel4 childD, StartElement startElement,
            String name) {
            Iterator<?> i = startElement.getAttributes();
            while(i.hasNext()) {
                Attribute attr = (Attribute)i.next();
                if(name.equals(attr.getName().getLocalPart())) {
                    childD.setMyfoo(attr.getValue());
                    break;
                }
            }
            return null;
        }
    }

    public static class MyFieldJsonDeserializer extends AbstractFieldJsonDeserializer {
        @Override
```

```
public Object deserialize(Object parser, Object object, String name,
    SerializableFieldMeta fieldNode, Map<String, Object> infoMap) {
    if("foo".equals(name)) {
        JsonParser jParser = asJacksonParser(parser);
        try {
            JsonToken next = jParser.nextToken();
            if(next == JsonToken.VALUE_STRING) {
                ((SomeModel4) object).setMyfoo(jParser.getText());
            }
        } catch (IOException e) {
            throw new AnnotationParseException(e, "E_PARSE_JSON_FIELD_ERROR",
                name, this);
        }
        return null;
    } else {
        throw new AnnotationParseException("E_UNDEFINED_FIELD_NAME", name);
    }
}

@Override
public boolean deserializable(Object next, SerializableFieldMeta fieldNode) {
    return true;
}
}
```

The `SomeModel4` class defines two *unknownFieldDeserializers* classes; one for JSON and one for XML. Use one of the classes (for JSON or XML) to parse all raw representation according your requirements.

Note the following items in the example:

1. `MyFieldXmlDeserializer` processes the raw XML representation, and only the `foo` element is able to be processed
2. `StartElement` is used to parse the XML attributes
3. `XMLEventReader` is used for parsing the XML elements
4. `MyFieldJsonDeserializer` reads the raw JSON representation, and only the `foo` element is able to be processed
5. `JsonParser` is used to parse fields



@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 and unmarshalling behavior.

Note: Care must be taken before applying the `@SerializableField` annotation to final or static fields. Unexpected behavior may occur when manipulating these final or static fields.

Table 8. @SerializableField Attributes

Attribute	Description	Supported Formats	Marshal /Unmarshal	Default
value	<p>Specifies the node name of the serializable type on one field of the class, for example, the name of an element in XML.</p> <p>For more information, see Example I.</p>	XML/JSON	in and out	not set
required	<p>Specifies whether a field is required to be non-null for marshalling.</p> <p>When the field is required but its value is null, an exception is thrown during marshalling.</p>	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	<p>This setting applies to a field of complex type.</p> <p>A complex type refers to a custom type annotated with <code>SerializableType</code>.</p> <p>Specifies whether to write <code>SerializableType.value</code> instead of <code>SerializableField.value</code> as the direct XML child element for a complex type.</p> <ul style="list-style-type: none"> • true - write <code>SerializableType.value</code> as the direct XML child element for a complex type. • false - write <code>SerializableField.value</code> as the direct XML child element for a complex type. 	XML	in and out	false
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

Attribute	Description	Supported Formats	Marshal /Unmarshal	Default
xmlAsAttribute	Specifies whether the field is represented as an attribute in the XML element attribute or a child element. For more information, see Example J .	XML	in and out	false
 Caution: Deprecated xmlListUnwrap	Specifies whether to unwrap the items from a <code>java.lang.List</code> type field as the direct member of the serialized object. <ul style="list-style-type: none"> 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. For more information, see Example K .	XML	in and out	false
 Caution: Deprecated 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 <code>List<String></code> . If the field is a list of custom classes, such as <code>List<MyType></code> , the attribute does not work and the default value <code>item</code> is used as the element name for the list. For more information, see Example L .	XML	in and out	item

Attribute	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. For more information, see Example M .	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. For more information, see Example M .	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 7-11. Field Serializable Name

In the following example, the field `vstamp` is serialized as `version-stamp`.

Java Definition	XML	JSON
<pre> @SerializableType ("business-object") public class BusinessObject { private String id; @SerializableField("version-stamp") private Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } // customize field name public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6); } } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <business-object> <id>09ae2d</id> <version-stamp>6</version-stamp> </business-object> </pre>	<pre> { "id": "09ae2d", "version-stamp": 6 } </pre>

Example 7-12. Field as XML Attribute

In the following example, the field `vstamp` is serialized as an attribute.

Java Definition	XML	JSON
<pre> @SerializableType ("business-object") public class BusinessObject { private String id; @SerializableField(xmlAsAttribute = true) private Integer vstamp; public BusinessObject(String id, Integer vstamp) { this.id = id; this.vstamp = vstamp; } // field as xml attribute public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6); } } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <business-object vstamp="6"> <id>09ae2d</id> </business-object> </pre>	<pre> { "id": "09ae2d", "vstamp": 6 } </pre>

Example 7-13. Field as XML Unwrapped List

In the following example, the list of `outEvents` are unwrapped.

<pre> @SerializableType ("business-object") public class BusinessObject { private String id; private Integer vstamp; @SerializableField(value = "in-events", xmlListUnwrap = false) private List<Event> inEvents; @SerializableField(value = "out-events", xmlListUnwrap = true) private List<Event> outEvents; public BusinessObject(String id, Integer vstamp, Event[] in, Event[] out) { this.id = id; this.vstamp = vstamp; this.inEvents = in == null ? new ArrayList<Event>() : Arrays.asList(in); this.outEvents = out == null ? new ArrayList<Event>() : Arrays.asList(out); } @SerializableType("event") public static class Event { @SerializableField("eid") private Integer eid; public Event (Integer eid) { this.eid = eid; } } // field as xml unwrapped list public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6, new Event[]{new Event(1), new Event(2)}, new Event[]{new Event(3), new Event(4)}); } } </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <business-object> <id>09ae2d</id> <vstamp>6</vstamp> <in-events> <event> <eid>1</eid> </event> <event> <eid>2</eid> </event> </in-events> <event>3</event> <event>4</event> </business-object> </pre>	<pre> { "id": "09ae2d", "vstamp": 6, "in-events": [{ "eid": 1 }, { "eid": 2 }], "out-events": [{ "eid": 3 }, { "eid": 4 }] } </pre>
---	--	---

Example 7-14. 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`.

Java Definition	XML	JSON
<pre>@SerializableViewType ("business-object") public class BusinessObject { private String id; private Integer vstamp; @SerializableViewField(value = "in-events") private List<String> inEvents; @SerializableViewField(value = "out-events", xmlListItemName = "out-event") private List<String> outEvents; public BusinessObject(String id, Integer vstamp, String[] in, String[] out) { this.id = id; this.vstamp = vstamp; this.inEvents = in == null ? new ArrayList<String>() : Arrays.asList(in); this.outEvents = out == null ? new ArrayList<String>() : Arrays.asList(out); } // field xml list item name public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6, new String[]{"e-1", "e-2"}, new String[]{"e-3", "e-4"}); } }</pre>	<pre><?xml version='1.0' encoding='UTF-8'?> <business-object> <id>09ae2d</id> <vstamp>6</vstamp> <in-events> <item>e-1</item> <item>e-2</item> </in-events> <out-events> <out-event>e-3</out-event> <out-event>e-4</out-event> </out-events> </business-object></pre>	<pre>{ "id": "09ae2d", "vstamp": 6, "in-events": ["e-1", "e-2"], "out-events": ["e-3", "e-4"] }</pre>

Example 7-15. Field XML Namespace

In the following example, XML namespaces and prefixes are added.

Java Definition	XML	JSON
<pre>@SerializableViewType (value = "business-object", xmlnsPrefix = "dm", xmlns = "http://documentum.emc.com") public class BusinessObject { private String id; @SerializableViewField(xmlAsAttribute = true, xmlnsPrefix = "xcp", xmlns = "http://xcp.emc.com") private Integer vstamp; @SerializableViewField(xmlNSPrefix = "d2", xmlns = "http://d2.emc.com") private String config; public BusinessObject(String id, Integer vstamp, String config) { this.id = id; this.vstamp = vstamp; this.config = config; } // field xml namespace public static void main(String[] args) { BusinessObject bo = new BusinessObject("09ae2d", 6, "sample"); } }</pre>	<pre><?xml version='1.0' encoding='UTF-8'?> <business-object xmlns="http://documentum.emc.com" xmlns:xcp="http://xcp.emc.com" xmlns:d2="http://d2.emc.com" xcp:vstamp="6"> <id>09ae2d</id> <d2:config >sample</d2:config> </business-object></pre>	<pre>{ "id": "09ae2d", "vstamp": 6, "config": "sample" }</pre>

@SerializableViewField4XmlList

This annotation provides dedicated support for java.util.List in XML. It uses existing List related attributes from @SerializableViewField and introduces new annotations. The @SerializableViewField4XmlList annotation can only be used with an XML List.

Table 9. @SerializableField4XmlList Attributes

Attribute	Description	Supported Formats	Default
boolean unwrap()	Specifies whether to unwrap the list or array items of this field as the direct members of the serializable type. Migrated from <i>@SerializableField.xmlListUnwrap</i> .	XML	false
boolean asPropBag()	Specifies the list items to be marshalled as a property bag or not.	XML	false
String itemName()	Specifies the list item element name within a <i>{@link java.util.List}</i> data type field. Migrated from <i>@SerializableField.xmlListItemName</i> .	XML	"item"

Note: `java.util.List` in JSON is always marshalled as a JSON array

@SerializableField4XmlMap

The *@SerializableField4XmlMap* annotation has been added to provide dedicated support for `java.util.List` with XML. This annotation can only be used with an XML map.

Table 10. @SerializableField4XmlMap Attributes

Attribute	Description	Supported Formats	Default
boolean asPropBag()	Specifies whether to serialize the key / value pair as a property bag or not.	XML	false

Attribute	Description	Supported Formats	Default
String propBagEleName()	Specifies the XML element name for each entry of <code>{@link java.util.Map}</code> data type field.	XML	"entry"
String propBagKeyName()	Specified the XML attribute name for each key of <code>{@link java.util.Map}</code> data type field.	XML	"key"

Note: When using JSON, `java.util.Map` is always marshalled as a JSON key-value object pair

Out-of-box Annotated Models

Documentum Platform 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`

Deprecations

The following annotations have been deprecated:

- **@SerializableEntry**
Java Map support has been added since version 7.3 of Documentum Platform REST Services. Therefore, `@SerializableEntry` has become redundant and is marked as `@Deprecated`. General key-value support is only available for Map objects.
- **@SerializableField#xmlListUnwrap and @SerializableField#xmlListItemName**
The annotation `@SerializableField` had attributes for XML specific support. Moving forward, all the functionality of `@SerializableField` has been incorporated into `@SerializableField4XmlList`.



Caution: Legacy Usage

Extended REST services that use `@SerializableEntry`, `@SerializableField.xmlListUnwrap` and `@SerializableField.xmlListItemName` can continue to use them moving forward. However, enhancements to these attributes will not be added.

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 7-16. Core Document Object

The following example illustrates how out-of-box annotated core document objects are serialized.

Java Definition	XML	JSON
<pre> DocumentObject doc = new DocumentObject(); doc.setType("dm_document"); doc.setTypeDef("http://localhost:8080/types/dm_document"); doc.addAttribute(new Attribute<String>("object_name", "Overview.pdf")); doc.addAttribute(new Attribute<List<String>>("authors", Arrays.asList("Bob", "Alice"))); doc.getLinks().add(new Link("self", "http://localhost:8080/documents/09acd2")); doc.getLinks().add(new Link("edit", "http://localhost:8080/documents/09acd2")); </pre>	<pre> <?xml version='1.0' encoding='UTF-8'?> <document xmlns="http://identifiers.emc.com/vocab/documentum" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="dm_document" definition="http://localhost:8080/types/dm_document"> <properties> <object_name>Overview.pdf</object_name> <authors> <item>Bob</item> <item>Alice</item> </authors> </properties> <links> <link rel="self" href="http://localhost:8080/documents/09acd2"/> <link rel="edit" href="http://localhost:8080/documents/09acd2"/> </links> </document> </pre>	<pre> { "name": "document", "type": "dm_document", "definition": "http://localhost:8080/types/dm_document", "properties": { "object_name": "Overview.pdf", "authors": ["Bob", "Alice"] }, "links": [{ "rel": "self", "href": "http://localhost:8080/documents/09acd2" }, { "rel": "edit", "href": "http://localhost:8080/documents/09acd2" }] } </pre>

Example 7-17. Core Repository Object

Example R-

The following example illustrates how out-of-box annotated core repository objects are serialized.

Java Definition	XML	JSON
<pre> Repository repo = new Repository(); repo.setId(15); repo.setName("acme"); repo.setDescription("ACME Company"); Server svr1 = new Server(); svr1.setName("CS715H"); svr1.setDocbroker("localhost"); svr1.setHost("CS715H"); svr1.setVersion("7.1.0010.0158 Win64.SQLServer"); repo.getServers().add(svr1); repo.getLinks().add(new Link("self", "http://localhost:8080/repositories/acme")); repo.getLinks().add(new Link("cabinets", "http://localhost:8080/repositories/acme/cabinets")); </pre>	<pre> <?xml version='1.0' encoding='UTF-8'?> <repository xmlns="http://identifiers.emc.com/vocab/documentum"> <id>15</id> <name>acme</name> <description>ACME Company</description> <servers> <server> <name>CS715H</name> <host>CS715H</host> <version>7.1.0010.0158 Win64.SQLServer</version> <docbroker>localhost</docbroker> </server> </servers> <links> <link rel="self" href="http://localhost:8080/repositories/acme"/> <link rel="cabinets" href="http://localhost:8080/repositories/acme/cabinets"/> </links> </repository> </pre>	<pre> { "id": 15, "name": "acme", "description": "ACME Company", "servers": [{ "name": "CS715H", "host": "CS715H", "version": "7.1.0010.0158 Win64.SQLServer", "docbroker": "localhost" }], "links": [{ "rel": "self", "href": "http://localhost:8080/repositories/acme" }, { "rel": "cabinets", "href": "http://localhost:8080/repositories/acme/" }] } </pre>

Example 7-18. Core Error Object

The following example illustrates how out-of-box annotated core error objects are serialized.

Java Definition	XML	JSON
<pre> RestError error = new RestError(); error.setCode("E_UNDEFINED_TYPE"); error.setStatus(400); error.setMessage("Undefined type is specified: dm_unknown"); error.addDetail("[E_BAD_TYPE] bad type 'dm_unknown' is specified for IDFSysobject"); error.addDetail("[E_DFC_OPERATION_ERROR] DFC operation error"); </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <error xmlns="http://identifiers.emc.com/vocab/documentum"> <status>400</status> <code>E_UNDEFINED_TYPE</code> <message>Undefined type is specified: dm_unknown</message> <details>[E_BAD_TYPE] bad type 'dm_unknown' is specified for IDFSysobject;[E_DFC_OPERATION_ERROR] DFC operation error </details> </error> </pre>	<pre> { "status": 400, "code": "E_UNDEFINED_TYPE", "message": "Undefined type is specified: dm_unknown", "details": "[E_BAD_TYPE] bad type 'dm_unknown' is specified for IDFSysobject;[E_DFC_OPERATION_ERROR] DFC operation error" } </pre>

Example 7-19. Core Atom Feed With Src Entry

The following example illustrates how out-of-box annotated atom feed objects with `src` entries are serialized.

Java Definition	XML	JSON
<pre> AtomFeed feed = new AtomFeed(); feed.setId("cabinets"); feed.setTitle("Cabinets in repository ACH"); feed.setPage(0); feed.setItemsPerPage(100); feed.setTotal(2); feed.setUpdated(new Date()); feed.getAuthors().add(new AtomAuthor("EMC Documentum")); AtomEntry entry1 = new AtomEntry(); entry1.setId("dc08ab"); entry1.setSummary("Dmadin"); entry1.setTitle("Dmadin's home cabinet"); entry1.setPublished(new Date()); entry1.setUpdated(new Date()); entry1.getAuthors().add(new AtomAuthor("Dmadin", "http://localhost:8080/users/dmadin", "dmadin@emc.com")); entry1.getAuthors().add(new AtomAuthor("tuser", "http://localhost:8080/users/tuser", "tuser@emc.com")); entry1.setContent(new AtomContent("application/xml", "http://localhost:8080/cabinets/dc08ab")); entry1.getLinks().add(new Link("self", "http://localhost:8080/cabinets/dc08ab")); feed.getEntries().add(entry1); AtomEntry entry2 = new AtomEntry(); entry2.setId("dc08ef"); entry2.setSummary("Bob"); entry2.setTitle("Bob's home cabinet"); entry2.setPublished(new Date()); entry2.setUpdated(new Date()); entry2.getAuthors().add(new AtomAuthor("alice", "http://localhost:8080/users/alice", "alice@emc.com")); entry2.setContent(new AtomContent("application/xml", "http://localhost:8080/cabinets/dc08ef")); entry2.getLinks().add(new Link("self", "http://localhost:8080/cabinets/dc08ef")); feed.getEntries().add(entry2); feed.getLinks().add(new Link("self", "http://localhost:8080/cabinets")); </pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <feed xmlns="http://www.w3.org/2005/Atom"> <id>cabinets</id> <title>Cabinets in repository ACH</title> <author> <name>EMC Documentum</name> </author> <updated>2014-04-08T22:53:50.686+08:00</updated> <page>0</page> <items-per-page>100</items-per-page> <total>2</total> <link rel="self" href="http://localhost:8080/cabinets"/> <entry> <id>dc08ab</id> <title>Dmadin's home cabinet</title> <author> <name>Dmadin</name> <uri>http://localhost:8080/users/dmadin</uri> <email>dmadin@emc.com</email> </author> <author> <name>tuser</name> <uri>http://localhost:8080/users/tuser</uri> <email>tuser@emc.com</email> </author> <summary>Dmadin</summary> <updated>2014-04-08T22:53:50.690+08:00</updated> <published>2014-04-08T22:53:50.686+08:00</published> <content content-type="application/xml" src="http://localhost:8080/cabinets/dc08ab"/> <link rel="self" href="http://localhost:8080/cabinets/dc08ab"/> <entry> <id>dc08ef</id> <title>Bob's home cabinet</title> <author> <name>bob</name> <uri>http://localhost:8080/users/bob</uri> <email>bob@emc.com</email> </author> <author> <name>alice</name> <uri>http://localhost:8080/users/alice</uri> <email>alice@emc.com</email> </author> <summary>Bob</summary> <published>2014-04-08T22:53:50.690+08:00</published> <content content-type="application/xml" src="http://localhost:8080/cabinets/dc08ef"/> <link rel="self" href="http://localhost:8080/cabinets/dc08ef"/> </entry> </entry> </feed> </pre>	<pre> { "id": "cabinets", "title": "Cabinets in repository ACH", "author": [{ "name": "EMC Documentum" }], "updated": "2014-04-08T22:53:50.686+08:00", "page": 0, "items-per-page": 100, "total": 2, "links": [{ "rel": "self", "href": "http://localhost:8080/cabinets" }], "entries": [{ "id": "dc08ab", "title": "Dmadin's home cabinet", "author": [{ "name": "Dmadin", "uri": "http://localhost:8080/users/dmadin", "email": "dmadin@emc.com" }, { "name": "tuser", "uri": "http://localhost:8080/users/tuser", "email": "tuser@emc.com" }], "summary": "Dmadin", "updated": "2014-04-08T22:53:50.690+08:00", "published": "2014-04-08T22:53:50.686+08:00", "content": { "content-type": "application/xml", "src": "http://localhost:8080/cabinets/dc08ab" }, "links": [{ "rel": "self", "href": "http://localhost:8080/cabinets/dc08ab" }] }, { "id": "dc08ef", "title": "Bob's home cabinet", "author": [{ "name": "bob", "uri": "http://localhost:8080/users/bob", "email": "bob@emc.com" }, { "name": "alice", "uri": "http://localhost:8080/users/alice", "email": "alice@emc.com" }], "summary": "Bob", "published": "2014-04-08T22:53:50.690+08:00", "content": { "content-type": "application/xml", "src": "http://localhost:8080/cabinets/dc08ef" }, "links": [{ "rel": "self", "href": "http://localhost:8080/cabinets/dc08ef" }] }] } </pre>

Example 7-20. Core Atom Feed With Inline Entry

The following example illustrates how out-of-box annotated atom feed objects with embedded entries are serialized.

[illegible]

Java Primitive Types Support

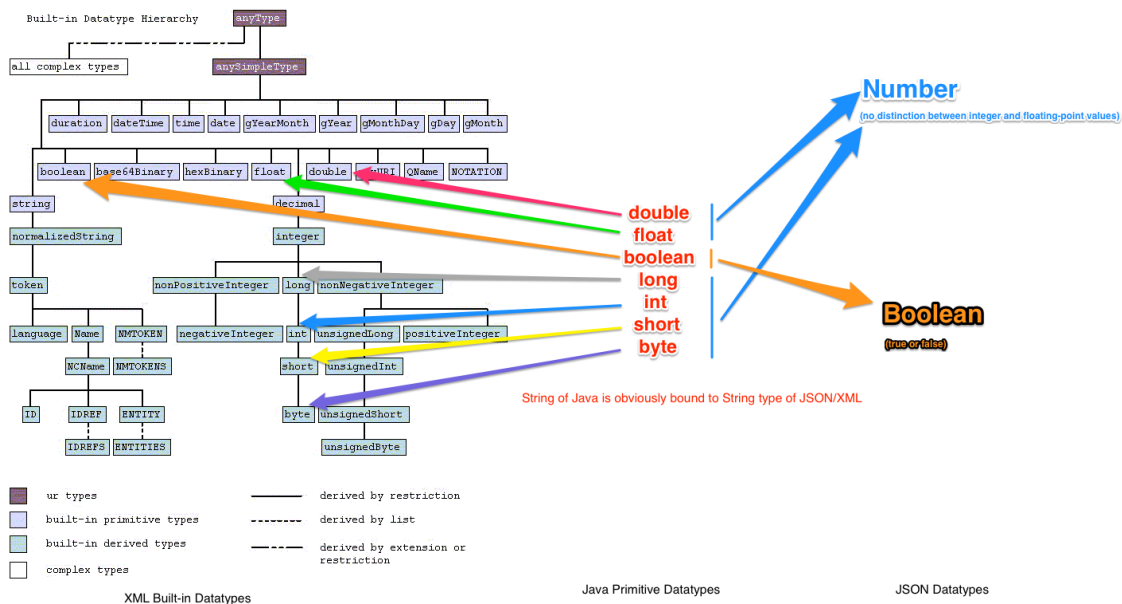
Documentum REST marshalling framework supports the following Java primitive types together with their wrapper classes:

Table 11. Supported Primitive Types and Wrapper Classes

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Note: 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 or JSON data types.

Figure 7. 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, following data types (and their wrapper classes) are represented in JSON by the `Number` data type:
 - `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`.

Java Interface, Abstract and Generic Types Support

Marshalling

For serialization (marshalling), the marshalling of a Java instance depends on the ability to serialize the data type of the runtime instance. Regardless of interface, abstract, or generic declaration of a class data field, the runtime class instance can be serialized as long as the instance's field types at runtime are serializable. For example, a concrete data type can be annotated by `@SerializableType`.

Here is a code sample that shows you how to do this:

Example 7-21. Annotating a Concrete Data Type

```
// Skipping all methods and constructors

// The root data type with generic field 'organization'
@SerializableType("bog")
public class BusinessObjectGeneric<T> {
    private T organization; // a generic field
}

// One organization implementation
@SerializableType("institution")
public class Institution {
    private String name;
    private Role role; // an abstract field
}

// Role interface
public interface Role { // interface
    String getName();
}

// One role implementation
@SerializableType("go")
public class GovernmentalOrg implements Role { // implementation of interface
    private String name;
    private int privilege;
}

// The other role implementation
@SerializableType("ngo")
public class NonGovernmentalOrg implements Role { // implementation of interface
```

```

    private String name;
    private String description;
}

```

Here's a code sample that shows you how to serialize a complex type:

Example 7-22. An Instance of `BusinessObjectGeneric 1`

```

BusinessObjectGeneric<Institution> origin =
    new BusinessObjectGeneric<Institution>(
        new Institution(
            "knowledge",
            new NonGovernmentalOrg(
                "wikimedia",
                "ubiquitous online encyclopaedia"
            )
        )
    );

```

Example 7-23. XML Representation for Sample 1

```

<bog>
  <organization>
    <name>knowledge</name>
    <role>
      <name>wikimedia</name>
      <description>ubiquitous online encyclopaedia</description>
    </role>
  </organization>
</bog>

```

Example 7-24. JSON Representation for Sample 1

```

{
  "organization":{
    "name":"knowledge",
    "role":{
      "name":"wikimedia",
      "description":"ubiquitous online encyclopaedia"
    }
  }
}

```

Example 7-25. An Instance of `BusinessObjectGeneric 2`

```

BusinessObjectGeneric<Institution> origin =
    new BusinessObjectGeneric<Institution>(
        new Institution(
            "funding",
            new GovernmentalOrg(
                "imf",
                8
            )
        )
    );

```

Example 7-26. XML Representation for Sample 2

```

<bog>
  <organization>
    <name>funding</name>
    <role>
      <name>imf</name>
      <priviledges>8</priviledges>
    </role>
  </organization>
</bog>

```

```
</role>
</organization>
</bog>
```

Example 7-27. JSON Representation for Sample 2

```
{
  "organization":{
    "name":"funding",
    "role":{
      "name":"imf",
      "priviledges":8
    }
  }
}
```



Caution: Unmarshalling the XML and JSON representations will not succeed because:

- In the XML representation, the `<organization>` tag did not interpret which implementation class to use during unmarshalling
- In the JSON representation, the key "organization" does not contain the class implementation information

Unmarshalling

You must add annotation metadata to the class definition to deserialize the data class. Adding this data type makes the type writable in XML or JSON.

- **For XML**

For XML, the serializable fields that have been declared as generic, interface, or abstract must set `@SerializableField#xmlWriteRoot = true`. The concrete data type for the field is written into XML as the XML element name. Therefore, it can be resolved during the deserialization process.



Caution: To deserialize the generic, interface or abstract field in XML, the data class cannot have more than one field that contains non-concrete data types.

Here's a code sample that demonstrates how to do this:

```
// xmlWriteRoot=true

@SerializableField(xmlWriteTypeRoot = true)
private T organization;

@SerializableField(xmlWriteTypeRoot = true)
private Role role;
```

- **For JSON**

For JSON, the serializable fields that have been declared as generic, interface or abstract must set `@SerializableField#jsonWriteRoot = true`. The concrete data type for the field

is written into XML as the XML element name. Therefore, it can be resolved during the deserialization process.

Here's a code sample that demonstrates how to do this:

```
// jsonWriteRoot=true

@SerializableType(value = "institution", jsonWriteRootAsField = true, jsonRootField = "type")
public static class Institution {...}

@SerializableType(value = "go", jsonWriteRootAsField = true, jsonRootField = "type")
public static class GovernmentalOrg implements Role {...}

@SerializableType(value = "ngo", jsonWriteRootAsField = true, jsonRootField = "type")
public static class NonGovernmentalOrg implements Role {...}

The XML and JSON contain additional type information that allows it be deserialized. Here's
the same code with all the changes:
```

Example 7-28. An Instance of a Writable `BusinessObjectGeneric` 1

```
BusinessObjectGeneric<Institution> origin =
    new BusinessObjectGeneric<Institution>(
        new Institution(
            "knowledge",
            new NonGovernmentalOrg(
                "wikimedia",
                "ubiquitous online encyclopaedia"
            )
        )
    );
```

Example 7-29. XML Representation for a Writable Sample 1

```
<bog>
  <institution>
    <name>knowledge</name>
    <ngo>
      <name>wikimedia</name>
      <description>ubiquitous online encyclopaedia</description>
    </ngo>
  </institution>
</bog>
```

Note: The serializable name `institution` that is used for the actual data type `Institution` is written as XML instead of using the generic field name `organization`.

The serializable name `ngo` that is used for the actual data type `NonGovernmentalOrg` is written as XML instead of using the generic field name `role`.

Example 7-30. JSON Representation for a Writable Sample 1

```
{
  "organization":{
    "type":"institution",
    "name":"knowledge",
    "role":{
      "type":"ngo",
      "name":"wikimedia",
      "description":"ubiquitous online encyclopaedia"
    }
  }
}
```

Note: The root information type: `institution` that is used for the actual data type `Institution` is written to the JSON object of `organization`.

The root information type: `ngo` that is used for the actual data type `NonGovernmentalOrg` is written to the JSON object of `role`.

Example 7-31. An Instance of a Writable `BusinessObjectGeneric` 2

```
BusinessObjectGeneric<Institution> origin =  
    new BusinessObjectGeneric<Institution>(  
        new Institution(  
            "funding",  
            new GovernmentalOrg(  
                "imf",  
                8  
            )  
        )  
    );
```

Note: The serializable name `institution` that is used for the actual data type `Institution` is written as XML instead of using the generic field name `organization`.

The serializable name `go` that is used for the actual data type `GovernmentalOrg` is written as XML instead of using the generic field name `role`.

Example 7-32. XML Representation for a Writable Sample 2

```
<bog>  
  <institution>  
    <name>funding</name>  
    <go>  
      <name>wikimedia</name>  
      <priviledges>8</description>  
    </go>  
  </institution>  
</bog>
```

Example 7-33. JSON Representation for a Writable Sample 2

```
{  
  "organization":{  
    "type":"institution",  
    "name":"funding",  
    "role":{  
      "type":"go",  
      "name":"imf",  
      "priviledges":8  
    }  
  }  
}
```

Note: The root information type: `institution` that is used for the actual data type `Institution` is written to the JSON object of `organization`.

The root information type: `go` that is used for the actual data type `GovernmentalOrg` is written to the JSON object of `role`.

Example 7-34. Complete Class Definition

```
/*  
 * Copyright (c) 2016. EMC Corporation. All Rights Reserved.  
 */
```

```

package com.emc.documentum.rest.mock;

import org.apache.commons.lang.builder.EqualsBuilder;

import com.emc.documentum.rest.binding.SerializableField;
import com.emc.documentum.rest.binding.SerializableType;

@SerializableType("bog")
public class BusinessObjectGeneric<T> {

    @SerializableField(xmlWriteTypeRoot = true)
    private T organization;

    public BusinessObjectGeneric() {}

    public BusinessObjectGeneric(T organization) {
        this.organization = organization;
    }

    public T getOrganization() {
        return this.organization;
    }

    @Override
    public boolean equals(Object other) {
        return EqualsBuilder.reflectionEquals(this, other);
    }

    @SerializableType(value = "institution", jsonWriteRootAsField = true,
        jsonRootField = "type")
    public static class Institution {
        private String name;
        @SerializableField(xmlWriteTypeRoot = true)
        private Role role;

        public Institution() {}

        public Institution(String name, Role role) {
            this.name = name;
            this.role = role;
        }

        public String getName() {
            return name;
        }

        public Role getRole() {
            return role;
        }

        @Override
        public boolean equals(Object other) {
            return EqualsBuilder.reflectionEquals(this, other);
        }
    }

    @SerializableType(value = "company", jsonWriteRootAsField = true,
        jsonRootField = "type")
    public static class Company {
        private String name;
        @SerializableField(xmlWriteTypeRoot = true)
        private Industry industry;

        public Company() {}
    }
}

```

```
        public Company(String name, Industry industry) {
            this.name = name;
            this.industry = industry;
        }

        public String getName() {
            return name;
        }

        public Industry getIndustry() {
            return industry;
        }

        @Override
        public boolean equals(Object other) {
            return EqualsBuilder.reflectionEquals(this, other);
        }
    }

    public static interface Role {
        String getName();
    }

    @SerializableType(value = "go", jsonWriteRootAsField = true, jsonRootField = "type")
    public static class GovernmentalOrg implements Role {
        private String name;
        private int privilege;

        public GovernmentalOrg() {}

        public GovernmentalOrg(String name, int privilege) {
            this.name = name;
            this.privilege = privilege;
        }

        @Override public String getName() {
            return name;
        }

        public int getPrivilege() {
            return privilege;
        }

        @Override
        public boolean equals(Object other) {
            return EqualsBuilder.reflectionEquals(this, other);
        }
    }

    @SerializableType(value = "ngo", jsonWriteRootAsField = true, jsonRootField = "type")
    public static class NonGovernmentalOrg implements Role {
        private String name;
        private String description;

        public NonGovernmentalOrg() {}

        public NonGovernmentalOrg(String name, String description) {
            this.name = name;
            this.description = description;
        }

        @Override public String getName() {
            return name;
        }
    }
```



```

        public String getDescription() {
            return description;
        }

        @Override
        public boolean equals(Object other) {
            return EqualsBuilder.reflectionEquals(this, other);
        }
    }

    @SerializableType
    public static abstract class Industry {
        @SerializableField("country")
        public String countryCode;

        public String getCountryCode() {
            return countryCode;
        }

        @Override
        public boolean equals(Object other) {
            return EqualsBuilder.reflectionEquals(this, other);
        }
    }

    @SerializableType(value = "it", jsonWriteRootAsField = true, jsonRootField = "type")
    public static class InformationTechnology extends Industry {
        private String scale;

        public InformationTechnology() {}

        public InformationTechnology(String scale, String countryCode) {
            super.countryCode = countryCode;
            this.scale = scale;
        }

        public String getScale() {
            return scale;
        }

        @Override
        public boolean equals(Object other) {
            return EqualsBuilder.reflectionEquals(this, other);
        }
    }

    @SerializableType(value = "pharm", jsonWriteRootAsField = true,
        jsonRootField = "type")
    public static class Pharmaceuticals extends Industry {
        private String field;

        public Pharmaceuticals() {
        }

        public Pharmaceuticals(String field, String countryCode) {
            super.countryCode = countryCode;
            this.field = field;
        }

        public String getField() {
            return field;
        }

        @Override
        public boolean equals(Object other) {

```

```
        return EqualsBuilder.reflectionEquals(this, other);
    }
}
```

Java Collection and Array Support

Array Serialization 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 Serialization Support

The marshalling framework supports `List` and its subtypes. In JSON representation, a `List` is bound to an `Array`. In XML representation, each element in a `List` is a sub-element of the element representing the `List`.

For more information, see [Example K](#) for details.

Supported Data Types

The following data types are supported as the parameters of a `List` or `Array`:

- Declared native data types such as `List<String>`, `List<Date>`, `List<Integer>`, `List<Double>`, `List<Boolean>`
- Declared enum data types such as `List<EnumUserTypes>`, `List<EnumDocTypes>`
- Declared complex data types with the `@SerializableType` annotation such as `List<AtomFeed>`, `List<Link>`, `List<PersistentObject>`
- Interface such as `List<Folder>`, `List<User>`
- Abstract class such as `List<AbstractFolder>`, `List<AbstractDocument>`
- Generic or wild card object data types such as `List<Object>`, `List<?>`, `List<T>`



Caution: Runtime check of parameterized data type

Whether or not instances of *Interface*, *Abstract class*, and *Generic or wild card* can be marshalled is determined at runtime by the data type of the instance.

Exclusions

The following parameterized data types are not supported:

- List or Array in a List
For example `List<List<String>>`, `List<String[]>`
- Map data type in a List `List<Map<String,String>>`
- Any other data type



Caution: List item order

Items in a list are marshalled according to their order within the list

`java.util.Collection` is not supported

The generic collection data type `java.util.Collection<>` is not supported. The List data type must be an Array, `java.util.List`, or a sub type of `java.util.List`

Marshalling with XML

The marshalling of an XML List or Array is more complex than marshalling a JSON List or Array because of the following:

- XML fields can have separate namespaces
- The XML fields for a List or Array can be wrapped or unwrapped
- The XML fields for a List or Array of complex data types can be serialized as property bags or standalone XML objects.

Example 7-35. Native Data Types as XML Elements

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    private List<String> keywords;
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getKeywords().add("2015");
boa.getKeywords().add("bedrock");
```

XML Code

```
<boa>
  <keywords>
    <item>2015</item>
    <item>bedrock</item>
  </keywords>
</boa>
```

Example 7-36. Array of Enums as XML Elements

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
```

```

    private LABEL[] labels;

    public static enum LABEL {
        DEV, PRODUCT, SUPPORT
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.setLabels(BusinessObjectArchive.LABEL.DEV,
    BusinessObjectArchive.LABEL.SUPPORT);

```

XML Code

```

<boa>
  <labels>
    <item>DEV</item>
    <item>SUPPORT</item>
  </labels>
</boa>

```

Example 7-37. List of Complex Data Type as XML Elements

Java Code

```

@SerializableType("boa")
public class BusinessObjectArchive {
    private List<Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));

```

XML Code

```

<boa>
  <roles>
    <admin-role>
      <name>admin</name>
      <privilege>16</privilege>
    </admin-role>
    <consumer-role>
      <name>ios</name>
      <description>iOS mobile device</description>
    </consumer-role>
  </roles>
</boa>

```

Example 7-38. List of Complex Data Type as XML property-bag Elements

Java Code

```

@SerializableType("boa")

```

```
public class BusinessObjectArchive {
    @SerializableField4XmlList(asPropBag = true, itemName="role")
    private List<Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));
```

XML Code

```
<boa>
  <roles>
    <role>
      <name>admin</name>
      <privilege>16</privilege>
    </role>
    <role>
      <name>ios</name>
      <description>iOS mobile device</description>
    </role>
  </roles>
</boa>
```

Example 7-39. List of Native Data Type as XML Unwrapped Elements

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    @SerializableField4XmlList(unwrap = true, itemName="keyword")
    private List<String> keywords;
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getKeywords().add("2015");
boa.getKeywords().add("bedrock");
```

XML Code

```
<boa>
  <keyword>2015</keyword>
  <keyword>bedrock</keyword>
</boa>
```

Example 7-40. List of Complex Data Type as XML Elements with namespaces

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    @SerializableField4XmlList(asPropBag = false)
    private List<Role> roles;
```

```

public static interface Role {}

@SerializableType(value = "admin-role", xmlns = "http://acme.org", xmlnsPrefix = "am")
public static class AdminRole implements Role {
    private String name;
    private int privilege;
}

@SerializableType(value = "consumer-role", xmlns = "http://d2.org",
    xmlnsPrefix = "d2")
public static class ConsumerRole implements Role {
    private String name;
    private String description;
}

...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));

```

XML Code

```

<boa>
  <roles>
    <am:admin-role xmlns:am="http://acme.org">
      <am:name>admin</am:name>
      <am:privilege>16</am:privilege>
    </am:admin-role>
    <d2:consumer-role xmlns:d2="http://d2.org">
      <d2:name>ios</d2:name>
      <d2:description>iOS mobile device</d2:description>
    </d2:consumer-role>
  </roles>
</boa>

```

Example 7-41. List of complex Data Type as XML unwrapped property—bag Elements**Java Code**

```

@SerializableType("boa")
public class BusinessObjectArchive {
    @SerializableField4XmlList(unwrapped=true, asPropBag = true, itemName="role")
    private List<Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }

    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));

```

XML Code

```
<boa>
  <role>
    <name>admin</name>
    <privilege>16</privilege>
  </role>
  <role>
    <name>ios</name>
    <description>iOS mobile device</description>
  </role>
</boa>
```

Marshalling with JSON

The data types mentioned in the preceding topic can be serialized (marshalled) into a JSON array. The following code samples show you how to serialize different data types:

Example 7-42. Native Data Types as a JSON array

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    private List<String> keywords;
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getKeywords().add("2016");
boa.getKeywords().add("Bedrock");
```

JSON Code

```
{
  "keywords":["2016","Bedrock"]
}
```

Example 7-43. Enum Array as a JSON array

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    private LABEL[] labels;

    public static enum LABEL {
        DEV, PRODUCT, SUPPORT
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.setLabels(BusinessObjectArchive.LABEL.DEV,
    BusinessObjectArchive.LABEL.SUPPORT);
```

JSON Code

```
{
  "labels":["DEV","SUPPORT"]
}
```

Example 7-44. Interface as a JSON array

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    private List<Role> roles;
```



```

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));

```

JSON Code

```

{
    "roles":[
        {"name":"admin","privilege":16},
        {"name":"ios","description":"iOS mobile device"}
    ]
}

```

Example 7-45. Abstract class as a JSON array**Java Code**

```

@SerializableType("boa")
public class BusinessObjectArchive {
    private List<Industry> industries;

    @SerializableType
    public static abstract class Industry {
        protected String industry;
    }

    @SerializableType(value = "healthcare")
    public static class Healthcare extends Industry {
        @SerializableField("cc")
        private int countryCode;
    }

    @SerializableType(value = "energy")
    public static class Energy extends Industry {
        private String type;
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getIndustries().add(new BusinessObjectArchive.Healthcare("hc", 1));
boa.getIndustries().add(new BusinessObjectArchive.Energy("en", "nuclear"));

```

JSON Code

```

{
    "industries":[
        {"cc":1,"industry":"hc"},
        {"type":"nuclear","industry":"en"}
    ]
}

```

Unmarshalling with XML and JSON

A List or Array can be deserialized using JSON or XML messages. However, there are certain constraints with respect to the class definition.

The following field types can be deserialized without constraints:

- **Declared Native data types**

`List<String>, List<Date>, List<Integer>, List<Double>, List<Boolean>`

- **Declared Enum data types**

`List<EnumUserTypes>, List<EnumDocTypes>`

- **Declared complex data types using the `@SerializableType` annotation**

`List<AtomFeed>, List<Link>, List<PersistentObject>`

The following data types can only be deserialized using annotation attributes for XML or JSON:

- **Interface**

`List<IFolder>, List<IUser>`

- **Abstract class**

`List<AbstractFolder>, List<AbstractDocument>`

- **Generic or wild card data types**

`List<Object>, List<?>, List<T>`

Unmarshalling with XML

The following data types can be deserialized (unmarshalled) using XML:

- **Native data types**
- **Enum data type**
- **Complex data types**
- **Native data types as XML unwrapped elements**
- **Complex data types as XML elements with namespaces**

Messages of the following data types cannot be deserialized (unmarshalled) using XML because their data type is not determined from their declared data type or XML message:

- **Complex data type as XML property-bag elements**
- **Complex data type as XML elements with namespaces**

To make a List of Interface, Abstract, or Wild card items deserializable, their fields must be marked as *wrapped* and *non-property bag* by setting the `@SerializableField4XmlList.asPropBag` and `@SerializableField4XmlList.unwrap` annotations.

Here's a code sample that shows you how to do that:

```
@SerializableField4XmlList.asPropBag=false
@SerializableField4XmlList.unwrap=false
```

This code sample shows you how to resolve the preceding limitation so you can deserialize a complex data type as XML elements:

Example 7-46. List of complex Data Type as XML Elements

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    private List<Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }
    ...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));
```

XML Code

```
<boa>
  <roles>
    <admin-role>
      <name>admin</name>
      <privilege>16</privilege>
    </admin-role>
    <consumer-role>
      <name>ios</name>
      <description>iOS mobile device</description>
    </consumer-role>
  </roles>
</boa>
```

This code sample shows you how to resolve the preceding limitation so you can deserialize a complex data type as XML elements with namespaces:

Example 7-47. List of complex Data Type as XML Elements with namespaces

Java Code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    @SerializableField4XmlList(asPropBag = false)
    private List<Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role", xmlns = "http://acme.org", xmlnsPrefix = "am")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }
}
```

```
@SerializableType(value = "consumer-role", xmlns = "http://d2.org",
    xmlnsPrefix = "d2")
public static class ConsumerRole implements Role {
    private String name;
    private String description;
}
...
}
BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));
```

XML Code

```
<boa>
  <roles>
    <am:admin-role xmlns:am="http://acme.org">
      <am:name>admin</am:name>
      <am:privilege>16</am:privilege>
    </am:admin-role>
    <d2:consumer-role xmlns:d2="http://d2.org">
      <d2:name>ios</d2:name>
      <d2:description>iOS mobile device</d2:description>
    </d2:consumer-role>
  </roles>
</boa>
```

Unmarshalling with JSON

The following data types can be deserialized (unmarshalled) using JSON:

- **Native data types**
- **Enum data type**

Messages of the following data types cannot be deserialized (unmarshalled) using JSON because their data type is not determined from their declared data type or the JSON message:

- **Interfaces**
- **Abstract classes**

To make a List of Interface, Abstract, or Wild card items deserializable, the item object definition must enable JSON write root. To enable JSON write root, set the following:

```
@SerializableType.jsonWriteRootAsField=true
```

Here's a code sample that shows you how to do that in more detail:

JSON code

```
{
  "roles":[
    {"type":"admin-role", "name":"admin","privilege":16},
    {"type","consumer-role", "name":"ios","description":"iOS mobile device"}
  ]
}
```

Java code

```
@SerializableType("boa")
public class BusinessObjectArchive {
    private List<Role> roles;

    public static interface Role {}
```

```

@SerializableType(value = "admin-role",
    jsonWriteRootAsField = true, jsonRootField = "type")
public static class AdminRole implements Role {
    private String name;
    private int privilege;
}

@SerializableType(value = "consumer-role",
    jsonWriteRootAsField = true, jsonRootField = "type")
public static class ConsumerRole implements Role {
    private String name;
    private String description;
}
}

BusinessObjectArchive boa = new BusinessObjectArchive();
boa.getRoles().add(new BusinessObjectArchive.AdminRole("admin", 16));
boa.getRoles().add(new BusinessObjectArchive
    .ConsumerRole("ios", "iOS mobile device"));

```

Note: The annotation `@SerializableType` for field data types have been set as `jsonWriteRootAsField = true, jsonRootField = "type"`

Summary of Support for Java List Data Type

List Item Data Type	Marshalling	Unmarshalling
Simple data types: <ul style="list-style-type: none"> • String • Date • Integer • Double • Boolean • Enum 	Yes	Yes
Complex type annotated by <code>@SerializableType</code>	Yes	Yes
<ul style="list-style-type: none"> • Interface • Abstract Class • Generic type • Wild card type 	Yes but only when the data type in the runtime instance is serializable	Yes but only when the data type in the runtime instance is serializable Specifically for JSON when: <ul style="list-style-type: none"> • The JSON root is written Specifically for XML when: <ul style="list-style-type: none"> • Wrapped

List Item Data Type	Marshalling	Unmarshalling
		<ul style="list-style-type: none">Written as a non property bag
None of above	No	No

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.

Java Map Support

Java Map Data Type Support

Documentum Platform REST Services version 7.3 and later have support for `java.util.Map`.

```
@SerializableField
private Map<String, String> countryCodes;
```

JSON objects can be marshalled as JSON key value pairs. For example:

```
"countryCodes": {
  "usa" : "001",
  "china" : "086"}
```

XML objects can also be marshalled using property bags. For example:

```
<countryCodes>
  <countryCode>
    <country name="usa">001</country>
    <country name="china">086</country>
  </countryCode>
</countryCodes>
```

Supported Data Type



Caution: To be consistent for XML and JSON marshalling, the map key must be a String data type

Inclusions

The first argument (key) data type in a Map must have a String data type. The second argument (value) data type in a Map can be any one of the following data types:

- **Declared native data type**

```
Map<String, String>, Map<String, Date>, Map<String, Integer>,
Map<String, Double>, Map<String, Boolean>
```

- **Declared enum data type**

```
Map<String, EnumUserTypes>, Map<String, EnumDocTypes>
```

- **Declared List data type**

```
Map<String, List<String>>, Map<String, List<?>>
```

- **Declared complex data type with annotation @SerializableType**

```
Map<String, AtomFeed>, Map<String, Link>, Map<String, PersistentObject>
```

- **Interface**

```
Map<String, IFolder>, Map<String, IUser>
```

- **Abstract class**

```
Map<String, AbstractFolder>, Map<String, AbstractDocument>
```

- **Generic or wild object data type**

```
Map<String, Object>, Map<String, ?>, Map<String, T>
```



Caution: Runtime check of parameterized data type

Whether or not instances of *List*, *Interface*, *Abstract class*, and *Generic or wild card* objects can be marshalled is determined at runtime by the data type of the instance.

Marshalling

Marshalling with XML

Marshalling a Map with XML is more complex than JSON due to the following:

- XML fields can have separate namespaces
- XML fields for a Map of complex types can be serialized as property bags or standalone XML objects

Here are some code samples that show you how to serialize (marshall) with XML:

Example 7-48. Map of Native Data Type as XML Elements

Java Code

```
@SerializableType("bom")
public class BusinessObjectMap {
    Map<String, String> codes;
```

```
...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getCodes().put("usa", "001");
bom.getCodes().put("china", "086");
```

XML Code

```
<bom>
  <codes>
    <usa>001</usa>
    <china>086</china>
  </codes>
</bom>
```

Example 7-49. Map of Complex Data Type as XML Elements**Java Code**

```
@SerializableType("bom")
public class BusinessObjectMap {
    @SerializableField(defaultImpl = TreeMap.class)
    Map<String, Industry> industries;

    @SerializableType
    public static abstract class Industry {
        protected String industry;
    }

    @SerializableType(value = "healthcare")
    public static class Healthcare extends Industry {
        @SerializableField("cc")
        private int countryCode;
    }

    @SerializableType(value = "energy")
    public static class Energy extends Industry {
        private String field;
    }
}

...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getIndustries().put("s1", new BusinessObjectMap.Healthcare("hc", 1));
bom.getIndustries().put("s2", new BusinessObjectMap.Energy("en", "nuclear"));
```

XML Code

```
<bom>
  <industries>
    <s1>
      <healthcare>
        <cc>1</cc>
        <industry>hc</industry>
      </healthcare>
    </s1>
    <s2>
      <energy>
        <field>nuclear</field>
        <industry>en</industry>
      </energy>
    </s2>
  </industries>
</bom>
```

Example 7-50. Map of Array of Native Data Type as XML Elements**Java Code**


```

@SerializableType("bom")
public class BusinessObjectMap {
    Map<String, String[]> codes;
    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getCodes().put("asia", new String[]{"086", "091"});
bom.getCodes().put("west", new String[]{"001", "044"});

```

XML Code

```

<bom>
  <codes>
    <asia>
      <item>086</item>
      <item>091</item>
    </asia>
    <west>
      <item>001</item>
      <item>044</item>
    </west>
  </codes>
</bom>

```

Example 7-51. Map of List of Complex Data Type as XML Elements

Java Code

```

@SerializableType("bom")
public class BusinessObjectMap {
    Map<String, List<Role>> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }
    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getRoles().put("r1", Arrays.asList(
    new BusinessObjectMap.AdminRole("admin", 16),
    new BusinessObjectMap.ConsumerRole("ios", "iOS mobile device")
));
bom.getRoles().put("r2", Arrays.asList(
    new BusinessObjectMap.AdminRole("dbowner", 8),
    new BusinessObjectMap.ConsumerRole("android", "Android mobile device")
));

```

XML Code

```

<bom>
  <roles>
    <r1>
      <admin-role>
        <name>admin</name>
        <privilege>16</privilege>
      </admin-role>
      <consumer-role>

```

```

        <name>ios</name>
        <description>iOS mobile device</description>
    </consumer-role>
</r1>
<r2>
    <admin-role>
        <name>dbowner</name>
        <privilege>8</privilege>
    </admin-role>
    <consumer-role>
        <name>android</name>
        <description>Android mobile device</description>
    </consumer-role>
</r2>
</roles>
</bom>

```

Example 7-52. Map of Native Data Type as XML prop-bag Elements**Java Code**

```

@SerializableType("bom")
public class BusinessObjectMap {
    @SerializableField4XmlMap(asPropBag = true)
    Map<String, String> codes;
    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getCodes().put("usa", "001");
bom.getCodes().put("china", "086");

```

XML Code

```

<bom>
    <codes>
        <entry key="usa">001</entry>
        <entry key="china">086</entry>
    </codes>
</bom>

```

Example 7-53. Map of Complex Data Type as XML prop-bag Elements**Java Code**

```

@SerializableType("bom")
public class BusinessObjectMap {
    @SerializableField(defaultImpl = TreeMap.class)
    @SerializableField4XmlMap(asPropBag = true,
        propBagEleName = "industry",
        propBagKeyName = "type")
    Map<String, Industry> industries;

    @SerializableType
    public static abstract class Industry {
        protected String industry;
    }

    @SerializableType(value = "healthcare")
    public static class Healthcare extends Industry {
        @SerializableField("cc")
        private int countryCode;
    }

    @SerializableType(value = "energy")
    public static class Energy extends Industry {
        private String field;
    }
}

```

```

    }
    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getIndustries().put("s1", new BusinessObjectMap.Healthcare("hc", 1));
bom.getIndustries().put("s2", new BusinessObjectMap.Energy("en", "nuclear"));

```

XML Code

Marshalling with JSON

When marshalling a Map with JSON, all map entries are serialized as key-value pairs of a JSON object.

Example 7-54. Map of Native Data Types as JSON Fields

Java Code

```

@SerializableType("bom")
public class BusinessObjectMap {
    Map<String, String> codes;

}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getCodes().put("usa", "001");
bom.getCodes().put("china", "086");

```

JSON Code

```

{
  "codes":{
    "usa":"001",
    "china":"086"
  }
}

```

Example 7-55. Complex Data Type as JSON Fields

```

@SerializableType("bom")
public class BusinessObjectMap {
    Map<String, Industry> industries;

    @SerializableType
    public static abstract class Industry {
        protected String industry;
    }

    @SerializableType(value = "healthcare")
    public static class Healthcare extends Industry {
        @SerializableField("cc")
        private int countryCode;
    }

    @SerializableType(value = "energy")
    public static class Energy extends Industry {
        private String type;
    }
    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getIndustries().put("s1", new BusinessObjectMap.Healthcare("hc", 1));
bom.getIndustries().put("s2", new BusinessObjectMap.Energy("en", "nuclear"));

```

JSON Code

```
{
  "industries": {
    "s2": {
      "type": "nuclear",
      "industry": "en"
    },
    "s1": {
      "cc": 1,
      "industry": "hc"
    }
  }
}
```

Example 7-56. List of Complex Data Type as JSON Fields

JAVA Code

```
@SerializableType("bom")
public class BusinessObjectMap {
    private Map<String, Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }

    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getRoles().put("r1", Arrays.asList(
    new BusinessObjectMap.AdminRole("admin", 16),
    new BusinessObjectMap.ConsumerRole("ios", "iOS mobile device")
));
bom.getRoles().put("r2", Arrays.asList(
    new BusinessObjectMap.AdminRole("dbowner", 8),
    new BusinessObjectMap.ConsumerRole("android", "Android mobile device")
));
{
```

JSON Code

```
"roles": {
  "r1": [
    {
      "name": "admin",
      "privilege": 16
    },
    {
      "name": "ios",
      "description": "iOS mobile device"
    }
  ],
  "r2": [
    {
      "name": "dbowner",
      "privilege": 8
    },
    {
```

```

        "name": "android",
        "description": "Android mobile device"
    }
}
}

```

Unmarshalling

A Map can be deserialized using JSON or XML messages but it has certain constraints with respect to the class definition.

The following field types can always be deserialized:

- Declared native data type as the Map value
`Map<String, String>, Map<String, Date>, Map<String, Integer>,
Map<String, Double>, Map<String, Boolean>`
- Declared Enum data type
`Map<String, Enum User Types>, Map<String, Enum DocTypes>`
- Declared List data type with concrete List item type
`Map<String, List<String>>`
- Declared complex data type with annotation `@SerializableType`
`Map<String, AtomFeed>, Map<String, Link>, Map<String, PersistentObject>`

The following field data types can be deserialized using additional annotation attributes for XML and JSON, respectively:

- **Interface**
`Map<String, IFolder>, Map<String, IUser>`
- **Abstract class**
`Map<String, AbstractFolder>, Map<String, AbstractDocument>`
- **Generic or wild card data type**
`Map<String, Object>, Map<String, ?>, Map<String, T>`
- **Declared List data type with unresolved list item type**
`Map<String, List<? extends IUser>>`

Unmarshalling with XML

For XML, the Map entry value is always deserializable and there are no constraints. Messages for the following data types can be deserialized to a Map field:

- Native data type as XML elements
- Complex data type as XML elements
- Array of native data types as XML elements

- List of complex data type as XML elements
- Simple data type as XML prop-bag elements
- Complex data type as XML prop-bag elements

Unmarshalling with JSON

The following data types can be deserialized:

- Native data type as JSON fields
- Complex data type as JSON fields

A List of complex data type cannot be deserialized into JSON fields because the type information is not determined from either the declared data type such as `Map<String, Role>`, or the JSON message such as `{name:xx, ..}`.

To make Map of Interface, Abstract class, and Wild card items deserializable, the object definition for the item must enable the JSON write root attribute as:

```
@SerializableType.jsonWriteRootAsField=true
```

Here's a code sample that shows you how to do that:

Example 7-57. Read a Map of Interfaces from JSON

Java Code

```
@SerializableType("bom")
public class BusinessObjectMap {
    private Map<String, Role> roles;

    public static interface Role {}

    @SerializableType(value = "admin-role",
                      jsonWriteRootAsField = true,
                      jsonRootField = "type")
    public static class AdminRole implements Role {
        private String name;
        private int privilege;
    }

    @SerializableType(value = "consumer-role",
                      jsonWriteRootAsField = true,
                      jsonRootField = "type")
    public static class ConsumerRole implements Role {
        private String name;
        private String description;
    }
    ...
}
BusinessObjectMap bom = new BusinessObjectMap();
bom.getRoles().put("r1", Arrays.asList(
    new BusinessObjectMap.AdminRole("admin", 16),
    new BusinessObjectMap.ConsumerRole("ios", "iOS mobile device")
));
bom.getRoles().put("r2", Arrays.asList(
    new BusinessObjectMap.AdminRole("dbowner", 8),
    new BusinessObjectMap.ConsumerRole("android", "Android mobile device")
));
```

Note: The annotation `@SerializableType` for field data types has been set to `jsonWriteRootAsField = true, jsonRootField = "type"`.

JSON Code

```
{
  "roles": {
    "r1": [
      {
        "type": "admin-role",
        "name": "admin",
        "privilege": 16
      },
      {
        "type": "consumer-role",
        "name": "ios",
        "description": "iOS mobile device"
      }
    ],
    "r2": [
      {
        "type": "admin-role",
        "name": "dbowner",
        "privilege": 8
      },
      {
        "type": "consumer-role",
        "name": "android",
        "description": "Android mobile device"
      }
    ]
  }
}
```

Note: The `type` attribute is written into the JSON object.

Summary of support for the Map Data Type

The following table shows Map data type support:

Map Entry Value Data Type	Marshalling	Unmarshalling
Simple data types: String, Date, Integer, Double, Boolean, Enum	Yes	Yes
Complex type annotated by <code>@SerializableType</code>	Yes	Yes
Interface, Abstract Class Generic type or Wild card type	Yes but only when the data type in the runtime instance is serializable	Yes but only when the data type in the runtime instance is serializable Specifically for JSON when: <ul style="list-style-type: none"> • json root is written

Map Entry Value Data Type	Marshalling	Unmarshalling
List data type with item type as above data types	Yes	Accordingly
None of above	No	No

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.

Custom Serializers and Deserializers

The unified data binding framework allows you to customize marshalling and unmarshalling data model in many aspects.

There are requirements at the field level that cannot be fulfilled by the out-of-box capability of the framework.

Here are several typical cases:

1. Representation of different media types is different, like the `AtomEntry` representation of XML and JSON media types
2. Data binding depends on runtime data. For example, don't marshall a wrapped list when it is empty
3. A field of the class must be written out during marshalling, but shouldn't be read in during unmarshalling

Programming Interface

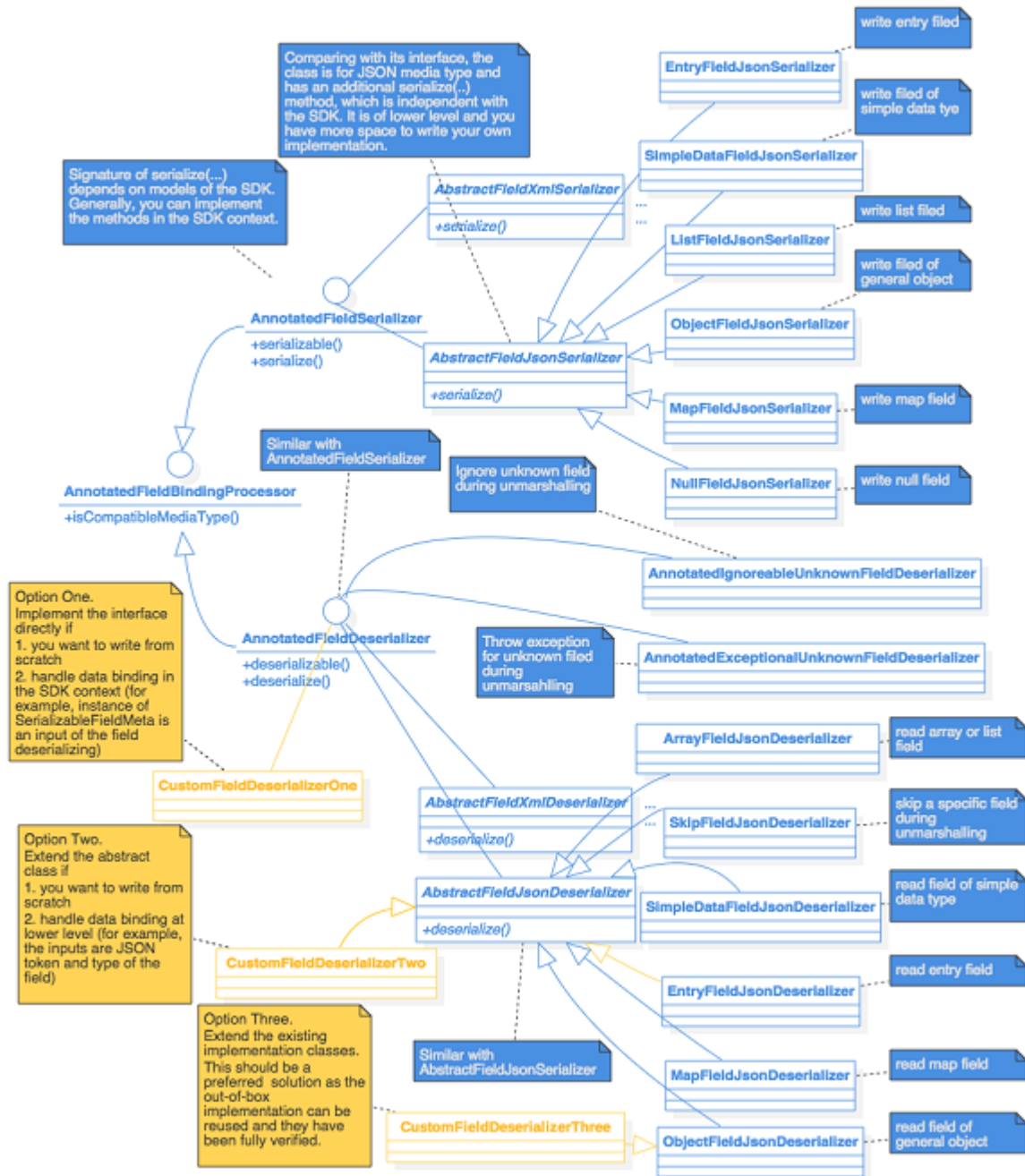
As you can see in the image below, the Documentum REST data binding framework is composed of many serializers and deserializers for different kinds of fields.

This architecture can be extended. Along with the `SerializableField#serializers` and `SerializableField#deserializers` annotations, you can also link new ones with specified fields.

Below is a class diagram for XML data binding classes:



Here's the class diagram for JSON



Generally, you can implement a new custom serializer or deserializer in two ways:

1. By extending abstract serializers or deserializers and implementing the abstract methods
2. By extending Documentum REST out-of-box serializers or deserializers

Building new serializers or deserializers from scratch is not easy work. We recommend that you extend existing serializers or deserializers of the Documentum Platform REST Services.

Samples are available in `<dtm-rest-sdk-root>/samples/custom-field-binding-samples`

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 dctm-rest-ext-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
```

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>7.3</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>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```
        <debug>false</debug>
        <failBuild>true</failBuild>
    </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.
- The `failBuild` element determines whether to terminate and fail the building process when the scanner detects errors.

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 plug-in configuration in the `pom.xml` file.

Developing Custom Resources

This section discusses how to develop simple custom resources with the Documentum REST Extensibility.

Typically, custom resource 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 Platform 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 Platform 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. For more information, see [Adding Links to Core Resources](#).

Setting up a Custom Resource Project

Documentum Platform 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
|---pom
```

The quick start is to use the Maven archetype project in Documentum Platform REST Services SDK to create such a project structure.

Programming for Custom Resources

This section introduces important features of custom resource programming. See [Tutorial: Documentum Platform REST Services Extensibility Development](#) for a comprehensive review of 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, we do not define any constructor for the `Server` class so that the default (no-argument) one provided by the compiler 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;
    }
}

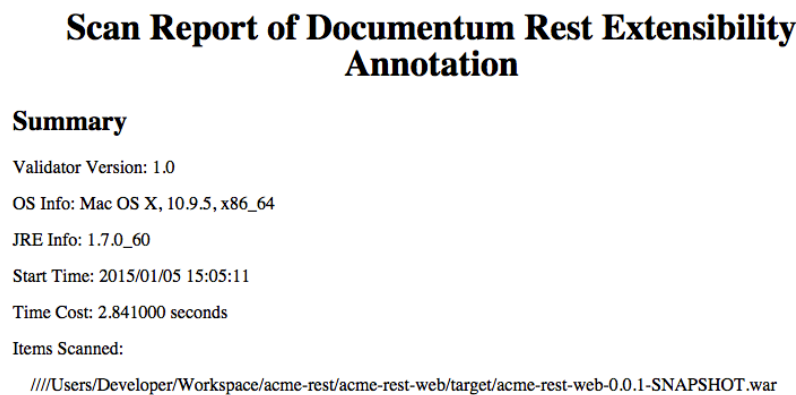
```

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 8. 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. Implement it as a Java bean.

There are two ways to create a Java bean:

- Spring Java code configuration
- Using an XML namespace

Example 7-58. Create a Java Bean Using Spring Code Configuration

```
@Configuration
@ComponentScan(
    basePackages = { "com.acme" },
    excludeFilters = {
        @ComponentScan.Filter(
            type = FilterType.CUSTOM,
            classes = {
                com.emc.documentum.rest.context.ComponentScanExcludeFilter.class
            }
        )
    }
)

public class CustomContextConfig {
    @Bean(name="customUserManager")
    public UserManager customUserManager() {
        return new UserManagerImpl();
    }
}
```

You must ensure that the package where you created your custom configuration class is specified in the `rest.context.config.location` property within the `rest-api-runtime.properties` file.

The `com.emc.documentum.rest.context.ComponentScanExcludeFilter` exclude filter is mandatory when you use the `@ComponentScan` Spring annotation in your custom defined configuration class. This exclude filter ensures that the Spring framework loads all of the resources that you have defined.

Example 7-59. Create a Java Bean Using Spring XML Namespace

Define the 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>
```

The Java beans can be referenced by resource controllers by using the `@Autowired` Spring annotation. Here is a code sample of the persistence API reference in a resource controller:

Example 7-60. 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 Platform 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: Getting Login User Context

The persistence API `com.emc.documentum.rest.config.RepositoryContextHolder` provides static methods to retrieve the login user's context information, such as the *login name*, *username*, *user privileges*, etc.

Some persistence API methods require additional DFC calls, and should be used as little as possible. Please refer to the JavaDoc API for more information.

Persistence: Session Management

Documentum Platform 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. Therefore, after a user logs in, the persistence API retrieves the right user session without the need to explicitly instantiate the DFC session manager. The `SessionAwareAbstractManager` persistence API can be used to extend the `com.emc.documentum.rest.dfc` interface to get the user session. Here is a code sample that shows you how to do this:

Example 7-61. Getting User Session by Extending the SessionAwareAbstractManager API

```

/**
 * 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 you are not extending the `SessionAwareAbstractManager` persistence API, it can use the auto wired `RepositorySession` instance to retrieve a session by using the `@Autowired` Spring annotation. Here is a code sample that shows you how to do this:

Example 7-62. Getting User Session without Extending the SessionAwareAbstractManager API

```

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);
    }
}
```

If you are using a version of the JDK that is Java 7 or later, you can use Core REST *IDfCloseableSession* API to manage the session. This API automatically handles the session release. Here is a code sample that shows you how to use this API:

Example 7-63. Session management usage in Java and later

```
public class CustomUserManager extends SessionAwareAbstractManager implements UserManager{
    @Override
    public UserObject get(String userName, AttributeView attributeView) throws DfException{
        IDfSession session = null;

        try (IDfCloseableSession session = dfcSessionRepository.getCloseableSession()) {
            // Use a method from super class to get the session for the login user
            IDfUser dfUser = session.getSession(false).getUser(userName);

            if(dfUser == null) {
                return null;
            }
            else {
                return convert(dfUser, attributeView);
            }
        }
    }
}
```

When the Documentum Platform REST Services security module is not used while performing unit testing of the persistence API, you can set the login username and password in your test code, which makes the session available to the test code. The code mocks a user login from a servlet request. Here is a code sample that shows you how to do this:

Example 7-64. Setting the Username and Password for Unit Testing

```
com.emc.documentum.rest.config.RepositoryContextHolder.setRepositoryName(...)
com.emc.documentum.rest.config.RepositoryContextHolder.setLoginName(...)
com.emc.documentum.rest.config.RepositoryContextHolder.setPassword(...)
```

When a transaction is required on the resource controller, the `com.emc.documentum.rest.dfc.ContextSessionManager` class can be used to commit the transaction with the persistence API as shown in the following code sample:

Example 7-65. Commit a Transaction with the Persistence API

```
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 SessionCallable<T>() {
            public T call(IDfSession session) throws Exception {
                return doRegularCopy(objectId, folderId);
            }
        });
    }
}

```

Note: Additional session management samples are available in the `documentum-rest-version.zip/samples/documentum-rest-java_api-samples/` SDK.

Persistence: Creating Feed Pages from DQL Result

For an implementation of a feed resource, the persistence API may execute a DQL query to get the object collection. Documentum Platform REST Services SDK library provides the `com.emc.documentum.rest.paging.Page<T>` class to retrieve the object collection as a feed page.

The `com.emc.documentum.rest.dfc.query.PagedQueryTemplate<T>` class is the basic DQL template that is used to produce a DQL query expression. A custom feed resource can extend the `PagedQueryTemplate` class to create a custom DQL query expression and call the `com.emc.documentum.rest.paging.PagedDataRetriever<T>` method to get a `Page<T>`. Here are some code samples that demonstrate how to do some typical tasks:

Example 7-66. Use the `PagedPersistentDataRetriever` to get a page for a custom `PagedQueryTemplate`

The Documentum Platform 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 a `com.emc.documentum.rest.model.PersistentObject` type or one of its sub types. Here is a code sample that shows you how to retrieve a page of document objects:

```

// 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();

```

Example 7-67. Use the `PagedDataRetriever` to get a page for a custom `PagedQueryTemplate` with custom models

You can create a custom object collection manager if the data model in the page does not extend `com.emc.documentum.rest.model.PersistentObject`. Here is a code sample that shows you how to do this:

```

PagedQueryTemplate pagedQueryTemplate = new CustomPagedQueryTemplate(getUsername());

PagedDataRetriever<MyDocument>
pagedDataRetriever = new PagedDataRetriever<MyDocument>(
    new MyDocumentCollectionManager(),

```

```
        pagedQueryTemplate, 1, 5, true, AttributeView.DEFAULT);
Page<MyDocument> page = pagedDataRetriever.get();
```

Example 7-68. 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 the `com.emc.documentum.rest.paging.Paginator` method to generate a page from the object collection. Here is a code sample that shows you how to do this:

```
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 detailed code samples in the `documentum-rest-version.zip/samples/documentum-rest-java_api-samples/` SDK.

Resource: Programming the Controller

The resource implementation uses 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 a code sample that shows you how to use the resource controller:

Example 7-69. Using The Spring 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 declares the `@Controller` Spring annotation, which identifies it as a resource controller. We recommend that you set a unique name for the `@Controller`. The name for the controller is used as the code name of this resource, which is used by configurations and log entries that reference this resource.

The controller class defines its request mapping with the `@RequestMapping` Spring annotation. There are some additional mapping rules, however they do not apply to the URI on this annotation. For more information, refer to the Spring framework documentation.

In the controller definition, the `@ResourceViewBinding` Core REST annotation defines the default view 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, the `@RequestMapping` annotation can be declared and used to map the HTTP method and media types for a resource operation.

The controller method defines the `@ResponseBody` Spring annotation, which indicates that the returning object format can be directly sent to the marshalling framework.

The controller method defines the `@ResponseStatus` Spring controller annotation, which specifies the HTTP status of this resource operation. In the event that there is an exception thrown by this method, error mapping takes place and the status of the method is reset to the corresponding error status.

For more information, see [Error Handling and Representations](#).

The resource method can call its controller super class method `getRenderedObject` or `getRenderedPage` to invoke dynamic view definitions that are used to render the model instance and produce the links or the other representation customizations.

Note: There are more resource controller samples in the `documentum-rest-version.zip` `/samples/documentum-rest-resource-samples/` SDK.

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 Platform REST Services library provides two view implementations to implement part of the methods. Custom views can extend these classes and 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 `documentum-rest-version.zip` `/samples/documentum-rest-resource-samples/` SDK.

Resource: Binding the View and the Controller

As previously mentioned, a resource can be bound to a view implementation by declaring the `@ResourceViewBinding` Core REST annotation in the resource controller. A `@ResourceViewBinding` can be applied to a controller at the class level or at the method level.

The annotation has an attribute that you can use to specify which view definitions the resource controller (or method) uses to render the resource model in the controller method. You can bind more than one view definition to a controller or a method (repeating value). However, each view definition must correspond to a unique resource model type, such as feed, document, folder, etc.

Here are some things to consider when deciding whether to apply the annotation at the class level or at the method level.

- When the annotation is applied at the class level, all methods in the controller by default uses the view definitions in the class level annotation.
- When the annotation is applied at 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 a code sample that shows you how to use the `@ResourceViewBinding` annotation at 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.

Example 7-70. Using the `@ResourceViewBinding` Annotation

```
@Controller("acme#relations")
@RequestMapping("/repositories/{repositoryName}/relations-x")
@ResourceViewBinding({ RelationsFeedView.class })

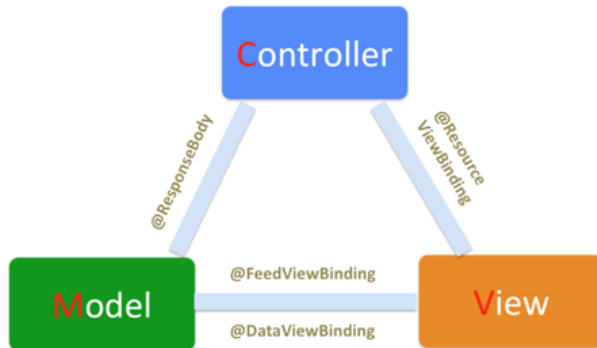
public class RelationsController extends AbstractController{
    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public AtomFeed get(
        @RequestParam(value = "name", required = false) final String relationName,
        @RequestUri final UriInfo 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` annotation can bind to multiple View definitions. When multiple View definitions are registered for a `@ResourceViewBinding`, each View definition must correspond to a unique data model. There can be at most one `FeedableView` definition in the `@ResourceViewBinding` definition because `FeedableView` corresponds to the `AtomFeed` data model.
- When `@ResourceViewBinding` annotation is used at the method level, the `@RequestUri final UriInfo uriInfo` declaration is required in the method definition.

Resource controllers, models, and views can be bound with one another by using the Java annotations shown in the following diagram:

Figure 9. Java Annotations and Documentum MVC



Resource: Customizing Resource Views

Customization involves the following tasks:

1. Write custom view class
2. Register the class to a resource

Customize Resource View

REST extensibility allows you to write your own view implementation classes for a specific data model.

Hide Link Relations

In a custom view, you can call the `removeLink(...)` method to hide link relations. You can manually extend from the Core View implementation when the modification that you want to make is small.

The only method that must be overridden is the `customize()` method.

Here's a code sample showing a View implementation that extends a Core repository View. In this View class, the link relation `relation-types` is removed from the repository resource:

Example 7-71. A View that Extends a Core Repository

```
/**
 * Customize repository with extension to remove relation-types link relation
 */
public class RepositoryViewExtend extends RepositoryView {
    public RepositoryViewExtend(Repository repository,
        UriInfo uriInfo,
        String repositoryName,
```

```

        Boolean returnLinks,
        Map<String, Object> others) {
            super(repository, uriInfo, repositoryName, returnLinks, others);
        }

        @Override
        public void customize() {
            // Remove relation-types link relation from the repository
            removeLink(CoreLinkRelation.RELATION_TYPES.rel(), null);
        }
    }
}

```

Edit Link Relations

In the custom view, you can call the `clearLinks()`, `makeLink(..)`, or `makeLinkIf(..)` methods to build arbitrary link relations to a resource model. When a significant change is required, you can write a completely new view implementation class for a resource model.

Here are some important points:

- A new view implementation class must extend from `EntryableView`, or `LinkableView` where all abstract methods must be implemented.
- A new view implementation class must have the `@DataViewBinding` annotation added to the class definition.

In the following code sample of the View implementation for the *Repository* resource, the links on the *Repository* resource are rebuilt in the View class. Take notice of the following items:

- The link relation *current-user* and *cabinets* are static. This means they are available in any repository resource.
- The link relation *users* is dynamic. This means that only the `admin` or `super` user can see it.

Here's a code sample that illustrates the preceding points:

Example 7-72. Rebuilding Links in the View Class

```

/**
 * Customize the repository with totally new view definitions that specify the
 * exact link relations
 */
@DataViewBinding(modelType = Repository.class)
public class RepositoryViewNew extends EntryableView<Repository> {
    public RepositoryViewNew(Repository repository, UriInfo uriInfo, String repositoryName,
        Boolean returnLinks, Map<String, Object> others) {
        super(repository, uriInfo, repositoryName, returnLinks, others);
    }

    @Override
    public String entryTitle() {
        return getDataInternal().getName();
    }

    @Override
    public String entrySummary() {
        return getDataInternal().getDescription();
    }

    @Override

```

```
public Date entryUpdated() {
    return new Date();
}

@Override
public Date entryPublished() {
    return new Date();
}

@Override
public List<AtomAuthor> entryAuthors() {
    return Collections.emptyList();
}

@Override
public List<Link> entryLinks() {
    return getDefaultEntryLinks();
}

@Override
public void customize() {
    customizeLinks();
}

@Override
public String canonicalResourceUri(boolean validate) {
    return getUriFactory(validate).repositoryUri(getDataInternal().getName(), null);
}

@Override
protected Map<String, Object> resolveUriTemplateVariables(Map<String,
    String> valueMapping) {
    return Collections.emptyMap();
}

private void customizeLinks() {
    // Clear all links
    clearLinks();
    // Build users link relation on condition when the login user is admin or
    // super user
    makeLinkIf(RepositoryContextHolder.getUserPrivileges() > 8,
        CoreLinkRelation.USERS.rel(), getUriFactory().usersUri(null));
    // Build current-user link relation without condition
    makeLink(CoreLinkRelation.CURRENT_USER.rel(),
        getUriFactory().currentUsersUri(null));
    // Build cabinets link relation without condition
    makeLink(CoreLinkRelation.CABINETS.rel(), getUriFactory().cabinetsUri(null));
}
}
```

Hide Properties

When using a custom view, you can also customize what object properties to return the client.

Here's a code sample that shows a view implementation on the cabinet resource model where all internal properties are hidden.

```
/**
 * Customize cabinet resource to remove internal attributes.
 */
```



```

public class CabinetViewExtend extends CabinetView {
    public CabinetViewExtend(CabinetObject cabinet, UriInfo uriInfo,
        String repositoryName, Boolean returnLinks,
        Map<String, Object> others) {
        super(cabinet, uriInfo, repositoryName, returnLinks, others);
    }

    @Override
    public void customize() {
        // Remove all internal attributes with prefix "i_"
        for (Attribute attr : serializableData.getAttributes()) {
            if (attr.getName().startsWith("i_")) {
                serializableData.removeAttributeIfExists(attr.getName());
            }
        }
    }
}

```

Customize atom Attributes

A custom view also allows you to customize the atom feed and entry attributes on a collection or on a single data object resource.

In the following example of the View implementation on the *Cabinets* feed resource, link relations and feed title are modified.

Example 7-73. Customize the Cabinet Resource to Add New Link Relations

```

/**
 * Customize cabinet resource to add new links and change feed title.
 */
public class CabinetsFeedViewExtend extends CabinetsFeedView {
    public CabinetsFeedViewExtend(Page<CabinetObject> cabinets, UriInfo uriInfo,
        String repositoryName, Boolean returnLinks,
        Map<String, Object> others) {
        super(cabinets, uriInfo, repositoryName, returnLinks, others);
    }

    @Override
    public String feedTitle() {
        // Customize feed title
        return "Cabinets in organization: " + getRepositoryName();
    }

    @Override
    public List<Link> feedLinks() {
        List<Link> links = super.feedLinks();
        // Add about link relation to feed links
        links.add(new Link("about", getUriFactory().productInfoUri()));
        return links;
    }
}

```

The view implementation classes must be built and packaged in a jar file where the final REST war file can access it and load the classes into the class path.

Register Custom Resource View

The next and last step is to register the custom views to resources. The new view implementations overrides the default views on resources. The configuration is in the same YAML file.

```
---
# RESOURCE VIEW REGISTRY #
#####
resource-view-registry:
# - resource: repository
#   view:      [org.acme.view.impl.custom.RepositoryViewExtend]
```

Here's a code sample that shows you how to register previous view for resources.

```
---
# RESOURCE VIEW REGISTRY #
#####
resource-view-registry:
- resource: repository
  view:      [org.acme.view.impl.custom.RepositoryViewExtend]
- resource: cabinet
  view:      [org.acme.view.impl.custom.CabinetViewExtend]
- resource: cabinets
  view:      [org.acme.view.impl.custom.CabinetsFeedViewExtend]
```

Using YAML Config

When multiple views are registered for a resource, each view definition should correspond to a different data model. There could at most be one custom `FeedableView` definition for a resource.

Here are some possible error codes or messages for invalid view registry values:

```
E_UNKNOWN_RESOURCE_NAME=Unknown resource name is found: {0}.
E_RESOURCE_VIEW_DISALLOWED=Preserved core resources {0} are not allowed for view
  customization. Please review your input resources: {1}
E_RESOURCE_VIEW_CLASS_NOT_FOUND=The view definition class is not found in class-path: {0}
```

View Loading Precedence

The precedence of the view definition used by a controller method is: YAML defined view > method level `@ResourceViewBinding` annotation > class level `@ResourceViewBinding` annotation.

View Precedence

- When a custom `FeedableView` is defined on the feed resource, the feed resource uses the custom `FeedableView` to render the atom feed. Otherwise, the feed resource uses the default `FeedableView` definition on the annotation `@ResourceViewBinding` of the feed resource
- When the feed is inline, the feed resource looks up the `EntryableView` definition on the annotation `@FeedViewBinding` of the `FeedableView` used by this feed resource

Troubleshooting

After enabling `DEBUG level log4j` logging for the `emc.emc.documentum.rest` package, the following messages, which have been formatted to fit this document, are printed in the log file:

```
+++++++ RESOURCE CUSTOM VIEW PRINT START ++++++
+ -- NAME --                -- VIEW -- +
```

```

+ -- cabinet                -- org.acme.view.impl.custom.CabinetViewExtend|
+ -- cabinets               -- org.acme.view.impl.custom.CabinetsFeedViewExtend|org
                           .acme.view.impl.custom.CabinetViewExtend|
+ -- repository             -- org.acme.view.impl.custom.RepositoryViewNew|
+ -- NAME --                -- VIEW -- +
+++++++ RESOURCE CUSTOM VIEW PRINT END ++++++

+++++++ RESOURCE DEFAULT VIEW PRINT START ++++++
+ -- NAME --                -- LEVEL --                -- VIEW -- +
+ -- acme#alias-set         -- CLASS                -- org.acme.view.impl
                           .AliasSetView|
+ -- acme#alias-sets        -- CLASS                -- org.acme.view.impl
                           .AliasSetsFeedView|
+ -- acme#alias-sets        -- METHOD:getAliasSets    -- org.acme.view.impl
                           .AliasSetsFeedView|
+ -- acme#alias-sets        -- METHOD:createAliasSet  -- org.acme.view.impl
                           .AliasSetView2|
+ -- acme#user              -- CLASS                -- com.emc.documentum.rest
                           .view.impl.UserView|
+ -- acme#users             -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .UsersFeedView|
+ -- batch-capabilities     -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .BatchCapabilitiesView|
+ -- cabinet               -- CLASS                -- com.emc.documentum.rest
                           .view.impl.CabinetView|
+ -- cabinets              -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .CabinetsFeedView|com
                           .emc.documentum.rest
                           .view.impl.CabinetView|
+ -- cabinets              -- METHOD:createCabinet  -- com.emc.documentum.rest
                           .view.impl.CabinetView|
+ -- checked-out-objects    -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .CheckedOutObjectsFeedView|
+ -- child-folder-link      -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .FolderLinkView|
+ -- child-folder-links     -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .FolderLinksFeedView|com
                           .emc.documentum.rest
                           .view.impl
                           .FolderLinkView|
+ -- child-folder-links     -- METHOD:link          -- com.emc.documentum.rest
                           .view.impl
                           .FolderLinkView|
+ -- content               -- CLASS                -- com.emc.documentum.rest
                           .view.impl.ContentView|
+ -- contents              -- CLASS                -- com.emc.documentum.rest
                           .view.impl
                           .ContentsFeedView|com
                           .emc.documentum.rest
                           .view.impl.ContentView|
+ -- contents              -- METHOD:createContent  -- com.emc.documentum.rest
                           .view.impl.ContentView|
+ -- current-user          -- CLASS                -- com.emc.documentum.rest
                           .view.impl.UserView|
+ -- current-user2         -- CLASS                -- com.emc.documentum.rest
                           .view.impl.UserView|
+ -- current-version        -- CLASS                -- com.emc.documentum.rest
                           .view.impl.SysObjectView|
+ -- default-folder        -- CLASS                -- com.emc.documentum.rest

```

```
+ -- document          -- CLASS          .view.impl.FolderView|
-- com.emc.documentum.rest
+ -- dql-query          -- CLASS          .view.impl.DocumentView|
-- com.emc.documentum.rest
-- view.impl
-- QueryResultFeedView|com
-- emc.documentum.rest
-- view.impl
-- QueryResultItemView|
-- com.emc.documentum.rest
+ -- folder            -- CLASS          .view.impl.FolderView|
-- com.emc.documentum.rest
+ -- folder-child-documents -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- DocumentsFeedView|com
-- emc.documentum.rest
-- view.impl.DocumentView|
+ -- folder-child-documents -- METHOD:createChildDocument -- com.emc.documentum.rest
-- view.impl.DocumentView|
+ -- folder-child-folders -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- FoldersFeedView|com
-- emc.documentum.rest
-- view.impl.FolderView|
+ -- folder-child-folders -- METHOD:createChildFolder -- com.emc.documentum.rest
-- view.impl.FolderView|
+ -- folder-child-objects -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- SysObjectsFeedView|com
-- emc.documentum.rest
-- view.impl
-- SysObjectView|com.emc
-- documentum.rest.view
-- impl
-- ContentfulObjectView|
+ -- folder-child-objects -- METHOD:postSysObject -- com.emc.documentum.rest
-- view.impl
-- SysObjectView|com.emc
-- documentum.rest.view
-- impl
-- ContentfulObjectView|
+ -- group             -- CLASS          -- com.emc.documentum.rest
-- view.impl.GroupView|
+ -- group-member-groups -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- GroupsFeedView|
+ -- group-member-users  -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- UsersFeedView|
+ -- groups             -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- GroupsFeedView|
+ -- lock              -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- SysObjectView|
+ -- network-location    -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- NetworkLocationView|
+ -- network-locations   -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- NetworkLocationsFeedView|
+ -- object             -- CLASS          -- com.emc.documentum.rest
-- view.impl
-- SysObjectView|com.emc
-- documentum.rest.view
-- impl
```

```

+ -- parent-folder-link      -- CLASS      .ContentfulObjectView|
                                -- com.emc.documentum.rest
                                .view.impl
                                .FolderLinkView|
+ -- parent-folder-links    -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .FolderLinksFeedView|com
                                .emc.documentum.rest
                                .view.impl
                                .FolderLinkView|
+ -- parent-folder-links    -- METHOD:link  -- com.emc.documentum.rest
                                .view.impl
                                .FolderLinkView|
+ -- product-information    -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .ProductInfoView|
+ -- relation               -- CLASS      -- com.emc.documentum.rest
                                .view.impl.RelationView|
+ -- relation-type          -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .RelationTypeView|
+ -- relation-types        -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .RelationTypesFeedView|
+ -- relations              -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .RelationsFeedView|com
                                .emc.documentum.rest
                                .view.impl.RelationView|
+ -- relations              -- METHOD:createRelation -- com.emc.documentum.rest
                                .view.impl.RelationView|
+ -- repositories           -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .RepositoriesFeedView|
+ -- repository             -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .RepositoryView|
+ -- search                 -- CLASS      -- com.emc.documentum.rest
                                .search.representation
                                .view.impl
                                .SearchFeedView|
+ -- type                   -- CLASS      -- com.emc.documentum.rest
                                .view.impl.TypeView|
+ -- types                  -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .TypesFeedView|
+ -- user                   -- CLASS      -- com.emc.documentum.rest
                                .view.impl.UserView|
+ -- users                  -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .UsersFeedView|
+ -- versions               -- CLASS      -- com.emc.documentum.rest
                                .view.impl
                                .VersionsFeedView|com
                                .emc.documentum.rest
                                .view.impl
                                .SysObjectView|com.emc
                                .documentum.rest.view
                                .impl.DocumentView|com
                                .emc.documentum.rest
                                .view.impl
                                .ContentfulObjectView|
+ -- versions               -- METHOD:checkIn -- com.emc.documentum.rest
                                .view.impl
                                .SysObjectView|com.emc

```

```
.documentum.rest.view
.impl.DocumentView|com
.emc.documentum.rest
.view.impl
.ContentfulObjectView|
+ -- versions          -- METHOD:checkInContent    -- com.emc.documentum.rest
.view.impl
.SysObjectView|com.emc
.documentum.rest.view
.impl.DocumentView|com
.emc.documentum.rest
.view.impl
.ContentfulObjectView|
+ -- versions          -- METHOD:checkInMetadata    -- com.emc.documentum.rest
.view.impl
.SysObjectView|com.emc
.documentum.rest.view
.impl.DocumentView|com
.emc.documentum.rest
.view.impl
.ContentfulObjectView|
+ -- NAME --          -- LEVEL --          -- VIEW -- +
+++++++ RESOURCE DEFAULT VIEW PRINT END ++++++

+++++++ QUERYABLE VIEW PRINT START ++++++
+ -- TYPE --          -- VIEW -- +
+ -- dm_alias_set     -- com.emc.documentum.rest.view.impl.UserView|
+ -- dm_cabinet       -- com.emc.documentum.rest.view.impl.CabinetView|
+ -- dm_document      -- com.emc.documentum.rest.view.impl.DocumentView|
+ -- dm_folder        -- com.emc.documentum.rest.view.impl.FolderView|
+ -- dm_group         -- com.emc.documentum.rest.view.impl.GroupView|
+ -- dm_network_location_map -- com.emc.documentum.rest.view.impl.NetworkLocationView|
+ -- dm_relation      -- com.emc.documentum.rest.view.impl.RelationView|
+ -- dm_relation_type -- com.emc.documentum.rest.view.impl.RelationTypeView|
+ -- dm_sysobject     -- com.emc.documentum.rest.view.impl.SysObjectView|
+ -- dm_type          -- com.emc.documentum.rest.view.impl.TypeView|
+ -- dm_user          -- com.emc.documentum.rest.view.impl.UserView|
+ -- dmi_dd_type_information -- com.emc.documentum.rest.view.impl.TypeView|
+ -- dmr_content      -- com.emc.documentum.rest.view.impl.ContentView|
+ -- TYPE --          -- VIEW -- +
+++++++ QUERYABLE VIEW PRINT END ++++++
```

Resource: Building Resource links

The abstract view classes `EntryableView`, `LinkableView`, and `FeedableView` provide 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(),
        ResourceUriBuilder
            .onResource("user")
            .pathVariables((String) serializableData.getAttributeByName("owner_name"))
    )
}
```

EntryableView

The `EntryableView` class extends `LinkableView` with additional atom entry customizations. Besides the link methods that the `LinkableView` class provides, you can use the `entryLinks()` method to define your own entry links in the atom entry representation. When your custom resource view class extends the `PersistentLinkableView` class, a default link relation called `edit` is added to the entry link collection.

FeedableView

By default, the `FeedableView` class 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 `feedLinks()` method to modify these link relations or add other link relations.

In the following code sample, we override the `feedLinks()` method to add an additional link relation called `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", ResourceUriBuilder.onResource("product-info")));
    return links;
}
```

Resource: Building Resource URIs with ResourceUriBuilder

In release 7.2, users had to call `com.emc.documentum.rest.http.UriFactory` to build links for core resources or custom resources. For links of custom resources, you must call `com.emc.documentum.rest.http.UriFactory#buildUriByTemplateName()` to build links from YAML URI template information.

In 7.3 release, a new API `com.emc.documentum.rest.context.ResourceUriBuilder` was introduced to build resource links in a consistent way. The detailed API description can be found in the SDK JavaDoc.

Here's a code sample of the resource URI builder:

```
// Build user resource URI: <base uri context>/repositories/  
// <repoName>/users/dmadmin  
String href = ResourceUriBuilder  
    .onResource("user")  
    .pathVariable(true, "dmadmin")  
    .build();  
  
// Build query resource URI as hreftemplate: <base uri context>/  
// repositories/<repoName>{?q}  
String href = ResourceUriBuilder  
    .onResource("query")  
    .asTemplate("q")  
    .build();  
  
// Build cabinets resource URI with query: <base uri context>/  
// repositories/<repoName>/cabinets?inline=true&view=:all  
String href = ResourceUriBuilder  
    .onResource("cabinets")  
    .queryParam("view", ":all")  
    .queryParam("inline", "true")  
    .build();  
  
// Build custom alias-set resource URI by resource name: <base uri context>/  
// repositories/<repoName>/alias-set/1234?view=:all  
String href = ResourceUriBuilder  
    .onResource("acme#alias-set")  
    .pathVariables("1234")  
    .queryParam("view", ":all")  
    .build();  
  
// Build custom alias-set resource URI by resource controller: <base uri context>/  
// repositories/<repoName>/alias-set/1234?view=:all  
String href = ResourceUriBuilder  
    .onResource(org.acme.rest.AliasSetController.class)  
    .pathVariables("1234")  
    .queryParam("view", ":all")  
    .build();  
  
// Build custom alias-set resource URI by custom URI template: <base uri context>/  
// repositories/<repoName>/alias-set/1234?view=:all  
String href = ResourceUriBuilder  
    .onTemplate("X_ALIAS_SET_URI_TEMPLATE")  
    .pathVariables("1234")  
    .queryParam("view", ":all")  
    .build();
```

At runtime, the `<base uri context>` and the format extension are resolved automatically. In unit testing, you can specify the base URI context and format extension to verify the full path. Here's a code sample that shows you how to do that:

```
UriInfo uriInfo = new UriInfo();  
uriInfo.setBaseUri("http://localhost:8080/dctm-rest");  
uriInfo.setFormatExtension("");  
String href = ResourceUriBuilder.onResource("user")  
    .repository("ACME")  
    .uriInfo(uriInfo)
```



```

        .pathVariable(true, "dmadmin")
        .build();
assertEquals("http://localhost:8080/dctm-rest/repositories/ACME/users/
dmadmin", href);

```

When implementing a resource view, you can use the newly added methods on the abstract views to get the URI resource builder. Here's a listing of those methods:

- `com.emc.documentum.rest.view.LinkableView#uriBuilder(string)`
- `com.emc.documentum.rest.view.LinkableView#idUriBuilder(string, string)`
- `com.emc.documentum.rest.view.LinkableView#nameUriBuilder(string, string)`
- `com.emc.documentum.rest.view.FeedableView#uriBuilder(string)`

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 Platform 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 must 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 must 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);
}
}

```

Using a Generic AtomEntry: The AtomFeed can have a generic AtomEntry entry type defined. Here's a code sample that illustrates how to do this:

```

@SerializableType(value = "feed", ...)
public class AtomFeed<T> extends AtomEntry<> {
    private String id;
    private String title;
    ...
    private List<T> entries;

    // To make this compatible with previous versions,
    // the getEntries() and setEntries() methods still use an
    // explicit List<AtomEntry>
    public List<AtomEntry> getEntries() {}
    public void setEntries(List<AtomEntry> entries) {}

    // For better leveraging of the generic T type,
    // two other similar functions are added, which use a T parameter
    public List<T> getGenericEntries() {}
    public void setGenericEntries(List<T> entries) {}
}

```

When you extend from AtomFeed, the new entry type must be provided, in addition to adding new fields.

```

@SerializableType(value = "feed", ...)
public class CustomizedAtomFeed extends AtomFeed<CustomizedAtomEntry> {
    private String customFeedField;
}

```

Now, CustomizedAtomFeed and CustomizedAtomEntry can be used to serialize and deserialize at both the server and client side.

A small compatibility issue exists in the 7.2 code. Code that is currently working with the 7.2 version of AtomFeed, AtomEntry, or both, may code as shown here:

```

AtomFeed feed = ...
// Iterate the entries
for(AtomEntry entry : feed.getEntries()) { //compile error
    ...
}

```

```

// Or get the entry by index directly
feed.getEntries().get(0).getContent(); //compile error

```

Due to a limitation of Java generics, the new version of AtomFeed in version 7.3 code has compilation issues. Since AtomFeed is now defined as a Java generic class, using it as a normal class causes its generic information to disappear. The return type of feed.getEntries() becomes List<Object> instead of List<AtomEntry> as expected.

To remedy this issue, you can use generic information with AtomFeed or you can explicitly cast the return type of feed.getEntries() to List<AtomEntry>. Here's a code sample that illustrates this:

```
AtomFeed<AtomEntry> feed = ...

// Iterate the entries
for (AtomEntry entry : (List<AtomEntry>) feed.getEntries()) {
    ...
}

// Or get the entry directly by index
((List<AtomEntry>) feed.getEntries()).get(0).getContent
```

Resource: Error Handling and Representation

Documentum Platform REST Services leverages the Spring framework to resolve the error mapping from Java exceptions to error messages. The Documentum Platform 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 override the following bean with the custom class, which is in the `BaseMarshallingConfig` class of the `rest-api-mvc-resource` jar file.

```
@Bean(name = "generalExceptionMapping")
public GeneralExceptionMapping generalExceptionMapping() {
    return new 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 GenericExceptionHandler(e,
        HttpStatus.BAD_REQUEST, "E_INPUT_ILLEGAL_ARGUMENTS"));
}
```

Resource: Consuming Multipart Contents in Custom Resource Controller

You can define an `Iterator<Part>` controller parameter for the Request body of custom resources and annotate your definition with the `@RequestBody` annotation. Doing this gives you access to all parts of the Request. Here is a code sample that shows you how to read the multipart input, and write back to the Response.

Example 7-74. Read Multipart Input and Write to Response

```
@RequestMapping(method = RequestMethod.POST,
    consumes = { "multipart/form-data", "multipart/mixed" })
@ResponseStatus(HttpStatus.OK)
public void processMultipart(@RequestBody final Iterator<Part> parts)
```

```

throws DfException, MissingServletRequestParameterException, IOException {
if(parts != null) {
    response.setContentType(MediaType.TEXT_PLAIN.toString());
    OutputStream out = response.getOutputStream();
    int p = 1;
    while(parts.hasNext()) {
        Part part = parts.next();
        for(String header : part.getHeaderNames()) {
            //write the part header to the response directly
            out.write((header + "=" + part.getHeader(header) + "\r\n" ).getBytes());
        }
        //write the part content to the response
        IOUtils.copy(part.getInputStream(), out);
        out.write("\r\n-----end-----\r\n".getBytes());
        out.flush();
    }
}
}

```

Each part of the multipart contents contains headers and the content stream. These parts can be read one by one using standard Iterator methods such as *hasNext* and *next*. All parts of the multipart contents must be read sequentially because they are all in one multipart stream, and the REST server does not cache any part of the stream.

Linking Custom and Core Resources

To make the custom REST services hypermedia driven, there must establish link relations between Core resources and custom resources. Documentum Platform 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 Platform 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.

Working with YAML Configuration

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 configurations for users to customize Core REST resources. You can also customize the YAML location within `dctm-rest.war/WEB-INF/classes/rest-api-runtime.properties`, as follows:

Example 7-75. Setting the Package for a Custom YAML File

```
# Sets the package for custom yaml file.
rest.ext.yaml.package=
```

You are free to rename the YAML file, but the file extension must remain `.yaml`, for example `my-custom-rest.yaml`. The `rest-api-runtime.properties.template` file, which is located in the same location, has more details on the configuration.

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 Platform 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.

In the REST WAR file, the `rest-api-custom-resource-registry.yaml` YAML file provides a configuration entry called `uri-template-registry` that can be used to register new URI templates.

Here's a sample showing the syntax for a URI template definition:

Example 7-76. A URI Template Definition

```
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 cause errors. Tab characters (`\t`) are never allowed for indentation in YAML.

Alternatively, when 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 that are added to existing resources. A URI template consists of the following elements as key-value pairs:

Table 12. URI Template Elements

Key	Value
name *	<p>Name of a custom URI template.</p> <p>The name must start with the prefix <code>X_</code>.</p> <p>We recommend that you use uppercase words delimited by underscore (<code>_</code>) to name a URI template.</p> <p>Example: <code>X_ALIAS_SET_RESOURCE_TEMPLATE</code></p>
[href hreftemplate] *	<p>URI template pattern for a link. The key is either <code>href</code> or <code>hreftemplate</code>. Only one <code>href</code> or <code>hreftemplate</code> exists in a URI template.</p> <p>The URI template pattern can contain variables or query parameters enclosed in curly brackets <code>{ }</code>. For example:</p> <pre>{repositoryUri}/<item1>/<item2>/<...>?<param1>&<param2>&<...>{ext}</pre> <p>For an <code>href</code> link, all variables in the URI pattern must be resolved on the server side, meaning that you must modify value-mapping correspondingly in the <code>ADD LINKS ON EXISTING RESOURCES</code> section for non-predefined variables.</p> <p>By contrast, an <code>hreftemplate</code> link must contain variables that are treated as placeholders, such as <code>{?foo, bar}</code>, which does not need resolving on the server side.</p> <p>Examples of hreftemplates</p> <ul style="list-style-type: none"> • <code>hreftemplate</code> with no fixed query parameters: <pre>/repositories/{repositoryName}/search?{?q,locations}</pre> • <code>hreftemplate</code> with both fixed query parameters and placeholders: <pre>/repositories/{repositoryName}/search?page=10&items-per-page={numberOfItems}&{&q,locations}</pre> <p>Predefined variables are resolved without additional settings in value-mapping. They are:</p> <ul style="list-style-type: none"> • <code>{baseUri}</code>: URI root of the current deployment. <p>Example: <code>{baseUri}</code> may refer to <code>https://current-deploy-host:8443/dctm-rest</code></p>

Key	Value
	<ul style="list-style-type: none"> <code>{repositoryUri}</code>: URI root of the current repository. <p>Example: <code>{repositoryUri}</code> may refer to <code>https://current-deploy-host:8443/dctm-rest/repositories/acme</code></p> <ul style="list-style-type: none"> <code>{ext}</code>: URI extension for the representation format, such as <code>.XML</code> or <code>.JSON</code>.
encoding	<p>Encoding method for path variables. Valid values are:</p> <ul style="list-style-type: none"> 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 (<code>/</code>).
external	<p>Specifies whether this URI template is external or internal. Valid values are:</p> <ul style="list-style-type: none"> 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 predefined variables <code>{repositoryUri}</code> or <code>{ext}</code>. 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 predefined variables <code>{baseUri}</code>, <code>{repositoryUri}</code>. <p>The default value is false.</p>
* Required	

The URI template name must be unique so that other resources used to build link relations can reference the template. In the code, the URI template can also be used to create an actual URL using the `buildUriByTemplateName` method of `com.emc.documentum.rest.http.UriFactory`. For example:

```
String moduleUri = uriFactory.buildUriByTemplateName(
    "X_MODULE_RESOURCE_TEMPLATE",
    Collections.singletonMap("moduleId", getDataInternal().getId()));
```

Normalization of URI Templates

Starting from REST Services 7.3, the system checks the format of an internal URI template and automatically appends the predefined variables `{baseUri}`, `{repositoryUri}`, and `{ext}` when necessary.

With this feature, the following URI templates are the same:

- `/help` vs. `/help{ext}` vs. `{baseUri}/help` vs. `{baseUri}/help{ext}`
- `/repositories/{repositoryName}/users?name={userId}` vs. `{repositoryUri}/users{ext}?name={userId}` vs. `{baseUri}/repositories/{repositoryName}/users?name={userId}`

Normalization of Custom URI Templates

While it is not necessary that you reference resources in YAML by defining custom URI templates, the custom URI template configuration in YAML is still useful when you want to:

- Build external link relations
- Build internal link relations with customizations. For example, when adding fixed query parameters, template parameters, etc.

From Documentum Platform REST Services version 7.3 and later, a complete definition of which customizations can be applied to a custom URI template have been added. Here's a code sample that shows you the URI template registry

```
uri-template-registry:
- name:      X_HELP_TEMPLATE
  href:      '{baseUri}/help{ext}'
- name:      X_POLICIES_FEED_TEMPLATE
  href:      '{repositoryUri}/objects/{objectId}/policies{ext}'
- name:      X_POLICY_TEMPLATE
  href:      '{repositoryUri}/objects/{objectId}/policies/{policyId}{ext}'
- name:      X_AUTHOR1_TEMPLATE
  href:      '{repositoryUri}/users/{userId}{ext}'
  encoding:  safe-text-encoding
- name:      X_AUTHOR2_TEMPLATE
  href:      '{repositoryUri}/users{ext}?name={userId}'
- name:      X_CUSTOM_SEARCH_TEMPLATE
  hreftemplate: '{baseUri}/x-search{?q,locations,page,items-per-page}'
- name:      X_CUSTOM_LOCATION_SEARCH_TEMPLATE
  hreftemplate: '{repositoryUri}/x-search?locations={path}{&q,page,items-per-page}'
- name:      X_EXTERNAL_SEARCH_TEMPLATE
  href:      'http://www.google.com?q={keyword}'
  external:  true
```

Normalization of Internal URI Templates

Starting from REST Services 7.2 two predefined variables *{baseUri}*, *{repositoryUri}*, and the suffix variable *{ext}* are applicable to internal URIs. When a URI template is internal (default to internal), the runtime automatically checks the URI format and appends the prefix or suffix when necessary.

With this feature, the following URI templates are the same:

- *{baseUri}/help{ext}* vs. */help* vs. *{baseUri}/help* vs. */help{ext}*
- *{repositoryUri}/users{ext}?name={userId}* vs. */repositories*
/[{repositoryName}]/users?name={userId} vs. *{baseUri}/repositories*
/[{repositoryName}]/users?name={userId}

Formats for the href Template

When a URI template is defined as an href template, there are href template parameters appended to the link. Documentum Platform REST Services supports a sub set of RFC6570.

- Appending template parameters `{?a,b,c,...}` onto a URI where there are no fixed query parameters
For example: `/repositories/{repositoryName}/search?{?q, locations,...}`
- Appending template parameters `{&a,b,c,...}` onto a URI where there are fixed query parameters
For example: `/repositories/{repositoryName}/search?page=10&items-per-page={numberOfItems}{&q, locations,...}`

Appending User Defined Variables onto the URI Template

User defined variables can be applied to internal and external URI templates, both href or href template URI templates. The variables must be resolved in custom Java code or by using variable value mapping in YAML.

Here's a code sample that shows you how to do that:

Example 7-77. Custom URI Templates With User Defined Variables

```
uri-template-registry:
- name:      X_AUTHOR_TEMPLATE
  href:      '{repositoryUri}/users/{userId}{ext}'
  encoding:  safe-text-encoding
- name:      X_MARKETING_GROUPS_TEMPLATE
  href:      '{repositoryUri}/groups{ext}?owner={userId}&subject={subject}&inline=true&items-per-page=50'
- name:      X_CUSTOM_LOCATION_SEARCH_TEMPLATE
  hreftemplate: '{repositoryUri}/x-search?locations={location}{&q,page,items-per-page}'
- name:      X_EXTERNAL_SEARCH_TEMPLATE
  href:      'http://www.google.com?q={keyword}'
  external:  true

---
resource-link-registry:
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/author'
  uri-template: 'X_AUTHOR_TEMPLATE'
  value-mapping: [userId:owner_name]
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/marketing-groups'
  uri-template: 'X_AUTHOR_TEMPLATE'
  value-mapping: [userId:owner_name,subject:subject]
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/folder-search'
  uri-template: 'X_CUSTOM_LOCATION_SEARCH_TEMPLATE'
  value-mapping: [location:r_folder_path]
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/gsearch'
  uri-template: 'X_EXTERNAL_SEARCH_TEMPLATE'
```

```
value-mapping:      [keyword:keywords]
```

Normalization of Custom URI Templates

Though it is no longer necessary to reference resources in YAML by defining custom URI templates, the custom URI template configuration in YAML is still useful when you want to:

- Build external links relations
- Build internal link relations with customizations. For example adding fixed query parameters, template parameters, and more.

Here are some samples of the URI template registry

```
uri-template-registry:
- name:      X_HELP_TEMPLATE
  href:      '{baseUri}/help{ext}'
- name:      X_POLICIES_FEED_TEMPLATE
  href:      '{repositoryUri}/objects/{objectId}/policies{ext}'
- name:      X_POLICY_TEMPLATE
  href:      '{repositoryUri}/objects/{objectId}/policies/{policyId}{ext}'
- name:      X_AUTHOR1_TEMPLATE
  href:      '{repositoryUri}/users/{userId}{ext}'
  encoding:  safe-text-encoding
- name:      X_AUTHOR2_TEMPLATE
  href:      '{repositoryUri}/users{ext}?name={userId}'
- name:      X_CUSTOM_SEARCH_TEMPLATE
  hreftemplate: '{baseUri}/x-search{?q,locations,page,items-per-page}'
- name:      X_CUSTOM_LOCATION_SEARCH_TEMPLATE
  hreftemplate: '{repositoryUri}/x-search?locations={path}{&q,page,items-per-page}'
- name:      X_EXTERNAL_SEARCH_TEMPLATE
  href:      'http://www.google.com?q={keyword}'
  external:  true
```

Normalization of Internal URI Templates

There are two URI prefix variables; *{baseUri}* and *{repositoryUri}* the other suffix variable *{ext}* that are applicable to internal resource URIs. When a URI template is internal (default to internal), the runtime automatically checks the URI format and appends the prefix or suffix where necessary.

For example, the internal URI template definitions below are equivalent:

- *{baseUri}/help{ext}* vs. */help* vs. *{baseUri}/help* vs. */help{ext}*
- *{repositoryUri}/users{ext}?name={userId}* vs. */repositories*
/ {repositoryName}/users?name={userId} vs. *{baseUri}/repositories*
/ {repositoryName}/users?name={userId}

Formats for href Template

When a URI template is defined as `hreftemplate`, there are href template parameters appended to the link. Documentum Platform REST Services supports a sub set of the RFC6570 standard.

- Appending template parameters `{?a,b,c,...}` onto a URI where there are no fixed query parameters.

For example: `/repositories/{repositoryName}/search?{?q,locations,...}`

- Appending template parameters `{&a,b,c,...}` onto a URI where there are fixed query parameters

For example: `/repositories/{repositoryName}/search?page=10&items-per-page={numberOfItems}{&q,locations,...}`

User Defined Variables of the URI Template

User defined variables can interfere with internal and external URI templates, including `href` or `hreftemplate` URI templates. The variables must be resolved either in custom Java code, or by using the variable value mapping in YAML.

Here's a code sample that shows you how to work with URI templates:

Example 7-78. Custom URI Templates with User Defined Variables

```
uri-template-registry:
- name:      X_AUTHOR_TEMPLATE
  href:      '{repositoryUri}/users/{userId}{ext}'
  encoding:  safe-text-encoding
- name:      X_MARKETING_GROUPS_TEMPLATE
  href:      '{repositoryUri}/groups{ext}?owner={userId}&subject={subject}&
             inline=true&items-per-page=50'
- name:      X_CUSTOM_LOCATION_SEARCH_TEMPLATE
  hreftemplate: '{repositoryUri}/x-search?locations={location}{&q,page,items-per-page}'
- name:      X_EXTERNAL_SEARCH_TEMPLATE
  href:      'http://www.google.com?q={keyword}'
  external:  true

---
resource-link-registry:
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/author'
  uri-template: 'X_AUTHOR_TEMPLATE'
  value-mapping: [userId:owner_name]
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/marketing-groups'
  uri-template: 'X_AUTHOR_TEMPLATE'
  value-mapping: [userId:owner_name,subject:subject]
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/folder-search'
  uri-template: 'X_CUSTOM_LOCATION_SEARCH_TEMPLATE'
  value-mapping: [location:r_folder_path]
- resource:  document
  link-relation: 'http://www.custom.com/sample/linkrel/gsearch'
  uri-template: 'X_EXTERNAL_SEARCH_TEMPLATE'
  value-mapping: [keyword:keywords]
```

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>
target-resource:<resource name register for this service>
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 for this resource>
```

Template Example using uri-template:

```
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]
```

Template Example using target-resource:

```
root-service-registry:
- name: User defined global search resource
link-relation: 'http://identifiers.emc.com/linkrel/x-search'
target-resource: 'acme#global-search'
allowed-methods: [POST]
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 13. Root Service Registry

Key	Value
name	Descriptive information of a resource registered to the Home Document.

Key	Value
uri-template	<p>URI template registered for this resource. The URI template must exist in the REGISTER NEW URI TEMPLATES section.</p> <p>For resources registered to the Home Document, only the following variables are allowed in a specified URI template:</p> <ul style="list-style-type: none"> • predefined variables {baseUri} and {ext} • variables that are treated as placeholders, such as query strings (<i>hreftemplate</i> only) <p>The predefined variable {repositoryUri} and any variables resolved via value-mapping settings are not allowed in the URI template.</p> <p>Note: One of the two attributes, <i>uri-template</i> or <i>target-resource</i> is required but not both. When both <i>uri-template</i> and <i>target-resource</i> are set, <i>target-resource</i> takes priority. Both of the <i>uri-template</i> and <i>target-resource</i> attributes cannot be left empty, at least one of them must contain a value.</p>
target-resource	<p>Allows users to reference an existing resource by name.</p> <p>For more information, see Note above.</p>
link-relation *	<p>Link relation name of a registered resource, which enables you to locate the resource in the Home Document.</p> <p>Only one link relation can be used to refer to a registered resource.</p>
allowed-methods *	<p>A comma separated list of the HTTP methods that can be performed on this resource.</p> <p>Example: [POST, GET]</p>
media-types *	<p>A comma separated list of the supported media types for this resource.</p> <p>Example: [application/vnd.emc.documentum+json, application/vnd.emc.documentum+xml]</p>
* 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 predefined 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-information"/>
    <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>
  target-resource:<the resource code name to link to>
  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>
```

Link Example Using uri-template:

```
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:[]
```

Link Example Using target-resource:

```
# build ACL resource link relation on core document resource using 'target-resource'
resource-link-registry:
- resource:      document
  link-relation: 'http://www.custom.com/sample/linkrel/acl'
  target-resource: 'acme#acl'
  value-mapping:  [aclName:acl_name]
```

This configuration allows you to register new links to Core resources. You can specify the following elements (key-value pairs):

Table 14. 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	<p>Value mappings are used to resolve values of path variables or query parameters on URI templates defined in uri-template.</p> <p>A value mapping follows the pattern: <code>[variableName1:propertyName1[,variableName2:propertyName2]]*</code></p> <p><i>variableName</i> represents a variable specified in a URI template. There cannot be any duplicated variables in a URI template.</p> <p><i>propertyName</i> 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.</p> <p>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.</p>
target-resource	Allows users to reference an existing resource by name.
* 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">
  <propertiesxsi: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>
```


Disabling Specific Resources

Disabling specific Core resources may become necessary for REST extensibility development work. For example, you may want to avoid exposing specific resources to end users, or you may want to develop a custom resource to override an existing core resource. In such scenarios, you can disable a specific core resource and package your own custom resource into the WAR.

Disable a Specific Resource with YAML

You can use the same YAML configuration file under `WEB-INF\classes` to define which resources will be disabled. Use the `disabling-resource-registry` attribute to define which core resources will be disabled.

There are some elements that you may want to reference when developing custom resources. For more information, see [Appendix B, Resource Coding Index](#).

Impact of Disabling Core Resources

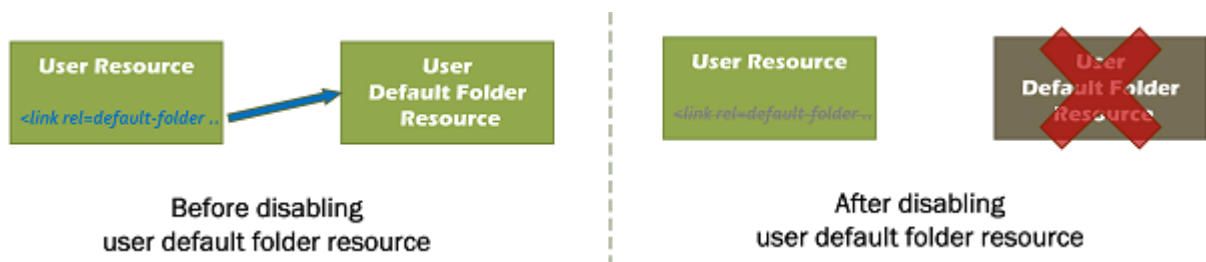
When you disable some particular core resources, it may cause link discovery of resources to be broken. In this case, a hypermedia driven client, which follows link relations to discover resources, cannot behave as expected.

It is imperative that you understand the impact of disabling resources before proceeding to disable any core resources. There are several different cases that impact representations of other resources when certain other resources are disabled.

For example:

- **Case 1:** Inactive link relations are removed on existing resources because certain other resources are disabled.

The user resource and the user default folder resource. When the user default folder resource is disabled, the link relation `http://identifiers.emc.com/linkrel/default-folder` on user resource is no longer presented. Here's an image that illustrates this case.



Example 7-79. Default User Resource XML Representation

The following code sample shows the default XML for the user resource, before the user default folder resource is disabled. Notice the link relation that is shown in bold.

```
<user xsi:type="dm_user"
```

```

definition="http://localhost/dctm-rest/repositories/REPO/types/dm_user"
xmlns="http://identifiers.emc.com/vocab/documentum"
xmlns:dm="http://identifiers.emc.com/vocab/documentum"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<properties>
  <user_name>dmadmin</user_name>
  <r_object_id>1100208080000501</r_object_id>
  ...
</properties>
<links>
  <link rel="self"
    href="http://localhost/repositories/REPO/users/dmadmin"/>
  <link rel="parent"
    href="http://localhost/repositories/REPO/groups?username=dmadmin"/>
  <link rel="http://identifiers.emc.com/linkrel/default-folder"
    href="localhost/repositories/REPO/users/dmadmin/home"/>
</links>
</user>

```

Example 7-80. User Resource XML Representation (user default resource disabled)

After the default folder resource is disabled, the link relation shown in bold in the preceding sample is no longer present in the XML representation.

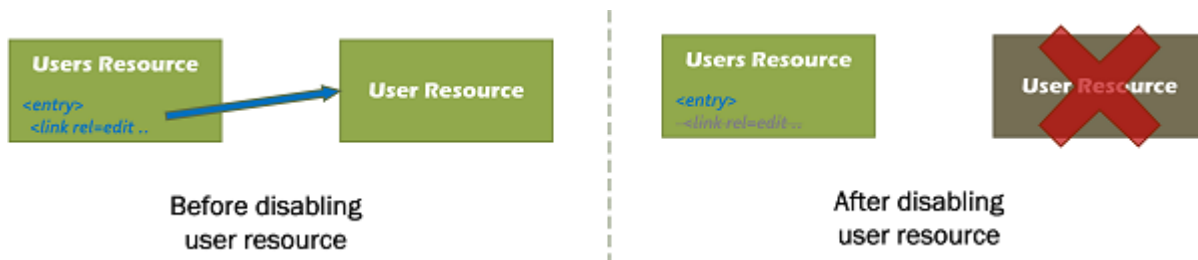
```

<user xsi:type="dm_user"
  definition="http://localhost/dctm-rest/repositories/REPO/types/dm_user"
  xmlns="http://identifiers.emc.com/vocab/documentum"
  xmlns:dm="http://identifiers.emc.com/vocab/documentum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <properties>
    <user_name>dmadmin</user_name>
    <r_object_id>1100208080000501</r_object_id>
    ...
  </properties>
  <links>
    <link rel="self"
      href="http://localhost/repositories/REPO/users/dmadmin"/>
    <link rel="parent"
      href="http://localhost/repositories/REPO/groups?username=dmadmin"/>
  </links>
</user>

```

- **Case 2:** The atom entry edit link is not present when the single object resource is disabled.

For example, the users resource and the user resource. When the user resource is disabled, the link relation edit on the users resource's entries is no longer present.



Example 7-81. Default Users Resource XML Representation

The following code sample shows the default XML for the users resource, before the user resource is disabled. Notice the link relation that is shown in bold:

```

<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/repositories/REPO/users</id>
<title>Users</title>
..
<link rel="self" href="http://localhost/repositories/REPO/users"/>
<entry>
  <id>http://localhost/repositories/REPO/users/Administrator</id>
  <title>dadmin</title>
  ...
  <content type="application/xml"
    src="http://localhost/repositories/REPO/users/dadmin"/>
  <link rel="edit" href="http://localhost/repositories/REPO/users/dadmin"/>
</entry>

```

Example 7-82. Users Resource XML Representation (user resource disabled)

After the user resource is disabled, the `edit` link relation shown in bold, is no longer present in the XML representation.

```

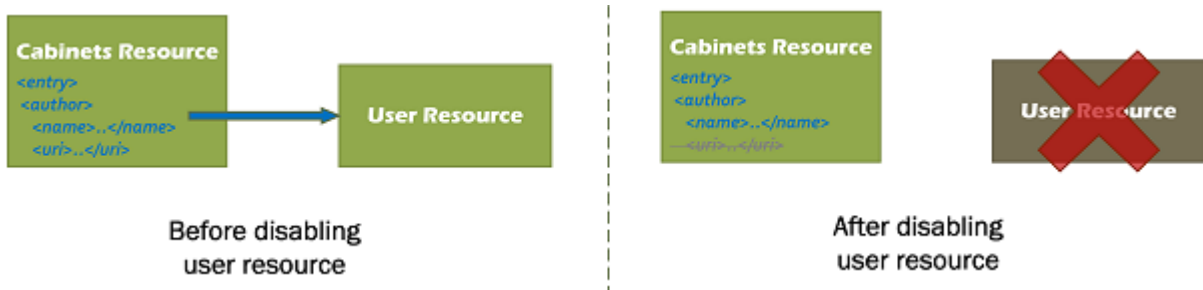
<feed xmlns="http://www.w3.org/2005/Atom"
  xmlns:dm="http://identifiers.emc.com/vocab/documentum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id>http://localhost/repositories/REPO/users</id>
<title>Users</title>
..
<link rel="self" href="http://localhost/repositories/REPO/users"/>
<entry>
  <id>http://localhost/repositories/REPO/users/Administrator</id>
  <title>dadmin</title>
  ...
</entry>

```

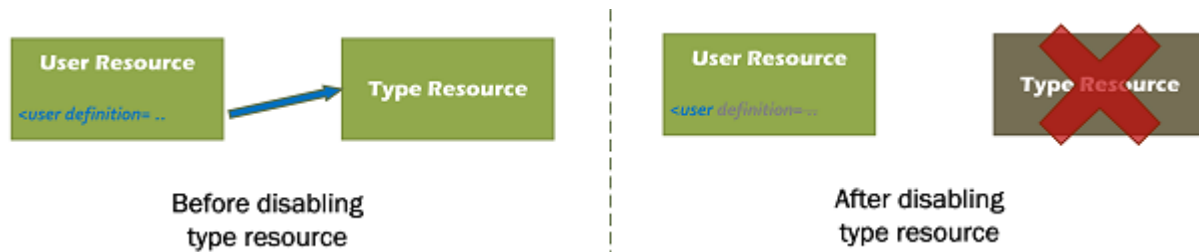
Note: Disabling resources has an impact on DQL query resources. When certain resources are disabled, the DQL query result does not provide the `edit` link for the query result items in their entries as expected.

The following diagrams illustrate other cases where disabling a resource has an impact on another resource.

When the user resource is disabled, the atom user URL in the atom feed is removed. One example is the cabinets resource.



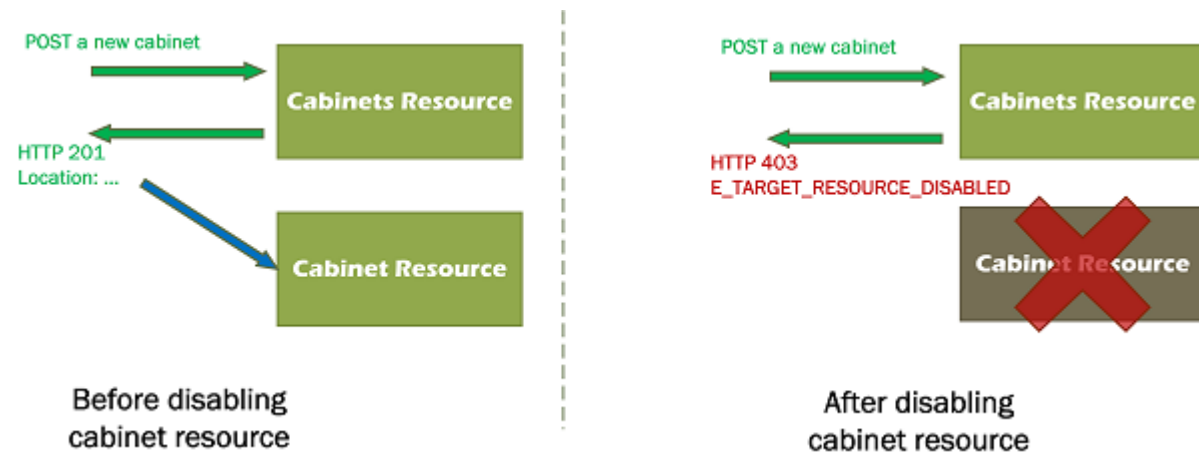
When the type resource is disabled, the type definition URL in the persistent object resource is removed. One example is the cabinet resource.



When the root is disabled, the link relation in the home document is removed. One example is the product information resource.



Some collection based resources support the HTTP POST method to create a new resource in the collections. When the single resource is disabled, the POST method on the collection resource fails. One example is the cabinets resource and cabinet resource.



Samples

Example 7-83. Disable a Resource Configuration

The `formats` resource and the `format` resource are disabled.

```
disabling-resource-registry:
- resources: [formats,format]
```

Representation

At runtime, both the `formats` and the `format` resources are not accessible.

```
GET /acme-rest/repositories/REPO/formats HTTP/1.1
```

```
GET /acme-rest/repositories/REPO/formats/crtext HTTP/1.1
```

```
Http 404 Not Found
```

In the repository resource, the link relation to formats is not available.

```
<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>
  <links>
    <link rel="self" href="/acme-rest/repositories/REPO"/>
    <link rel="http://identifiers.emc.com/linkrel/alias"
      href="/acme-rest/repositories/REPO/alias-sets"/>
    <link rel="http://identifiers.emc.com/linkrel/users"
      href="/acme-rest/repositories/REPO/users"/>
    <link rel="http://identifiers.emc.com/linkrel/current-user"
      href="/acme-rest/repositories/REPO/currentuser"/>
    <link rel="http://identifiers.emc.com/linkrel/groups"
      href="/acme-rest/repositories/REPO/groups"/>
    <link rel="http://identifiers.emc.com/linkrel/cabinets"
      href="/acme-rest/repositories/REPO/cabinets"/>
    <link rel="http://identifiers.emc.com/linkrel/network-locations"
      href="/acme-rest/repositories/REPO/network-locations"/>
    <link rel="http://identifiers.emc.com/linkrel/relations"
      href="/acme-rest/repositories/REPO/relations"/>
    <link rel="http://identifiers.emc.com/linkrel/relation-types"
      href="/acme-rest/repositories/REPO/relation-types"/>
    <link rel="http://identifiers.emc.com/linkrel/checked-out-objects"
      href="/acme-rest/repositories/REPO/checked-out-objects"/>
    <link rel="http://identifiers.emc.com/linkrel/types"
      href="/acme-rest/repositories/REPO/types"/>
    <link rel="http://identifiers.emc.com/linkrel/dql"
      hreftemplate="/acme-rest/repositories/REPO/{?dql,page,items-per-page}"/>
    <link rel="http://identifiers.emc.com/linkrel/search"
      hreftemplate="/acme-rest/repositories/REPO/search{?q,collections,locations,facet,
        inline,page,items-per-page,include-total,sort,view}"/>
    <link rel="http://identifiers.emc.com/linkrel/batches"
      href="/acme-rest/repositories/REPO/batches"/>
    <link rel="http://identifiers.emc.com/linkrel/batch-capabilities"
      href="/acme-rest/repositories/REPO/batch-capabilities"/>
  </links>
  ...
</repository>
```

Overriding Specific Resources

When a new resource overrides an existing resource, the new resource will become active, and the existing resource will become inactive. All link relations that were pointing to the existing resource, which is the one that is overridden, will now be pointing to new resource.

This configuration to develop new resources to deprecate Documentum Core resources.

You may be wondering why resources that have been overridden must implement the same URI patterns as Core resources. This is because the Core REST Link Builder checks whether the target URI pattern is active before writing that link relation to the resource. When the new resource URI is not implemented in the same way as the Core resource, the target URI in the link relation of the sourcing resource is inactive, and therefore it is not pointing to the new (overridden) resource.

To get around this limitation, you can introduce a new registry entry in YAML and explicitly tell a resource to override another resource. At runtime, you can also make the link relation builder understand the overriding relationship between two resources. Then the link relation targeting the original resource can be redirected dynamically to the new resource. This approach avoids the need for any hardcoded URI availability checking logic.

Resource Overriding Configuration

The `override-resource-registry` key in YAML causes the overridden resource to be replaced by the overriding resource. The resource that is overridden is disabled. Here is a code sample showing the override configuration key in YAML:

```
override-resource-registry:
  -new: new-document
    origin: document
  -new: new-user
    origin: user
```

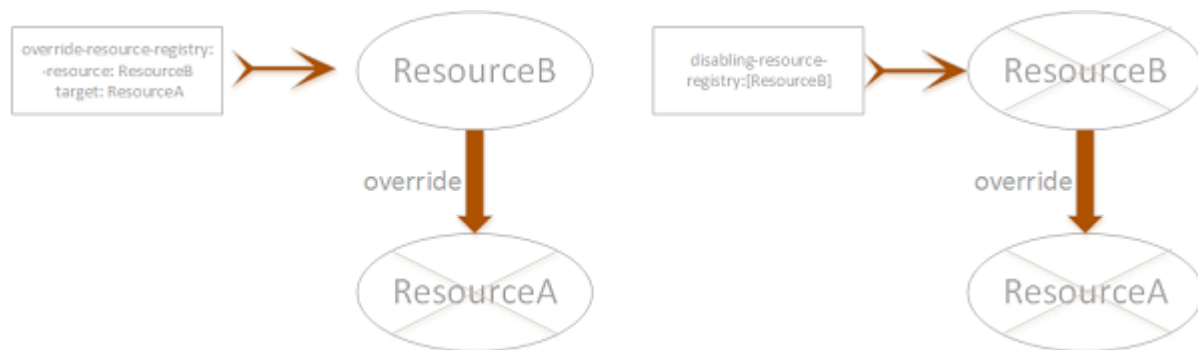
Key	Description
new	The name of the overriding resource
origin	The name of the resource that is overridden

Scenarios and Expected Results

Here are some examples of scenarios along with their expected results

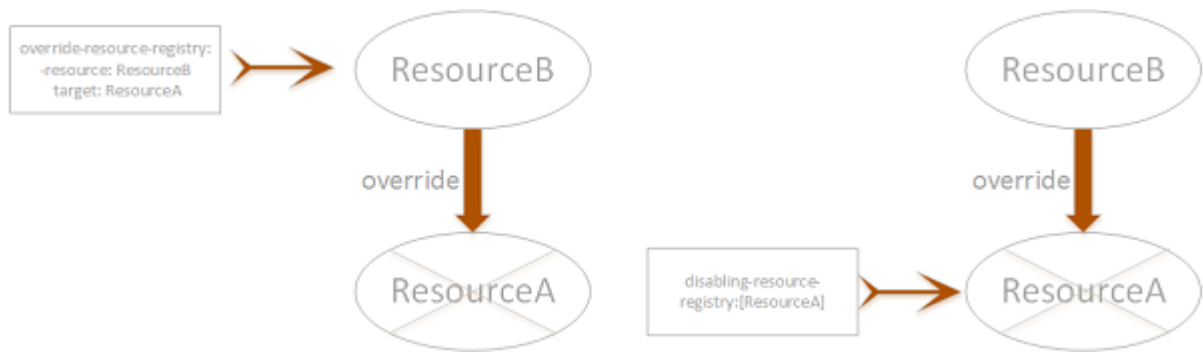
Example 7-84. Disable a Resource that Overrides Another Resource

When disabling an overriding resource, the overriding (Resource B) is disabled and the resource that is overridden resource (Resource A) is also disabled.

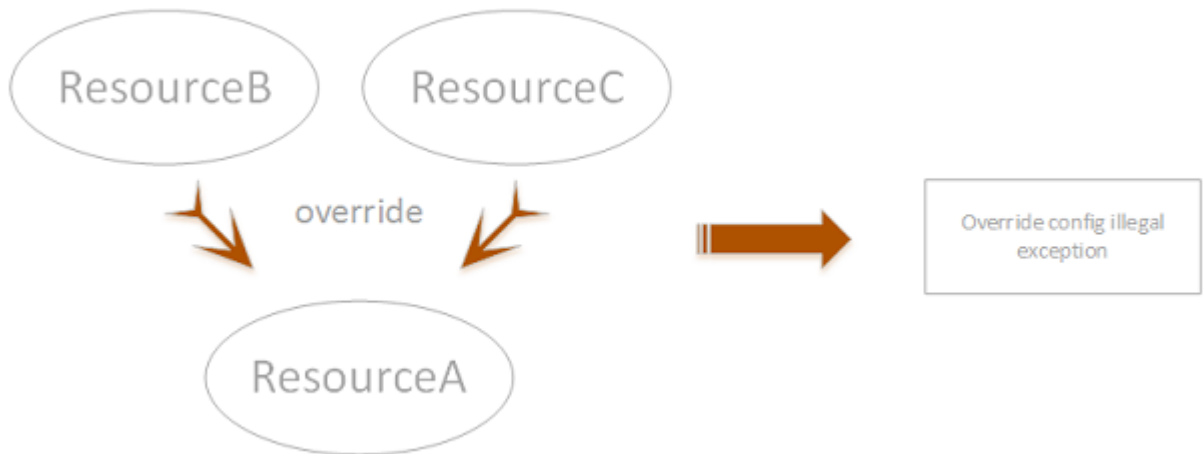


Example 7-85. Disable a Resource that is Overridden by Another Resource

In this example, no change takes place in the `disabling-resource-registry` setting since Resource A has been disabled in the `override-resource-registry` setting. The overriding resource (Resource B) remains active and the overridden resource (Resource A) remains disabled.

**Example 7-86. Two New Resources Incorrectly Override the Same Resource**

In this example, two new resources (Resource B and Resource C) are incorrectly configured to override the same resource (resource A). This configuration error leads to an `override config illegal` exception when deployed.



Working With a URI Template When Overriding

When you override a resource, the new (overriding) resource must have the same parameters, in the same order, as the resource that is being overridden. Documentum Platform REST Services validates the URL template of new resources to ensure that the new resource meets the above constraint. When the new resource being overridden does not meet the above constraint, initialization fails with an `override config illegal` exception and an exception detail message is written to the log.

How Resource Overriding Impacts Batchable Resources

When a resource is overridden:

- It is no longer in `batchable-resources`.
- It is no longer in `non-batchable-resources`.

The overriding resource can use the `BatchProhibition` annotation. Here is a listing of the impact that the `BatchProhibition` annotation has on the resource.

- The `BatchProhibition` is *true*:
 - The resource is no longer in `batchable-resources`.
 - The resource is in `non-batchable-resources`.
- The `BatchProhibition` is *false*:
 - The resource is in `batchable-resources`.
 - The resource is no longer in `non-batchable-resources`.

Turning Off XML or JSON Media Types

Documentum Platform REST Services supports both XML and JSON media types out-of-the-box. Documentum REST extensibility allows you to build application specific REST services. Typically, resources are developed in either XML or JSON but not both. This section discusses how to turn off the media type (XML or JSON) that you are not using.

An additional YAML configuration parameter has been exposed during the deployment phase so you can choose the media type that is supported. Here's a code sample to help illustrate the point:

```
---
# ##### SET SUPPORTED MEDIA TYPES (FOR INTERNAL USE) #####
# # Sets the supported media types at runtime. XML only or JSON only support can be
# # configured. The setting applies to both core resources and extended resources.
# # The syntax for a resource view definition is shown:
# #
# #   - media-type: <media type name>
# #
# # There are three choices at the moment.
# # - default, supporting both XML and JSON media types at runtime.
# # - xml, supporting XML media types only at runtime.
# # - json, supporting JSON media types only at runtime.
# # The default media type support is 'default'.
# #####
media-type-registry:
# - media-type: default
```

By default, both XML and JSON media types are supported:

- When *media-type:xml*, the only and default media type is XML. Any request with a JSON message body will get a 415 status code, and any request for a JSON response will get a 406 status code. The error message is always in XML
- When *media-type:json*, the only and default media type is JSON. Any request with a XML message body will get a 415 status code, and any request for an XML response will get a 406 status code. The error message is always in JSON
- When *media-type:default*, both XML and JSON are supported.



Caution: The setting has no impact on resource controller implementations. It just directs the Spring marshalling framework to choose the right supported media types when resolving an inbound or outbound HTTP message.

You can choose to declare one specific media type to support in resource controllers.

For example:

- *@RequestMapping* consumes
- *@RequestMapping*. produces

Here are a couple of code samples to help illustrate the point. The first code sample shows you how to declare both XML and JSON media types:

```
@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
    },
    consumes = {
        SupportedMediaTypes.APPLICATION_VND_DCTM_JSON_STRING,
        SupportedMediaTypes.APPLICATION_VND_DCTM_XML_STRING
    })
@ResponseBody
@ResponseStatus(HttpStatus.CREATED)
public CabinetObject createCabinet()
```

The next code sample shows you how to declare XML media type:

```
@RequestMapping(
    method = RequestMethod.POST,
    produces = {
        SupportedMediaTypes.APPLICATION_VND_DCTM_XML_STRING,
        MediaType.APPLICATION_XML_VALUE
    },
    consumes = {
        SupportedMediaTypes.APPLICATION_VND_DCTM_XML_STRING
    })
@ResponseBody
@ResponseStatus(HttpStatus.CREATED)
public CabinetObject createCabinet()
```

Shown in bold in the preceding code samples, the *produces* attribute sets the supported media type for messages that are created on the server. In this case, we have chosen to have XML messages created and sent from the server. There is also a *consumes* attribute that defines the media type, in this case XML, that is accepted by your REST client.

The preferable solution is to declare both XML and JSON support in resource controllers and set the specific media type support in YAML file. There is no additional development cost when you declare media type support in resource controllers, and there is the flexibility to configure the YAML file that defines the media type support.

Creating Custom Error Code Mapping Files

Documentum Platform REST Services allows the client to create error code mapping JSON files. The name of a custom error code mapping JSON file must follow this pattern:
`rest-*error-mapping.json`

Where *** can be any string of characters. For example: `rest-myresource-error-mapping.json`.

You must ensure that the package where you create your custom error code mapping files is specified in the property `rest.ext.error.code.mapping.packages` within the `rest-api-runtime.properties` file.

Creating Custom Message Files

Documentum Platform 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, which is found within the `rest-api-runtime` properties file.

Tutorial: Documentum Platform REST Services Extensibility Development

This tutorial walks through a reference implementation of custom resources using the Documentum REST extensibility feature. It is 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 15. 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"}],
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 16. 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+xml application/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"/>
</links>
</alias-set>

```

JSON representation for the alias set resource

```

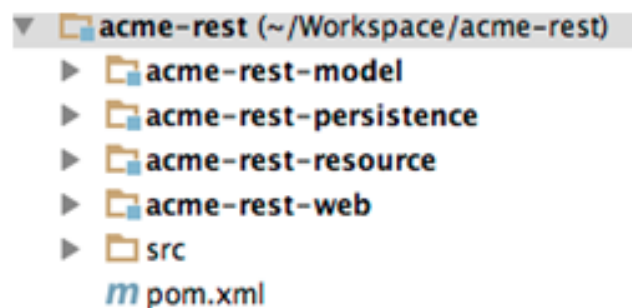
{
  "name": "alias-set",
  "type": "dm_alias_set",
  "definition": "http://localhost:8080/acme-rest/repositories/REPO/
types/dm_alias_set",
  "properties": {
    "owner_name": "Administrator",
    "object_name": "AdminAccess",
    ...
  },
  "links": [
    { "rel": "self", "href": "http://localhost:8080/acme-rest/repositories/
REPO/alias-sets/6600208080000105" },
    { "rel": "author", "href": "http://localhost:8080/acme-rest/repositories/
REPO/users/Administrator" }
  ]
}

```

Quickstart

Documentum Platform 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 10. 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

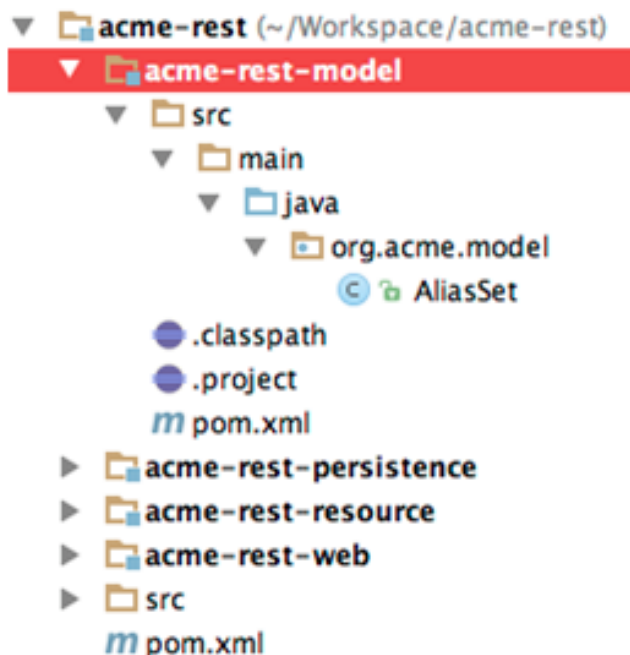
update the alias set object using DFC. The resource module creates the resource controllers. And the web module creates the WAR file for the entire custom services.

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 11. Structure of Model



The class definition for `AliasSet` is shown:

```
/**
 * 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 Platform 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:

```
<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>
```

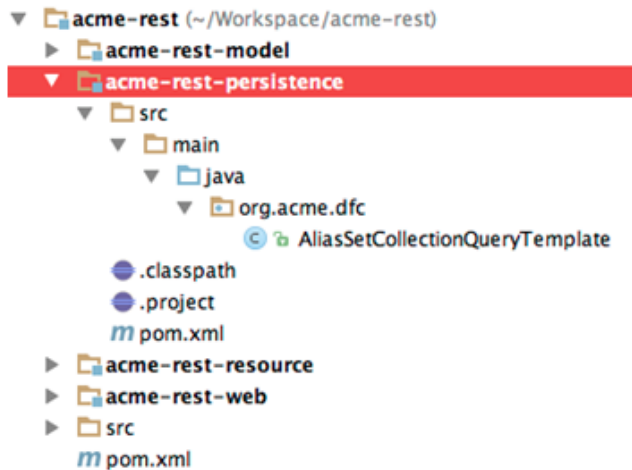
The section [Annotation Scanner](#) provides more information about Documentum REST annotation validation.

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 Platform 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 12. 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; }
}
```

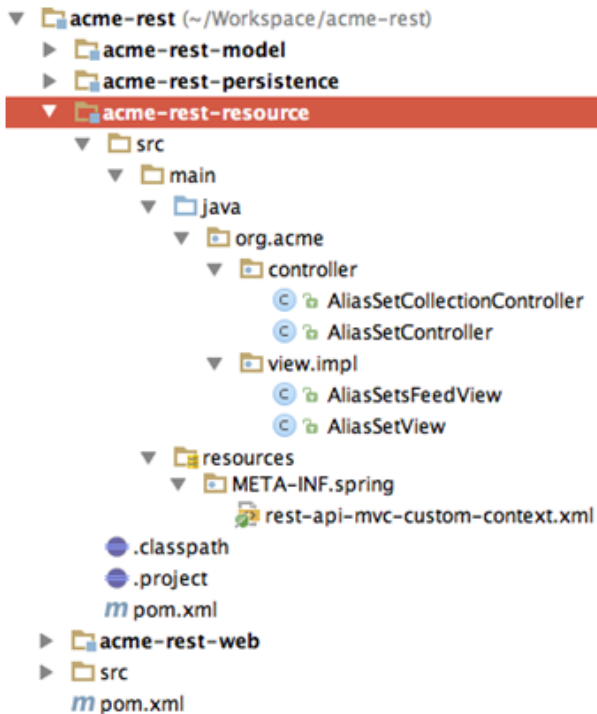
`PagedQueryTemplate` is an out-of-the-box class that defines the abstraction for the DQL query string construction for a type-specific collection. A collection resource's persistence API can extend this class to build the DQL query string for the returning collection. The next section shows an example 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 13. 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 Platform REST Services annotations

- `@TypedParam final CollectionParam param` - a Documentum Platform 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 Platform 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 Platform 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.

Spring Context Configuration

In addition to the resource controller definition, you need to define the Spring context configuration to scan the controller classes during the server startup. This Spring configuration file tells the Spring framework where to load the controllers.

Note: You must use the `com.emc.documentum.rest.context.ComponentScanExcludeFilter` exclude filter when you use the Spring `@ComponentScan` annotation in your custom defined configuration class. This exclude filter ensures that the Spring framework loads all of the resources that you have defined.

Here is a code sample that demonstrates this technique:

```
@Configuration
@ComponentScan(basePackages = "org.acme",
    excludeFilters = { @ComponentScan.Filter(type = FilterType.CUSTOM,
        value = { com.emc.documentum.rest.context.ComponentScanExcludeFilter.class }) })
public class ExtensionContextConfig {
}
```

You must ensure that the package where you created the `ExtensionContextConfig` class is specified in the `rest.context.config.location` property in the `rest-api-runtime.properties` file.

The above is equivalent to the following XML configuration:

```
<context:component-scan base-package="org.acme" use-default-filters="true">
    <context:exclude-filter type="custom"
        expression="com.emc.documentum.rest.context.ComponentScanExcludeFilter"/>
</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 ownerLink = ResourceUriBuilder
            .onResource("user")
            .pathVariables((String) serializableData.getAttributeByName("owner_name"));
        return Arrays.asList(new AtomAuthor(owner, ownerLink, null));
    }

    @Override
    public AliasSet entryContent() { return data(); }

    @Override
    public String entrySrc() { return canonicalResourceUri(boolean validate); }
}

```

```
@Override
public void customize() {
    String ownerLink = ResourceUriBuilder
        .onResource("user")
        .pathVariables((String)serializableData.getAttributeByName("owner_name"));
}

@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 single 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 17. 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+xml application/vnd.emc .documentum+json
		POST	application/vnd.emc .documentum+xml application/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 Platform 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>
...

```

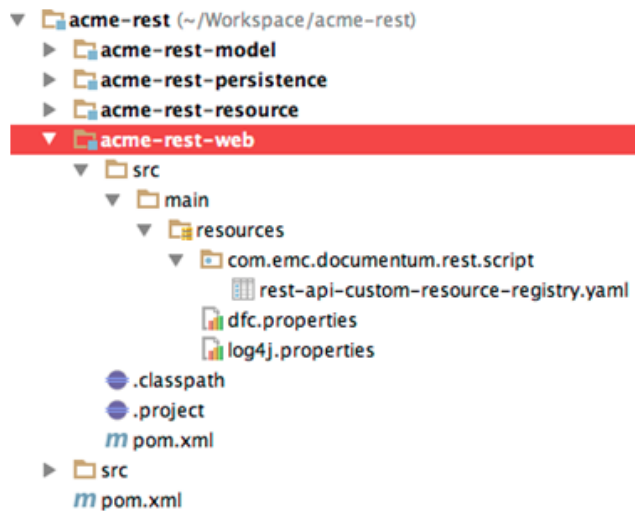
Making Resources Linkable

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:

Figure 14. 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">7.1.0010.0158Win64.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 batchable. 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 in the directory `/acme-rest-web/target`, which contains both Core resources and new resources.

Troubleshooting

During 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.

Authentication Extensibility

You can use authentication extensibility to intercept the authentication flow or it can be used to create a custom authentication scheme for Documentum Platform REST Services.

Anonymous Access

Anonymous Access is an extension based on Documentum Platform REST Services security framework. This extension can be used to configure anonymous access for custom resources or static resource files in your custom REST services. There are two user scenarios for this feature:

- Add new static resource files into your REST application. An example of this is adding custom CSS style sheets or JavaScript files.
 - If a static resource file URL is not configured with authentication or anonymous access, that request is rejected with a 403 status code.
 - Anonymous configuration provides users with a way to bypass the authentication process.
- Add custom anonymous REST resources into a custom REST services. An example of this is a resource that contains a built-in login user.
 - Your particular REST services may have a built-in login user on the server side, and you want to skip the usual authentication schema that is already configured.

Implementation

You must create a custom Java controller that uses the `@AnonymousAccess` annotation to get the functionality that you want. There are two ways to set anonymous access in your custom REST service:

- By Java annotation using `@AnonymousAccess`
- By configuration using `rest-api-runtime.properties` file

Note: The two approaches shown here do not need to be implemented at the same time for one resource. They can take effect individually.

By Java Annotation

In Spring you can plug add-on features to resource controllers as annotations, which allows you to do many things using Spring itself while reducing the amount of work you have to do by using Reflection.

Typically, all REST service functionality is implemented into one controller that uses the `@AnonymousAccess` annotation. Your controller can provide anonymous access functionality to your REST application. The `@AnonymousAccess` annotation allows:

- All resource controllers to be excluded from a regular configured authentication schema.
- All resource controllers to be provided with their own repository security context, initialization, and cleanup procedures.

`@AnonymousAccess` contains two fields:

- `contextInitializerClass`

This is an implementation of the `contextInitializerClass` interface. This interface defines the customized details for your context, such as username and password.

- `contextCleanerClass`

This field extends from `DefaultRepositoryContextCleaner`, and is used to clean up any security related context.

Example 8-1. Typical Usage Scenario

```
@AnonymousAccess (
contextInitializerClass = MyRepositoryContextInitializer.class,
contextCleanerClass = DefaultRepositoryContextCleaner.class
)
@Controller("acme#custom-resource")
public class MyAnonymousResourceController extends AbstractController {...}
```

The `initialize()` and `clean()` methods can be implemented as needed:

- For example, during initialization you can use the `initialize()` method to set the repository name, login name and password in `RepositoryContextHolder`. The `authType` can be set to `AuthType.PASSWORD`, which indicates that user authentication will be done using a login name and password.
- The default `clean()` method is used to clean the context in `RepositoryContextHolder`. When you have more complex logic for cleaning context, you can override this method with you own custom code to perform the cleaning.

By Runtime Configuration

Runtime configuration is not recommended for the implementation of anonymous REST services, but it is the only way to configure anonymous static resource files.

A runtime property called `rest.security.anonymous.url.patterns=` is added to `rest-api-runtime.properties`. This property value must be written as Ant style URL patterns. Multiple URL patterns are created as a comma-separated values.

Example 8-2. Configuration in rest-api-runtime.properties

```
rest.security.anonymous.url.patterns=/another-anonymous-resource,  
/one-more-resource
```

The patterns defined in it are added to any other anonymous patterns.

Extension Samples

There are samples named `documentum-rest-anonymous-samples` available in the `documentum-rest-anonymous-samples` package. These samples illustrate how to use this security extension, and both approaches are demonstrated in them. To get these samples, see the `README.md` file in the package and follow the instructions provided in it.

Generic Servlet Filter Customization

In Documentum Platform REST Services the `SpringSecurityFilterChain` is created by Java code. Your REST applications may want to inject more HTTP servlet filters into your REST application to handle the requests with additional business logic, such as counting requests, logging specific requests, and more.

In Documentum REST 7.3, an XML configuration file is exposed to allow you to inject custom (non-security) filters before or after `SpringSecurityFilterChain` is created.

Note: Modifying the `web.xml` directly in the WAR file is not recommended because the filter sequence cannot be guaranteed.

Here's a code sample that shows you how to edit the `<dctm-rest.war>\WEB-INF\classes\META-INF\spring\extension\rest-api-custom-filters.xml` file for customizing servlet filters:

Example 8-3. /META-INF/spring/extension/rest-api-custom-filters.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    <!-- allow users to add custom filters into the REST servlet; all filters are
         mapped to the root context '/' -->
    <bean id="restFilterFactory" class="com.emc.documentum.rest.filter.FilterFactory">
        <property name="customBeforeSecurityFilters">
            <list>
                <!-- an ordered list of filters added before springSecurityFilterChain;
                     empty by default -->
            </list>
        </property>
        <property name="customAfterSecurityFilters">
            <list>
                <!-- an ordered list of filters added after springSecurityFilterChain;
                     empty by default -->
            </list>
        </property>
    </bean>
</beans>
```

In a custom REST application, you can update this file to inject filters. The REST SDK provides such a sample for you to see:

Example 8-4. Sample rest-api-custom-filters.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

    <!-- allow users to add custom filters into the REST servlet; all filters are
         mapped to the root context '/' -->
    <bean id="restFilterFactory" class="com.emc.documentum.rest.filter.FilterFactory">
        <property name="customBeforeSecurityFilters">
            <list>
                <!-- an ordered list of filters added before springSecurityFilterChain;
                     empty by default -->
                <bean class="com.emc.documentum.rest.sample.filter.RequestCountingFilter"/>
            </list>
        </property>
    </bean>
</beans>
```



```

    </property>
    <property name="customAfterSecurityFilters">
        <list>
            <!-- an ordered list of filters added after springSecurityFilterChain;
                 empty by default -->
            <bean class="com.emc.documentum.rest.sample.filter
                    .RequestCounterCountingFilter"/>
        </list>
    </property>
</bean>
</beans>

```

Here is the entire filter chain sequence, ordered as shown below:

```

MessageLoggingFilter
↓
CharacterEncodingFilter
↓
RepositoryNamingFilter
↓
<customBeforeSecurityFilters ..>
↓
SpringSecurityFilterChain
↓
<customAfterSecurityFilters ..>
↓
HiddenHttpMethodFilter
↓
ApplicationFilter

```

Custom Authentication Development

This section discusses how to develop a custom authentication scheme in Documentum Platform REST Services 7.3 and later. Documentum Platform REST Services provides you with a rich set of authentication schemes. However, you may want to add their own authentication schemes that are not supported out-of-the-box.. For example *OAuth*, *OpenID*, *Two-factor authentication*, and others.

There are various requirements for custom authentication deployment, to accommodate many of these requirements we have exposed the authentication extension development framework, which allows you to develop a custom authentication scheme according to your specific needs.

Understanding The Security Filter Flows

The REST authentication framework leverages the security found in the Spring framework to setup the servlet security context.

For more information, see [Generic Servlet Filter Customization, page 320](#) in Documentum Platform REST Services to understand how the Spring Security Filter Chain takes place within the entire servlet filter chain.

Specific for the security filter chain itself, the Spring Security framework puts multiple HTTP security configurations into a chain called `SpringSecurityFilterChain`. The high level flow of the security filter chain in Core REST is as shown:

Anonymous access URL interceptors

-

Core or Custom Authentication Filter (s) for URL pattern `/repositories/**`

-

Access denial security filter

For any request that is sent to the REST server, the following applies:

- When the Request URL maps to an anonymous access URL pattern, the Request is bypassed by the resource controller without security handling. Some static resources, such as `js`, `css` files, follow this pattern. The Core REST authentication framework exposes an extension for you to configure anonymous access resources or URL patterns.

For more information, see [Anonymous Access](#), page 317.

- The out-of-the-box authentication schemes or custom authentications manage the Request authentication for the resource URL pattern `/repositories/**` by default.

The URL pattern can be configured globally in `rest-api-runtime.properties` with the property name `rest.security.auth.root.url`.

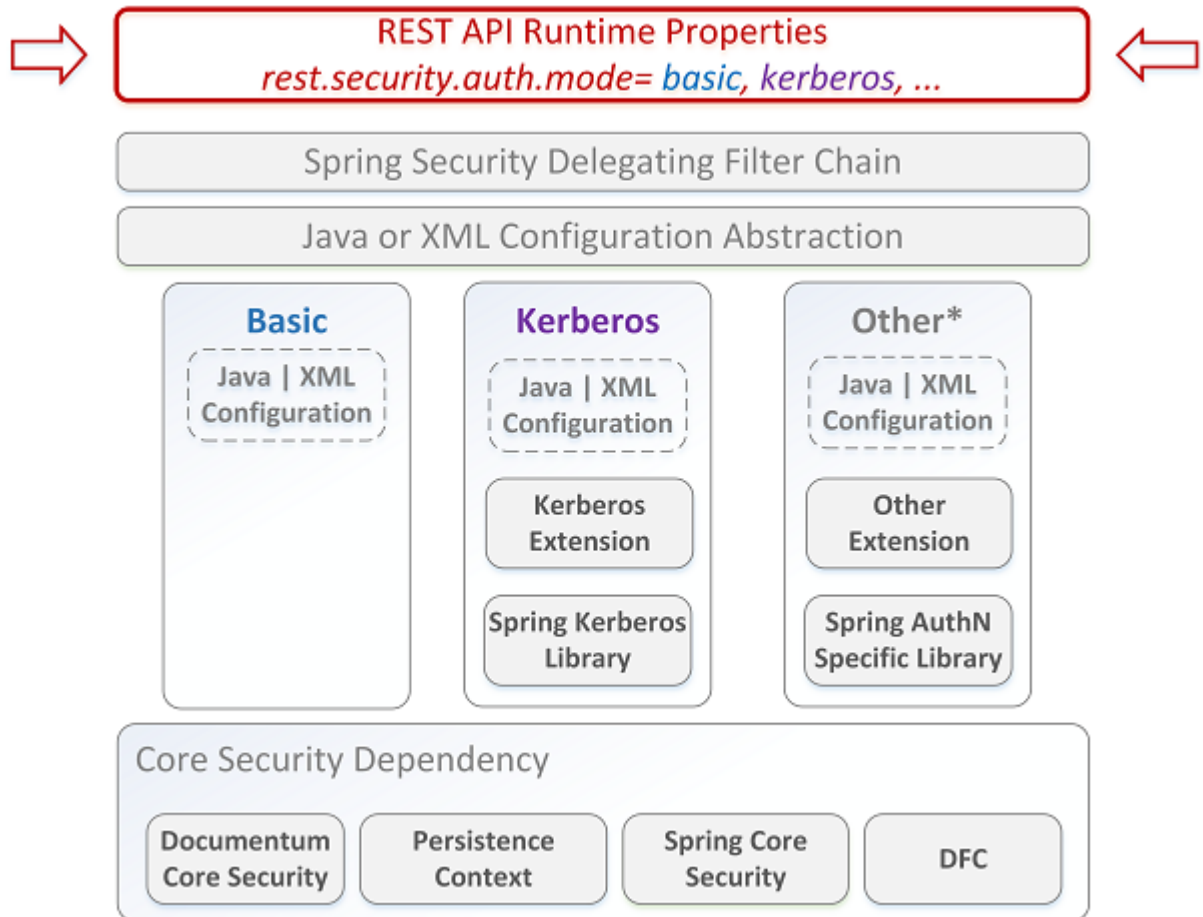
- For other cases not shown above, the Request is rejected with a 403 status code.

Understanding The Authentication Framework

As a Core REST custom authentication developer, we assume you are familiar with the following:

- Spring Security 4.x / 3.x
- Maven
- Documentum Platform REST Services Extensibility

The Documentum Platform REST Services authentication framework has a layered and extensible architecture, as illustrated in the diagram shown. An individual authentication scheme in Documentum Platform REST Services is taken as a configuration of Spring HTTP Security:



- Configuration parameters, such as `rest.security.auth.mode`, in the REST runtime properties file provides the uniform interface for you to choose which authentication scheme to use at runtime.
- Following the runtime configuration, Spring Security loads the corresponding configurations from the `class-path`.
- More specifically, Spring security looks for the effective Java or XML configuration to load a configured authentication scheme.
- There may be multiple authentication scheme implementations deployed to one server, but only the one specified in the runtime properties is used.
- Each authentication scheme has its own Java or XML configuration, its own implementation, and dependent libraries.
- Documentum Platform REST Services provides a common set of security libraries to facilitate the development of authentication, such as client token integration, session manager utility, and more.

The authentication flow for a Documentum Platform REST Services authentication scheme is similar to any typical Spring security authentication flow. The following diagram shows the flow of a typical authentication:

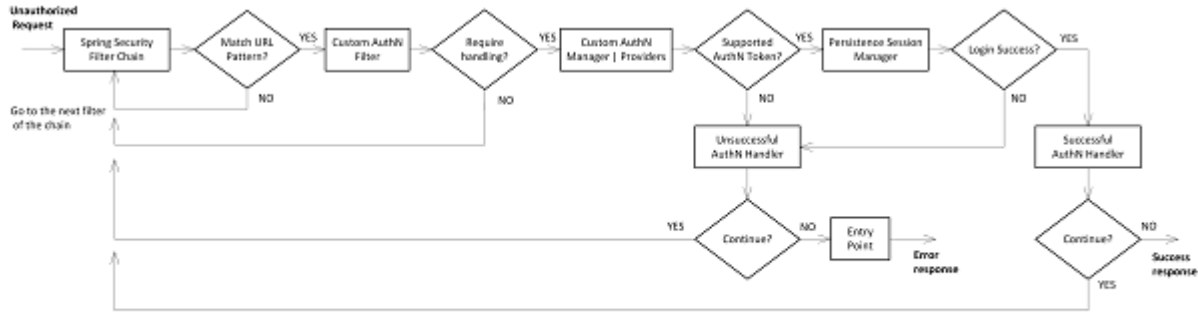


Diagram-2 A Typical Authentication Flow

- When an unauthorized request comes, Spring Security examines the Request and finds a matching security filter to handle the Request.
- Within the authentication filter, there is the logic to determine whether to process the Request or not, usually by examining the Request information and the local security context.
- When the authentication filter determines that it should authenticate the Request, it extracts the Request information into a request token, such as an `Authorization` header, and calls its wired authentication manager to handle the Request.
- Each authentication manager may have more than one authentication provider. Each provider calls the persistence authentication manager to authenticate the Request token.
- The persistence authentication manager calls the DFC session manager to authenticate the user in the Request. For Documentum, the authentication success is the same as the successful retrieval of the DFC session.
- When the token is not supported or not authenticated, an unsuccessful authentication handler handles the error. The handler usually clears the local security context and delegates to an entry point to return the error response, such as a 401 status code.
- When the token is authenticated, a successful authentication handler can return the Response directly or it can delegate it to other filters in the chain.
- After the successful authentication, the authenticated token is stored in the local security context, to be used by the controller that handles the business logic of the Request. For Documentum Platform REST Services, `RepositoryContextHolder` stores the authenticated user credential during the lifecycle of the Request handling.
- After the Request handling finishes, the security context can determine whether to clear up the authenticated token or cache it for a stateful session. For Documentum Platform REST Services, `RepositoryContextHolder` always clears up the local security context by design.

During the deployment phase, you are usually only required to configure REST runtime properties to enable a specific authentication scheme. For example **`rest.security.auth.mode=basic`**

Authentication Extension

The custom authentication scheme provides the same convenience of deployment as other Core authentication schemes. It is important to follow the Core REST authentication extension development model to develop the custom authentication scheme. Similar to Documentum REST

Extensibility, authentication extension development requires you to use Documentum Platform REST Services Archetype to create a sub module for the custom authentication scheme. After you've created your custom authentication scheme, you must package it as a dependent library of the Documentum Platform REST Services WAR. The development lifecycle of a custom authentication scheme is illustrated below:

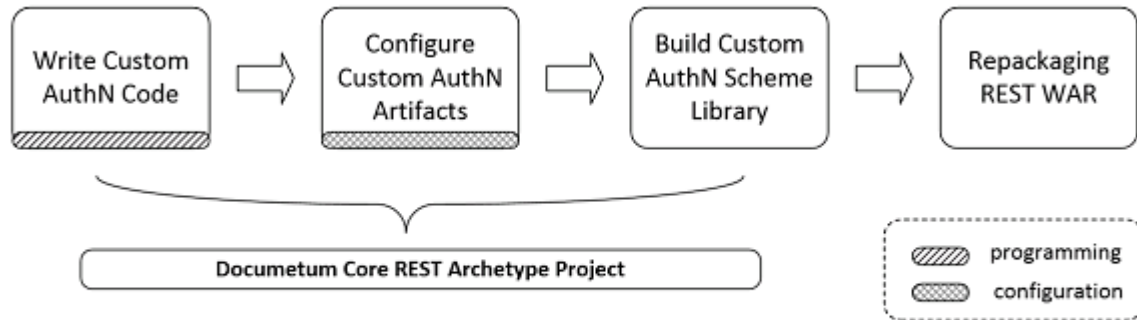


Diagram-3 Lifecycle of Building Custom Authentication Schemes

The amount of work required to develop a custom authentication scheme depends on third party software used and Documentum dependency support. The following diagram shows the implementation stack of a custom authentication scheme and possible extension points:

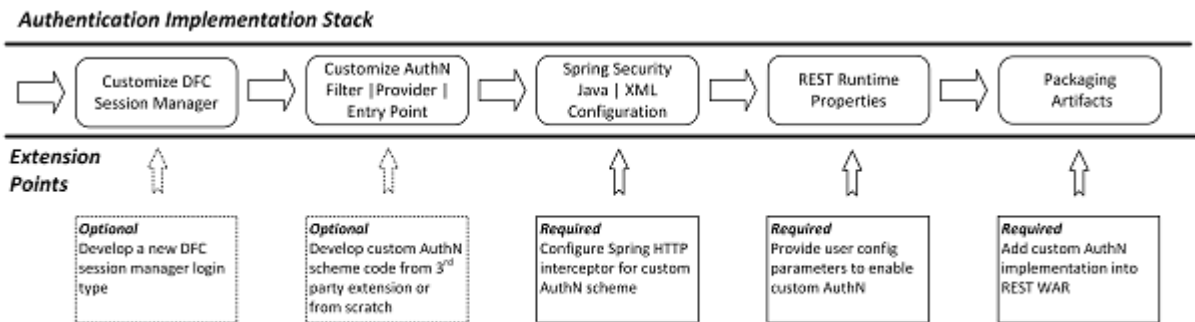


Diagram-4 Custom Authentication Scheme Implementation Stack And Extension Points

Customize DFC Session Manager

The login for Content Server requires a DFC session manager, which must be set up. Documentum Platform REST Services provides an interface called `com.emc.documentum.rest.dfc.RepositorySessionManager` for REST server code to instantiate the DFC session manager. This interface provides a public method:

```
public IDfSessionManager get(String repositoryName, String user, Object credential, AuthType authType)
```

Depending on the authentication type used, the credentials are different for each login user. The factory supports the following authentication types:

Authentication Type	Description	Credential
DEFAULT	Default login , same to PASSWORD.	User password

Authentication Type	Description	Credential
PASSWORD	Password login	User password
LOGIN_TICKET	Documentum Login Ticket login	Login ticket in BASE64 string
CAS_PROXY	CAS SSO login	CAS AttributePrincipal
KERBEROS_TGT	Kerberos SSO login	Kerberos TGT as binary
PRINCIPAL	Content Server principal login	No credentials
RSA	RSA ClearTrust SSO login	RSA token
SITEMINDER	CA SiteMinder SSO login	SiteMinder token
CUSTOM	Preserved for user defined login	User defined credential

Documentum Platform REST Services provides a default implementation for the `com.emc.documentum.rest.dfc.RepositorySessionManager` interface. The implementation class is `com.emc.documentum.rest.dfc.impl.MemoryRepositorySessionManager`. As its name implies, this class puts the initialized session manager object into server memory (*Ehcache*) for reuse. Caching gives it the best performance with Content Server login.

Note: To avoid possible session leaks, the `com.emc.documentum.rest.dfc.impl.MemoryRepositorySessionManager` class never caches the DFC session. It is recommended that you use the default implementation to initialize DFC sessions.

There are two ways to get the implementation instance:

1. Call `MemoryRepositorySessionManager.INSTANCE`. The implementation is a Singleton, so the easiest way to use it is to call its Singleton called `INSTANCE`.
2. Get it from `com.emc.documentum.rest.security.provider.AbstractAuthProvider`. You can get the session manager instance by extending from the abstract provider.

When To Customize

In most cases, you do not need to customize DFC session managers because the default implementation already has support for most authentication types that are also supported by Content Server.

A custom authentication scheme should first examine existing authentication types to see whether any of them can be used for the custom authentication scheme. You can use the `CUSTOM` authentication type when none of the default implementations meet your requirements. The default implementation creates an empty DFC session manager for the authentication type `CUSTOM`. You must set up custom identities for the obtained session manager. Here's a code sample that shows you a custom DFC manager:

```
// Get memory repository session manager
RepositorySessionManager repoSessionManager = MemoryRepositorySessionManager.INSTANCE;
IDfSessionManager dfcSessionManager = repoSessionManager.get("ACME", "dave", "5XHR6fTZ==",
    AuthType.CUSTOM);

// Check whether it is initialized -- if it is not the first time you login
// then it may already be initialized
if (!dfcSessionManager.hasIdentity("ACME")) {
```

```
// Set identity
IDfLoginInfo login = new DfLoginInfo();
login.setUser("dave");

// This is a dummy password plugin sample -- set a valid password plugin which is
supported by DFC login.setPassword("DM_PLUGIN=dm_custom/" + "5XHR6fTZ==");
dfSessionManager.setIdentity("ACME", login);
}
```

Customize Authentication Filter, Provider, and Entry Point

Spring Security introduces the concepts of authentication filter, manager, provider, and entry point. They work together to authenticate a web request. Here's a diagram that shows the Spring web security classes.

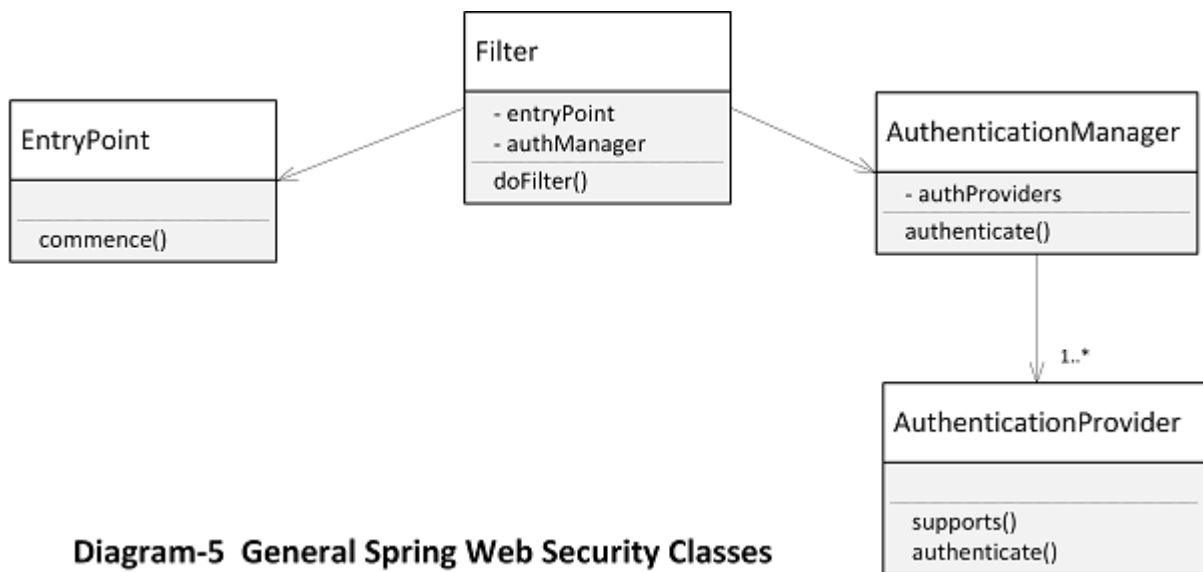


Diagram-5 General Spring Web Security Classes

A custom authentication scheme may directly use an out-of-the-box security filter, such as a Spring dependency, to authenticate the Request, or it may extend from third party software. Spring Security provides a number out-of-the-box security filters. You can also add your own filters.

Authentication managers provide the authentication services for filters. You cannot customize the authentication managers, but you can register different authentication providers.

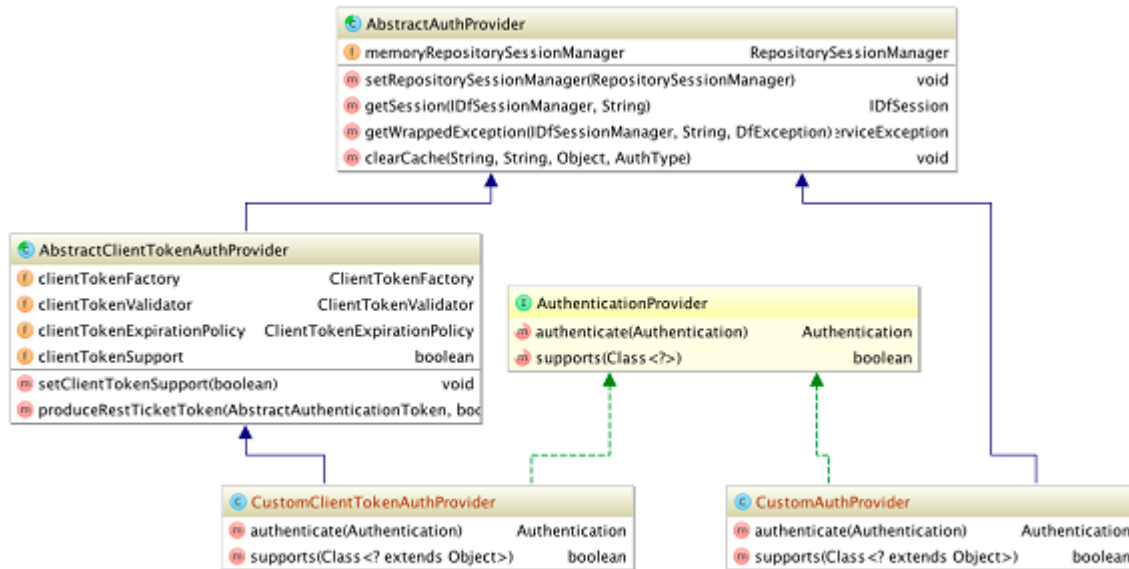
Each authentication provider authenticates a specific authentication token.

Authentication Provider

Developers often write their own authentication providers, because the provider must call the Core REST session manager to authenticate the user. Spring Security provides an interface called *AuthenticationProvider*, which is used to implement the provider. Documentum Platform

REST Services additionally provides two abstract classes for extension, *AbstractAuthProvider* and *AbstractClientTokenAuthProvider*

- *AbstractAuthProvider* provides methods to call the DFC session manager to authenticate the user. Extending it when the authentication requires a Content Server login.
- *AbstractClientTokenAuthProvider* provides methods to exchange the authenticated token with a Client Token. Extending it when authentication success requires a Content Server login, and return a Client Token cookie.



Example 8-5. CustomAuthenticationProvider

```

public class CustomAuthenticationProvider extends AbstractAuthProvider
    implements AuthenticationProvider {

    @Override
    public boolean supports(Class<? extends Object> authentication) {
        return UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);
    }

    @Override
    public Authentication authenticate(Authentication authentication)
        throws AuthenticationException {
        if (authentication == null || Strings.isNullOrEmpty(authentication.getName())) {
            throw new UsernameNotFoundException(MessageBundle.INSTANCE
                .get("E_BAD_CREDENTIALS_ERROR"));
        }

        String repositoryId = RepositoryContextHolder.getRepositoryName();
        String user = (String) authentication.getPrincipal();
        String password = (String) authentication.getCredentials();
        IDfSessionManager sm = null;
        IDfSession session = null;
        try {
            sm = memoryRepositorySessionManager.get(repositoryId, user, password,
                AuthType.PASSWORD);
            session = getSession(sm, repositoryId);
            Collection<GrantedAuthority> grantedAuthorities = new ArrayList<GrantedAuthority>();
            grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_USER"));
            AbstractAuthenticationToken authorizedToken = new UsernamePasswordAuthenticationToken(
                authentication.getPrincipal(),
  
```



```

        authentication.getCredentials(),
        grantedAuthorities);
    if (!authorizedToken.isAuthenticated()) {
        authorizedToken.setAuthenticated(true);
    }
    return authorizedToken;
} catch (DfException e) {
    clearCache(repositoryId, user, password, AuthType.PASSWORD);
    throw getWrappedException(sm, repositoryId, e);
} finally {
    DfcSessions.release(session);
}
}
}

```

Example 8-6. CustomAuthenticationProvider

```

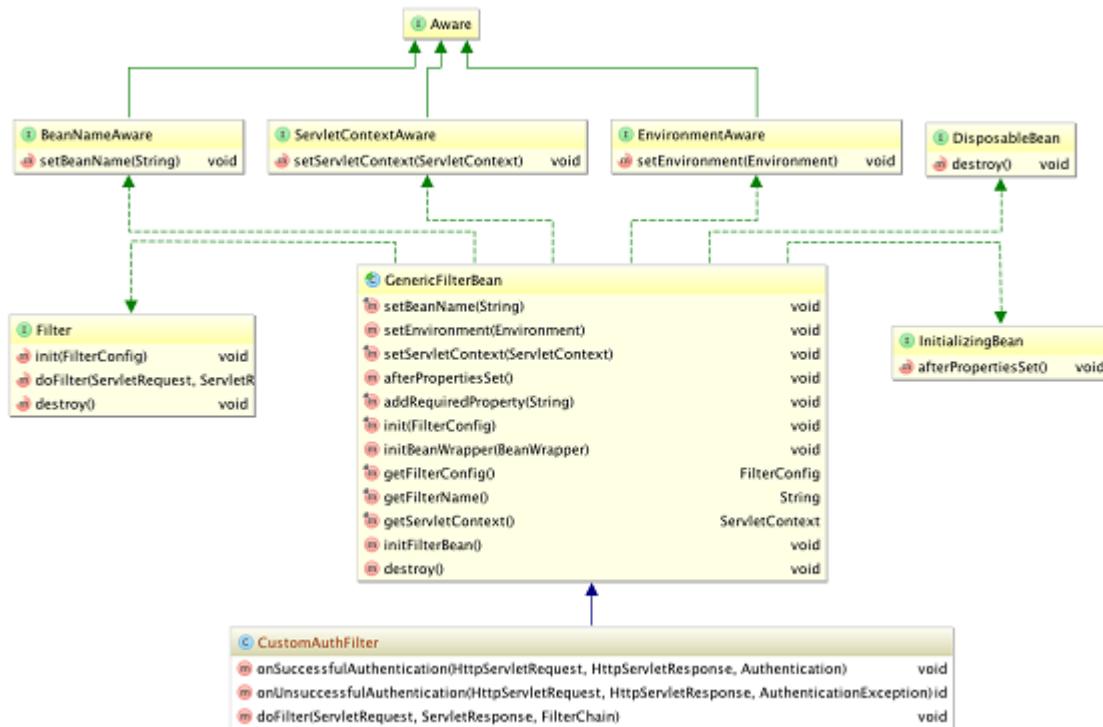
public class CustomAuthenticationProvider extends AbstractClientTokenAuthProvider
                                           implements AuthenticationProvider {
    ...

    @Override
    public Authentication authenticate(Authentication authentication)
        throws AuthenticationException {
        ...
        try {
            ...
            AbstractAuthenticationToken authorizedToken = new UsernamePasswordAuthenticationToken(
                authentication.getPrincipal(),
                authentication.getCredentials(),
                grantedAuthorities);
            if (clientTokenSupport) {
                authorizedToken = produceRestTicketToken(authorizedToken, false, session);
            }
            authorizedToken.setDetails(authentication.getDetails());
            if (!authorizedToken.isAuthenticated()) {
                authorizedToken.setAuthenticated(true);
            }
            return authorizedToken;
        } catch (DfException e) {
            ...
        } finally {
            ...
        }
    }
}

```

Authentication Filter

To create a custom authentication filter, you can extend from an existing third party security filter, or you can implement your custom filter by using an abstract class or interface such as *javax.servlet.Filter*. Here's a diagram of the filter UML:



In the previous section we said that the authentication provider calls the DFC session manager to login. The session is actually released immediately after the login success. It is important that your filter implementation have the logic to store and clear the security context. The security context is used by the resource controller to obtain the DFC session manager in the following cases:

- When an authentication succeeds, the authentication filter must store the user credential into *RepositoryContextHolder*.
- When an authentication fails, the authentication filter must clean up the *RepositoryContextHolder*.

Here's a code sample that shows you how to implement a custom security filter:

Example 8-7. Custom Security Filter

```
// authenticate the servlet request
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    if (!requireHandle(request)) {
        chain.doFilter(request, response);
        return;
    }

    try {
        AuthenticationToken authToken = parseAuthToken(request);
        Authentication authResult = authenticationManager.authenticate(authToken);
        SecurityContextHolder.getContext().setAuthentication(authResult);
        onSuccessfulAuthentication(request, response, authResult);
    }
    catch (AuthenticationException failed) {
        SecurityContextHolder.clearContext();
        onUnsuccessfulAuthentication(request, response, failed);
    }
}
```

```

        if (ignoreFailure) {
            chain.doFilter(request, response);
        }
        else {
            authenticationEntryPoint.commence(request, response, failed);
        }
        return;
    }

    chain.doFilter(request, response);
}

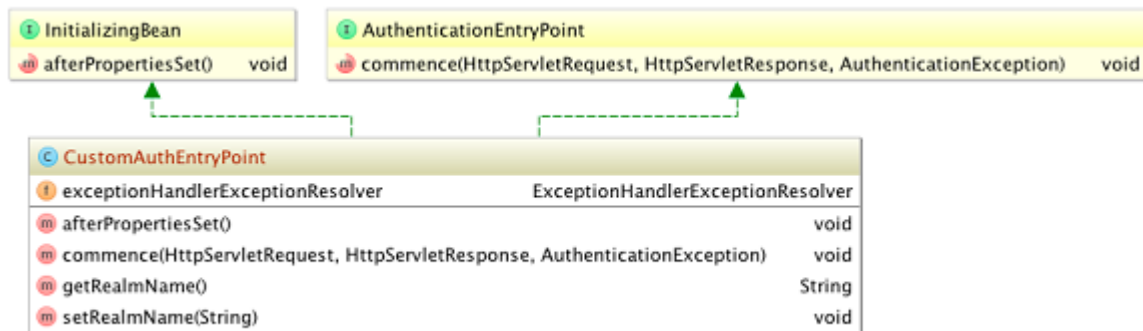
// success handler
protected void onSuccessfullAuthentication(HttpServletRequest request,
    HttpServletResponse response,
    Authentication authResult) throws IOException {
    RepositoryContextHolder.setLoginName(authResult.getPrincipal().toString());
    RepositoryContextHolder.setPassword(authResult.getCredentials());
    RepositoryContextHolder.setAuthType(AuthType.PASSWORD);
}

// failure handler
protected void onUnsuccessfulAuthentication(HttpServletRequest request,
    HttpServletResponse response,
    AuthenticationException failed) throws IOException {
    RepositoryContextHolder.clearUserCredentials();
}

```

Authentication Entry Point

Authentication errors, such as HTTP status 401 for error or status 302 for redirecting, must be returned to the client in the correct manner. Using an entry point helps to implement this logic.



An entry point must at least implement the *Commence* method, and use that method to set the Response Header and status, as well as handle the Response body.

Here is a code sample that shows you a custom authentication entry point:

Example 8-8. Custom Entry Point

```

@Override
public void commence(HttpServletRequest request, HttpServletResponse response,
    AuthenticationException authException)
    throws IOException, ServletException {
    response.setStatus(HttpStatus.UNAUTHORIZED);
    response.addHeader(ChallengeConfig.WWW_AUTHENTICATE, getChallenge(request,
        AuthSchemes.BASIC.value())); exceptionHandlerExceptionHandlerResolver
}

```

```
        .resolveException(request, response, null, authException);  
    }
```

Client Token Integration

The custom authentication scheme can integrate with a Documentum Client Token in a scenario where a successful custom authentication login results in a Documentum Client Token cookie being sent back in the Response.

After receiving the Response, the REST client can access Documentum Platform REST Services by using the Client Token cookie for subsequent requests. This approach avoids having to go through the custom authentication scheme to authenticate the REST client every time. When a custom authentication is used with a Documentum Client Token cookie, a log out must be set up. This can be achieved by putting Client Token filters and a custom authentication filter into an ordered filter chain. Below are the instructions for the filter sequence:

others

-

ClientTokenPreAuthFilter

-

Custom Authentication Filter for URL pattern /repositories/**

-

ClientTokenPostAuthFilter

-

others

- **ClientTokenPreAuthFilter** is a Core REST security filter that validates the Client Token from the Request cookie Header. It must filter requests prior to the other two filters.
- **Custom authentication filter** is the developer implemented authentication filter that authenticates the token from the Request and stores a *ClientTokenAuthenticationToken* into Spring *SecurityContextHolder*. The custom authentication filter must use a provider implementation of *AbstractClientTokenAuthProvider* to produce the *ClientTokenAuthenticationToken* after login.
- **ClientTokenPostAuthFilter** is the other Core REST security filter that sends back the Client Token cookie and Headers for the Response after the custom login succeeds or the Client Token must be refreshed.

Spring Security Java Configuration

Spring Security provides [Java Configuration](#) and [Security Namespace \(XML\) Configuration](#) to configure the security filters for the web application. Both approaches provide the same capabilities for you to configure Spring security. Documentum Platform REST Services supports both configurations, but Java configuration is preferred.

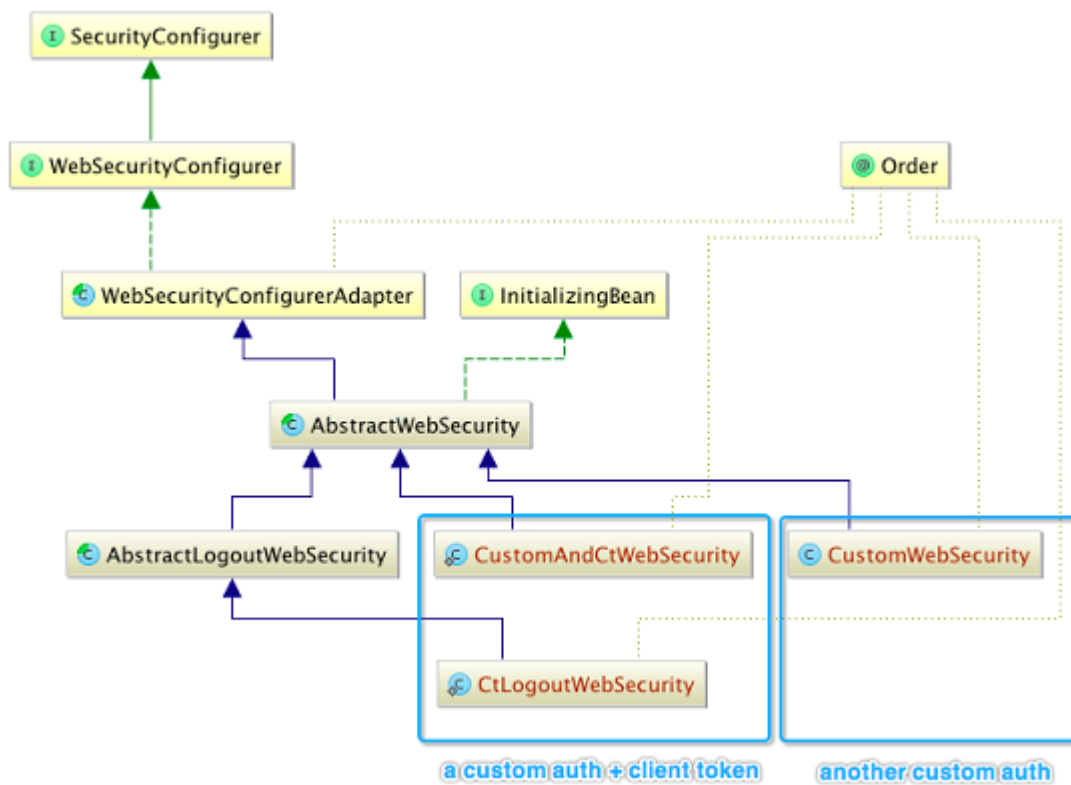
In this section, we introduce the optimized Java configuration for developing new authentication schemes.

Implement Web Security

For Spring Security Java configuration, you typically implement a specific authentication scheme by extending a class from *WebSecurityConfigurerAdapter* and annotating it with *@EnableWebSecurity*.

You can do the same when implementing a Documentum custom authentication. To ease the effort required for development, Documentum Platform REST Services provides an optimized abstraction for you to develop the custom web security in the *documentum-rest-security-core.jar* file.

The following diagram shows the flow of web security integration.



AbstractWebSecurity

Core REST API *AbstractWebSecurity* extends from Spring *WebSecurityConfigurerAdapter* and implements the default configuration and leaves three methods for the implementation to override.

Note: There are two annotations applied to the custom web security: *@AuthSchemeProfile* and *@Order*.

Example 8-9. A Custom Web Security Sample

```
/**
 * Custom authentication
 */
@AuthSchemeProfile(schemes = CustomWebSecurity.MODE)
@Order(1)
public class CustomWebSecurity extends AbstractWebSecurity {

    public static final String MODE = "custom";

    @Override
    protected AuthenticationProvider[] authenticationProviders() {
        return new AuthenticationProvider[] { customAuthProvider() };
    }

    @Override
    protected AuthenticationEntryPoint entryPoint() {
        return customAuthEntryPoint();
    }

    @Override
    protected void configureSecurityFilters(HttpSecurity http) throws Exception {
        http.addFilterBefore(customAuthFilter(), BasicAuthenticationFilter.class);
    }

    @Bean
    public CustomAuthFilter customAuthFilter() {
        return new CustomAuthFilter(authenticationManagerBean());
    }

    @Bean
    public CustomAuthProvider customAuthProvider() {
        return CustomAuthProvider();
    }

    @Bean
    public CustomAuthEntryPoint customAuthEntryPoint() {
        return CustomAuthEntryPoint();
    }
}
```

@AuthSchemeProfile

This is a Core REST annotation to name your custom authentication implementation. It is used, in conjunction with the Spring annotations *@EnableWebSecurity*, *@Configuration*, and *@Conditional*, to make this class work as a Spring security configuration. When the annotation is named, you can enable the authentication scheme at runtime by using REST runtime properties. Here is a code sample that shows you how to do this:

```
@AuthSchemeProfile("custom") --> rest.security.auth.mode=custom
```

@Order

Multiple configurations can be enabled for an authentication scheme. When multiple configurations are enabled for an authentication scheme, the annotation *@Order* specifies the order or priority in which the configurations work relative to each other. The lower value, the higher priority. The number range for *@Order* is from 1 to 98 for a custom configuration. All other numbers are reserved.

Client Token Integration

For client token integration, a web security implementation must create multiple inner classes. One of those inner classes is used to configure the HTTP logout security. Here is a code sample that shows you how to do this:

Example 8-10. Custom Web Security Sample With Client Token Integration

```
/**
 * Custom + client token authentication.
 */
@AuthSchemeProfile(schemes = CustomCtCtWebSecurity.MODE)
public class CustomCtCtWebSecurity {

    public static final String MODE = "custom-ct";

    @AuthSchemeProfile(schemes = CustomCtCtWebSecurity.MODE)
    @Order(1)
    public static class CtLogoutWebSecurity extends AbstractLogoutWebSecurity {
    }

    @AuthSchemeProfile(schemes = CustomCtCtWebSecurity.MODE)
    @Order(2)
    public static class CustomAndCtWebSecurity extends AbstractWebSecurity {

        @Autowired
        protected CtAuthBeans ctBeans;

        @Override
        protected AuthenticationProvider[] authenticationProviders() {
            return new AuthenticationProvider[] {
                customAuthProviderWithClientToken(), ctBeans.clientTokenAuthProvider()
            };
        }

        @Override
        protected AuthenticationEntryPoint entryPoint() {
            return customEntryPoint();
        }

        @Override
        protected void configureSecurityFilters(HttpSecurity http) throws Exception {
            http
                .addFilterBefore(customAuthFilter(), BasicAuthenticationFilter.class)
                .addFilterAfter(clientTokenPreAuthFilter(), BasicAuthenticationFilter.class)
                .addFilterAfter(clientTokenPostAuthFilter(), ClientTokenPreAuthFilter.class);
        }

        @Bean
        public CustomAuthFilter customAuthFilter() {
            return new CustomAuthFilter(authenticationManagerBean());
        }

        @Bean
        public CustomAuthProvider customAuthProvider() {
            return CustomAuthProvider();
        }

        @Bean
        public CustomAuthEntryPoint customAuthEntryPoint() {
            return CustomAuthEntryPoint();
        }
    }
}
```

```

@Bean
public ClientTokenPreAuthFilter clientTokenPreAuthFilter() {
    return ctBeans.clientTokenPreAuthFilter(authenticationManagerBean());
}

@Bean
public ClientTokenPostAuthFilter clientTokenPostAuthFilter() {
    return ctBeans.clientTokenPostAuthFilter();
}
}
}

```

- The first inner class *CtLogoutWebSecurity* extends from *AbstractLogoutWebSecurity* and does not need any customization. This configuration handles Client Token logout. It is annotated with `@AuthSchemeProfile("ct-custom")` and `@Order(1)`.
- The second inner class *CustomAndCtWebSecurity* extends from *AbstractWebSecurity* and performs customizations similar to *CustomWebSecurity*. This configuration performs custom login with Client Token integration. The difference is it must add the Client Token filters into its override method `configureSecurityFilters(HttpSecurity http)`.

It is also annotated with the `@AuthSchemeProfile("ct-custom")` annotation, but has a lower prioritized order of `@Order(2)`.

Note: Core REST API *CtAuthBeans* provides Client Token authentication beans for you to import in the custom web security configuration.



Caution: For the client token integration, when the custom authentication filter wants to send a redirect URI to the Response after the successful authentication, it must not call the Response redirect directly. Instead, it must pass the redirected URL to the authenticated token and let the next filter, the *ClientTokenPostAuthFilter* filter, redirect the URI. Otherwise, the Response won't set the Client Token cookie correctly. Here is a code sample that shows you how to do this:

```

// Delegate to Core ClientTokenPostAuthFilter to do redirect
ClientTokenAuthToken authenticated = (ClientTokenAuthToken)
    getAuthenticationManager().authenticate(token);
if (authenticated.isAuthenticated()) {
    authenticated.setRedirectUri(redirectUrl);
    SecurityContextHolder.getContext().setAuthentication(authenticated);
}

```

Externalize Configuration Parameters

In a custom authentication configuration, some variables must be externalized so that they are only specified at runtime, not compile time. You can implement a runtime property class that imports variables from *rest-api-runtime.properties*. This class can be accessed by your implementation classes to import external runtime variables.

Example 8-11. Runtime Properties Sample

```

/**
 * Default custom security runtime properties values.
 * Properties set in 'rest-api-runtime.properties' will override the default.
 */
@Configuration
@PropertySource("classpath:rest-api-runtime.properties")
public class CustomRuntime {

```



```

/**
 * A static place holder to load properties from the file 'rest-api-runtime.properties'.
 *
 * @return placeholder configurer
 */
@Bean
public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer() {
    return new PropertySourcesPlaceholderConfigurer();
}

/**
 * Property value for 'rest.security.custom.key'. Defaults to empty.
 */
@Value("${rest.security.custom.key}")
public String key;

/**
 * Property value for 'rest.security.custom.max_age'. Defaults to 3600.
 */
@Value("${rest.security.custom.max_age:3600}")
public Integer maxAge;
}

```

Configure Runtime Properties

In the custom REST war deployment, you can enable the custom authentication scheme and configure each parameter through *rest-api-runtime.properties*.

Example 8-12. Runtime Properties Sample for Custom Authentication

```

set authentication mode (required)
rest.security.auth.mode=ct-custom

# set realm name
rest.security.realm.name=ACME.COM

# set Java configuration package (required)
rest.context.config.location=com.acme.rest.security.config

# set other parameters
rest.security.custom.max_age=60
rest.security.custom.key=A42gHx47X

```

Note: The runtime property *rest.context.config.location* requires you to specify the Java package of the configuration classes that extend from *AbstractWebSecurity* or *AbstractLogoutWebSecurity*. It is required for Java configuration.

Spring Security XML Configuration

For developers that are more familiar with XML configuration, it is possible to configure the security in XML. This is not the recommended solution because additional security extensions in Documentum Platform REST Services are based on Java configurations.

For XML configuration, you write the XML file to configure HTTP security for Documentum custom authentication. The XML file must be packaged with the `jar` file. The content of the XML configuration is a Spring `<http>` configuration and relevant bean definitions.

For more information on how to configure a custom filter, see the *Spring Security Reference* called [Adding in Your Own Filters](#).

Write XML Configuration

Here is a code sample that show the `<http>` configuration for the custom authentication sample that is the same as the Java configuration *CustomWebSecurity*. It is assumed that the file name is `rest-api-ct-custom-security.xml`.

Example 8-13. Custom Authentication XML Configuration Sample 1

```
<!-- import properties -->
<beans:bean id="propertyConfigurer" class="org.springframework.beans.factory.config
    .PropertyPlaceholderConfigurer">
    <beans:property name="locations">
        <beans:list>
            <beans:value>classpath:rest-api-runtime.properties</beans:value>
        </beans:list>
    </beans:property>
</beans:bean>

<!-- HTTP security configuration: custom basic authentication -->
<http use-expressions="true" create-session="never"
    entry-point-ref="customAuthEntryPoint">
    <custom-filter ref="customAuthFilter" before="BASIC_AUTH_FILTER"/>
    <intercept-url pattern="/repositories/**" access="hasRole('ROLE_USER')"/>
    <csrf disabled="true"/>
</http>

<!-- filter definition: customized HTTP authentication filter -->
<beans:bean id="customAuthFilter" class="com.acme.security.filter.CustomAuthFilter">
    <beans:constructor-arg ref="customAuthManager"/>
</beans:bean>

<!-- authentication manager definition: custom authentication providers -->
<authentication-manager alias="customAuthManager" erase-credentials="false">
    <authentication-provider ref="customAuthProvider"/>
</authentication-manager>

<!-- provider definition -->
<beans:bean id="customAuthProvider" class="com.acme.security.provider.CustomAuthProvider"/>

<!-- entry point definition -->
<beans:bean id="customEntryPoint" class="com.acme.security.entry.CustomAuthEntryPoint">
    <beans:property name="realmName" value="{rest.security.realm.name}"/>
</beans:bean>
```

Example 8-14. Custom Authentication XML Configuration Sample 2

Here is an XML code sample of the `<http>` configuration for the custom authentication that is the same as the Java configuration *CustomCtWebSecurity*.

```
<!-- import properties -->
<beans:bean id="propertyConfigurer" class="org.springframework.beans.factory.config
    .PropertyPlaceholderConfigurer">
    <beans:property name="locations">
```

```

        <beans:list>
            <beans:value>classpath:rest-api-runtime.properties</beans:value>
        </beans:list>
    </beans:property>
</beans:bean>

<import resource="classpath*:META-INF/spring/template/rest-api-ct-security"/>

<!-- HTTP security configuration: ct sign out -->
<http use-expressions="true" create-session="never" entry-point-ref="basicEntryPoint"
    pattern="/logout">
    <custom-filter ref="dctmClientTokenSignOutFilter" position="LOGOUT_FILTER"/>
    <intercept-url pattern="/logout" access="ROLE_USER"/>
    <csrf disabled="true"/>
</http>
<!-- HTTP security configuration: custom basic authentication -->
<http use-expressions="true" create-session="never"
    entry-point-ref="customAuthEntryPoint">
    <custom-filter ref="customAuthFilter" before="BASIC_AUTH_FILTER"/>
    <custom-filter ref="clientPreTokenFilter" position="BASIC_AUTH_FILTER"/>
    <custom-filter ref="clientPostTokenFilter" after="BASIC_AUTH_FILTER"/>
    <intercept-url pattern="/repositories/**" access="hasRole('ROLE_USER')"/>
    <csrf disabled="true"/>
</http>

<!-- filter definition: customized HTTP authentication filter -->
<beans:bean id="customAuthFilter" class="com.acme.security.filter.CustomAuthFilter">
    <beans:constructor-arg ref="customAuthManager"/>
</beans:bean>

<!-- authentication manager definition: custom authentication providers -->
<authentication-manager alias="customAuthManager" erase-credentials="false">
    <authentication-provider ref="customAuthProvider"/>
    <authentication-provider ref="clientTokenAuthProvider"/>
</authentication-manager>

<!-- filter definition -->
<beans:bean id="customAuthFilter" class="com.acme.security.provider.CustomAuthFilter">
    <beans:constructor-arg index="0" name="authenticationManager"
        ref="customAuthManager"/>
</beans>

<!-- provider definition -->
<beans:bean id="customAuthProvider" class="com.acme.security.provider
    .CustomClientTokenAuthProvider">
    <beans:property name="clientTokenSupport" value="true"/>
</beans>

<!-- entry point definition -->
<beans:bean id="customEntryPoint" class="com.acme.security.entry.CustomAuthEntryPoint">
    <beans:property name="realmName" value="{rest.security.realm.name}"/>
</beans:bean>
<!-- filter definition: Documentum REST client token authentication filters -->
<beans:bean id="clientTokenPreFilter" class="com.emc.documentum.rest.security.filter
    .ClientTokenPreAuthFilter">
    <beans:property name="authenticationManager" ref="customAuthManager"/>
</beans:bean>
<beans:bean id="clientTokenPostFilter" class="com.emc.documentum.rest.security.filter
    .ClientTokenPostAuthFilter"/>

<!-- filter definition: customized Client Token sign out filter -->
<beans:bean name="dctmClientTokenSignOutFilter"
    class="com.emc.documentum.rest.security.filter.ClientTokenSignOutFilter">
    <beans:constructor-arg index="0" name="logoutSuccessUrl"
        value="{rest.security.logout.success.url}"/>

```

```
</beans:bean>
```

Note: The `rest-api-ct-security.xml` file is imported into the XML file. The `rest-api-ct-security.xml` file provides beans for Client Token authentication, and it is the same as the `CtAuthBeans` Java configuration class

Configure Security XML Extension

To register your custom authentication configuration XML file into the REST services, you must update `${war}/META-INF/spring/extension/rest-api-custom-security.xml`.

Example 8-15. Modify Security Extension for Custom Authentication

```
<beans:beans>
  <!-- ===== -->
  <!-- CAUTION: XML CONFIGURATION IS DISCOURAGED.
        USE JAVA CONFIGURATION INSTEAD. -->
  <!-- ===== -->

  <!-- Legacy XML configuration for adding custom authentication beans -->
  <!-- Adds a custom authentication XML configuration in two steps: -->

  <!-- Step 1: Import system default anonymous URL patterns -->
  <beans:import resource="rest-api-anonymous-security"/>

  <!-- Step 2: Import oauth2 XML configuration file -->
  <beans:import resource="classpath*:rest-api-ct-custom-security"/>
</beans:beans>
```

Configure Runtime Properties

Similar to the Java configuration, in the custom REST war deployment, you can enable the custom authentication scheme and configure each parameters through `rest-api-runtime.properties`.

Example 8-16. Modify Security Extension for Custom Authentication

```
# set authentication mode (required)
rest.security.auth.mode=ct-custom

# set realm name
rest.security.realm.name=ACME.COM

# set other parameters
rest.security.custom.max_age=60
rest.security.custom.key=A42gHx47X
```

Packaging Artifacts

The packaging process produces the custom authentication `jar` file that includes all implemented classes and Spring XML configurations. You can package the `jarfile` into the Documentum REST war file manually or by using the war overlay of Documentum REST Archetype project.

Samples

The Documentum REST SDK provides two samples that show you how to build custom authentications. One sample is for a custom login form with Client Token, the other sample is for an OAuth2 login.

Please refer to the *Documentum REST SDK* for the sample code.

Tutorial: Documentum REST Authentication Extensibility Development

In this section, we discuss how to setup a Maven project to develop a custom authentication scheme. It is recommended that you create the custom authentication project based on the Documentum REST Archetype.

For more information, see [Get Started With the Development Kit](#).

The project structure for the extended authentication scheme is shown here. In the following sample we give you the Java configuration, and we name the new authentication scheme *ct-custom*.

For XML configuration samples, see the Documentum REST SDK.

Example 8-17. Maven Project Layout

```
acme-rest (from archetype)
├── acme-rest-custom-security (new)
│   ├── src
│   │   ├── main
│   │   │   ├── java
│   │   │   │   ├── com
│   │   │   │   │   ├── acme
│   │   │   │   │   │   ├── security
│   │   │   │   │   │   │   ├── config
│   │   │   │   │   │   │   │   ├── CustomRuntime.java
│   │   │   │   │   │   │   │   ├── CustomWebSecurity.java
│   │   │   │   │   │   │   ├── entry
│   │   │   │   │   │   │   │   ├── CustomEntryPoint.java
│   │   │   │   │   │   │   ├── filter
│   │   │   │   │   │   │   │   ├── CustomAuthFilter.java
│   │   │   │   │   │   │   ├── provider
│   │   │   │   │   │   │   │   ├── CustomClientTokenAuthProvider.java
│   │   │   │   │   │   ├── resources
│   │   │   │   │   │   ├── test
│   │   │   │   │   │   │   ├── java
│   │   │   │   │   │   │   ├── resources
│   │   │   │   │   ├── pom
│   │   ├── acme-rest-web (from archetype)
│   │   │   ├── src
│   │   │   │   ├── main
│   │   │   │   │   ├── resources
│   │   │   │   │   │   ├── rest-api-runtime.properties
│   │   │   │   ├── pom
│   │   ├── pom
│   ├── pom
acme-rest-custom-security
```

This is a new Maven sub module that is added to the Archetype project. It includes all the Java code and configurations for the custom authentication scheme. The build output of the module is a jar file.

CustomRuntime.java

Custom runtime allows you to read custom runtime properties in Java configuration.

```
/**
 * Default custom security runtime properties values.
 * Properties set in 'rest-api-runtime.properties' will override the default.
 */
@Configuration
@PropertySource("classpath:rest-api-runtime.properties")
public class CustomRuntime {

    /**
     * A static place holder to load properties from the file 'rest-api-runtime.properties'.
     *
     * @return placeholder configurer
     */
    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    /**
     * Property value for 'rest.security.custom.key'. Defaults to empty.
     */
    @Value("${rest.security.custom.key:}")
    public String key;
}
```

CustomWebSecurity.java

Please follow [Java Configuration](#) to implement the custom Web Security. The sample code *CustomWebSecurity* shows the logic to implement multiple HTTP security for an authentication scheme.

CustomEntryPoint.java

Please follow the instructions in [Authentication Entry Point, page 331](#) to write the *CustomEntryPoint* code. HTTP status 401 and 302 are the most often used codes to indicate an authentication failure.

- According To [RFC 7235](#), when an HTTP status 401 is used, a Response Header *WWW-Authenticate <challenge>* should also be returned.

The authentication exception must be transformed into an XML or JSON error. You can access Spring resolver `org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver` into the *CustomEntryPoint* class to resolve the authentication exception in the servlet response.

For more information, see the sample code at [Authentication Entry Point, page 331](#).

- When an HTTP status 302 is used to indicate a user login failure, the user must be redirected to a login page or the other authentication service. Here is a code sample that shows HTTP redirection:

Example 8-18. HTTP Redirection on Login Failure

```
@Override
public void commence(HttpServletRequest request, HttpServletResponse response,
    AuthenticationException authException)
    throws IOException, ServletException {
```

```

        if (response.isCommitted()) {
            return;
        }
        // use Spring redirect strategy
        new DefaultRedirectStrategy().sendRedirect(request, response, "http://acme.com/sign-on");
    }
}

```

CustomAuthFilter.java

Please follow [Authentication Filter, page 329](#) to write the *CustomAuthFilter* code.

CustomClientTokenAuthProvider.java

Please follow the instructions at [Authentication Provider, page 327](#) to write the *CustomClientTokenAuthProvider* code.

An authentication provider does two jobs:

- Authenticate the user with DFC session manager
- Produce and return an authenticated token

To view a code sample, see [Authentication Provider, page 327](#).

Client Token Support

When a custom authentication scheme is used with a Documentum Client Token cookie, the provider must return the authenticated token as a *ClientTokenAuthToken* token.

Example 8-19. Example of Custom Authentication Provider

```

@Override
public Authentication authenticate(Authentication authentication)
    throws AuthenticationException {
    if (authentication == null || Strings.isNullOrEmpty(authentication.getName())) {
        throw new UsernameNotFoundException(MessageBundle
            .INSTANCE.get("E_BAD_CREDENTIALS_ERROR"));
    }

    String repositoryId = RepositoryContextHolder.getRepositoryName();
    String user = (String) authentication.getPrincipal();
    String password = (String) authentication.getCredentials();
    IDfSessionManager sm = null;
    IDfSession session = null;
    try {
        sm = memoryRepositorySessionManager.get(repositoryId, user, password,
            AuthType.PASSWORD);

        session = getSession(sm, repositoryId);
        Collection<GrantedAuthority>
            grantedAuthorities = new ArrayList<GrantedAuthority>();
        grantedAuthorities.add(new SimpleGrantedAuthority("ROLE_USER"));
        AbstractAuthenticationToken authorizedToken =
            new UsernamePasswordAuthenticationToken(
                authentication.getPrincipal(),
                authentication.getCredentials(),
                grantedAuthorities);
        ///// CHANGE CODE BELOW
        // ClientTokenAuthToken is produced
        if (clientTokenSupport) {
            authorizedToken = produceRestTicketToken(authorizedToken, false, session);
        }
        ///// ADD CODE ABOVE
        if (!authorizedToken.isAuthenticated()) {
            authorizedToken.setAuthenticated(true);
        }
    } catch (Exception e) {
        // ...
    }
}

```

```
        }
        return authorizedToken;
    } catch (DfException e) {
        clearCache(repositoryId, user, password, AuthType.PASSWORD);
        throw getWrappedException(sm, repositoryId, e);
    } finally {
        DfcSessions.release(session);
    }
}
```

pom.xml (custom-security)

The custom authentication scheme module must add dependencies on both Core REST Security and Spring Security. Here is a code sample that shows you the minimal dependencies.

Example 8-20. Custom Authentication Scheme Pom

```
<dependency>
  <groupId>com.emc.documentum.rest</groupId>
  <artifactId>documentum-rest-config</artifactId>
</dependency>
<dependency>
  <groupId>com.emc.documentum.rest</groupId>
  <artifactId>documentum-rest-security-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <scope>provided</scope>
</dependency>
```

acme-rest-web

It is the existing Maven sub module in the Archetype project. Its purpose is to overwrite the WAR packaging and settings in the extended REST Services. So after resources files are put into the module location, the files overwrite the default files in the same location within the WAR file.

rest-api-runtime.properties

Please follow the instructions in [Configure Runtime Properties, page 337](#) to set the runtime properties for the custom authentication scheme. During the build of the web module, this file overwrites the default (empty) REST Runtime Properties file in the WAR file.

Example 8-21. Runtime Properties for Custom Authentication

```
# set authentication mode (required)
rest.security.auth.mode=ct-custom

# set realm name
rest.security.realm.name=ACME.COM
```



```
# set Java configuration package (required)
rest.context.config.location=com.acme.rest.security.config
```

pom.xml (web)

The web module must add dependency for the module `acme-rest-custom-security` and the custom authentication jar file packages everything into the WAR file. Here is a code sample that shows you the pom file:

Example 8-22. Web Module Pom

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>acme-rest</artifactId>
    <groupId>com.acme.rest</groupId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>acme-rest-web</artifactId>
  <packaging>war</packaging>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <includes>
          <include>*.properties</include>
        </includes>
        <filtering>true</filtering>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>${org.apache.maven.plugins.war.version}</version>
        <configuration>
          <overlays>
            <overlay>
              <groupId>com.emc.documentum.rest</groupId>
              <artifactId>documentum-rest-web</artifactId>
              <excludes />
            </overlay>
          </overlays>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>com.acme.rest</groupId>
      <artifactId>acme-rest-custom-security</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>com.emc.documentum.rest</groupId>
      <artifactId>documentum-rest-web</artifactId>
      <version>${com.emc.documentum.rest.version}</version>
      <type>war</type>
```

```
        <scope>runtime</scope>
      </dependency>
      <!-- other new dependencies -->
    </dependencies>
  </project>
```

pom.xml (root)

This is the Archetype project's root module. It must add `acme-rest-custom-security` as a sub module. Here is a code sample that shows you the root pom file:

Example 8-23. The Root Pom

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>acme-rest</artifactId>
    <groupId>com.acme.rest</groupId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <packaging>pom</packaging>
  <modules>
    <module>acme-rest-custom-security</module>
    <!-- other sub modules -->
  </modules>
</project>
```

Advanced Security Configuration

This section contains configurations for advanced security configuration items and features, such as *Default Security Headers*, *Cross Site Request Forgery (CSRF) protection with Client token*, *Request Sanitization*, and *Cross-Origin Resource Sharing* or (CORS).

Default Security Headers

Documentum Platform REST Services version 7.3 and later provides the following configurable security defaults to help protect your REST services:

Vulnerability	Solution	Runtime Properties	Defaults
Strict Transport Security Misconfiguration	Add the Strict Transport Security Response Header	<p>Specifies whether to disable the Strict-Transport-Security header. When false, the HTTP Strict Transport Security (HSTS) is enabled when the network protocol is HTTPS and the response adds the following header: - Strict-Transport-Security: max-age=31536000 ; includeSubDomains For more information, see Tools.ietf.org. Defaults to false. rest.security.headers.hsts.disabled=</p> <p>Specifies whether to include sub domains for the Strict-Transport-Security header. When true, sub domains should be considered HSTS Hosts. For more information, see Tools.ietf.org for defaults to true.</p>	<pre>Strict -Transport -Security: max-age= 31536000 ; includeSub Domains</pre>

Vulnerability	Solution	Runtime Properties	Defaults
		rest.security.headers .hsts.include_sub_domains= Specifies the value (in seconds) for the max-age directive of the Strict-Transport-Security header. For more information, see Tools.ietf.org Defaults to one year. rest.security.headers.hsts .max_age_in_seconds=	
Cacheable HTTPS Response	Add Cache-Control , Pragma , Expires header	Specifies whether to disable the Cache-Control header. When false, the response adds the following headers: - Cache-Control: no-cache, no-store, max-age=0, must-revalidate - Pragma: no-cache - Expires: 0 Defaults to false. rest.security.headers.cache_control.disabled=	Cache-Control: no-cache, no-store, max-age=0, must-revalidate Pragma: no-cache Expires: 0
Content Sniffing Not Disabled	Add Content-Type-Options header	Specifies whether to disable the X-Content-Type-Options header. When false, the response adds the following header: - X-Content-Type-Options: nosniff For more information, see MSDN Defaults to false. rest.security.headers.content_type_options.disabled=	X-Content-Type-Options: nosniff
Missing Cross-Frame Scripting Protection	Add X-Frame-Options header	Specifies whether to disable the X-Frame-Options header. When false, the response adds the following headers: - X-Frame-Options: DENY For more information, see Mozilla Development Defaults to false. rest.security.headers.x_frame_options.disabled= Specifies the X-Frames-Options policy. Allowed values are DENY and SAMEORIGIN. Defaults to	X-Frame-Options: DENY

Vulnerability	Solution	Runtime Properties	Defaults
		<code>DENY. rest.security.headers .x_frame_options.policy=</code>	
Browser Cross-Site Scripting Filter Misconfiguration	Add X-XSS-Protection header	<p>Specifies whether to disable the X-XSS-Protection header. When false, the response adds the following header: - X-XSS-Protection: 1; mode=block For more information, see MSDN Blog Defaults to false.</p> <p><code>rest.security.headers.xss_protection.disabled=</code></p> <p>Specifies whether to enable X-XSS-Protection explicitly. When true, the Response adds the following header: - X-XSS-Protection: 1 When false, the Response adds the following header: - X-XSS-Protection: 0 Defaults to true.</p> <p><code>rest.security.headers.xss_protection.explicit_enable=</code></p> <p>Specifies whether to set block mode for the X-XSS-Protection header. When true, the response adds the block mode to the X-XSS-Protection header: - X-XSS-Protection: 1; mode=block When false, the response does not add a mode. Defaults to true.</p> <p><code>rest.security.headers.xss_protection.block=</code></p>	X-XSS- -Protection: 1; mode= block

CSRF Protection

Cross Site Request Forgery (CSRF) is an attack by a malicious Web site, email, blog, instant message, or application. The attacker uses a visitor's Web browser and authentication credentials to perform an unwanted action on a trusted site that the user is visiting.

Documentum Platform REST Services supports CSRF protection using a Client Token cookie for ticked sign on, and this functionality can be integrated with CAS, Kerberos, or a custom SSO.

The CSRF token is used to protect the Documentum Client Token cookie. The client must provide the token in a request, and the server validates the token provided. When validation fails, the server returns a response with a 403 (Permission Denied) HTTP status code. This can still occur even when the initial authentication succeeds.

Enabling CSRF Protection

When CSRF protection is enabled, the server requires the Client to send a Client Token cookie together with the CSRF token to validate a user login. Since a CSRF attack is considered to be a severe security issue, Documentum Platform REST Services since 7.3 enables CSRF by default. This means that a REST client that authenticates with Client Token cookies against older REST servers does not work against a new REST server, unless the REST client implements CSRF protection. Documentum Platform REST Services runtime properties provide a configuration parameter that you can use to enable or disable CSRF protection. Here a code sample showing you how to disable CSRF protection in the `rest-api-runtime.properties` file:

```
// Disable CSRF protection for Client Token cookie.  
// Its default value is true.  
rest.security.csrf.enabled=false
```

Generating a Token

Documentum Platform REST Services uses the *Synchronizer Token Pattern* to protect the Client Token cookie. A CSRF token must be generated either by the Client or by the Server. Documentum Platform REST Services runtime properties provides two ways to generate a token.

1. **Using the Client** - In authentication schemes that use a Client Token cookie, the Client sends a defined CSRF token and parameter names to the REST server during the initial authentication (Basic, SSO, etc.) phase. The REST server sends back a Client Token cookie that all future requests, that authenticate with Client Token cookie, must use in their Request Header or Request query parameter.
2. **Using the Server** - In the authentication schemes that use a Client Token cookie, when clients pass the initial authentication (Basic, SSO, etc.) phase, the REST server sends back a Client Token cookie together with a CSRF token in the Response Headers. All subsequent Requests that authenticate with a Client Token cookie must provide the CSRF token together with the Client Token cookie, either in the Request Header or in the Request query parameter.

Generating a Client-side Token

Initial Authentication

In client-side CSRF token generation, the client must send a CSRF token and parameters in the initial authentication phase. Here's some code sample that shows a client sign on and server response:

```
// Client sign on  
GET /dctm-rest/repositories/REPO HTTP/1.1
```

```
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZA==
DOCUMENTUM-CSRF-HEADER-NAME: {csrfHeaderName}
DOCUMENTUM-CSRF-QUERY-NAME: {csrfQueryName}
DOCUMENTUM-CSRF-TOKEN: {csrfTokenValue}

// Server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json
Set-Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58="; ...; HttpOnly

//...
```

Note the following items in the Response Headers:

- **DOCUMENTUM-CSRF-HEADER-NAME:** This Header tells the client in which Request Header it should pass to the CSRF token.
- **DOCUMENTUM-CSRF-QUERY-NAME:** This Header tells the client which Request query parameter contains the CSRF token.
- **DOCUMENTUM-CSRF-TOKEN:** This Header specifies the CSRF token and is mandatory.

The Request must send the CSRF token together with the Client Token cookie for authentication. When both Header and query are specified, the client can choose either the Header or the query to carry the CSRF token. More information is available in the Token Validation section below.

Runtime Configuration

Here's a code sample that shows you the token generation method:

```
# Specifies the token generation method; defaults to 'server'
rest.security.csrf.generation.method=client
```

Errors for Incomplete Authentication

When the CSRF token is generated by the client, the initial authentication Request must not provide CSRF token and parameters. When the authentication Request does not provide the CSRF token or one of the CSRF parameter, the server fails to authenticate with the following Response status:

- Bad Request (400)
 - E_LACK_OF_CSRF_TOKEN
 - E_LACK_OF_CSRF_KEY_DEFINITION

Generating a Server-side Token

Initial Authentication

Generating a server-side token is the default CSRF token generation method. The initial authentication request remains the same as before, but the Response has a CSRF token and parameters send back to the client.

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

// Server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json
Set-Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58="; ...; HttpOnly
DOCUMENTUM-CSRF-HEADER-NAME: {csrfHeaderName}
DOCUMENTUM-CSRF-QUERY-NAME: {csrfQueryName}
DOCUMENTUM-CSRF-TOKEN: {csrfTokenValue}

// ...
```

Note the following items in the Response Headers:

- **Set-Cookie:** This is the Documentum Client Token cookie.
- **DOCUMENTUM-CSRF-HEADER-NAME:** This Header tells the client in which Request Header it should pass to the CSRF token.
- **DOCUMENTUM-CSRF-QUERY-NAME:** This Header tells the client which Request query parameter contains the CSRF token.
- **DOCUMENTUM-CSRF-TOKEN:** This Header specifies the CSRF token and is mandatory.

Token Lifetime

When the CSRF token is generated on the server side, the CSRF token and the Client Token cookie are generated at the same time, which means that the CSRF token lifetime is same as the Client Token

lifetime. Each time a new Client Token cookie is sent back to the client, the client must reset the CSRF token by parsing the Response Header.

Runtime Configuration

Documentum Platform REST Services runtime properties provides configurations for server side token generation. Here's a code sample that shows the runtime properties:

```
# Specifies the token generation method; defaults to 'server'
rest.security.csrf.generation.method=

# Specifies {csrfHeaderName} for 'server' token generation; defaults to 'DOCUMENTUM-CSRF-TOKEN'
rest.security.csrf.header_name=

# Specifies {csrfQueryName} for 'server' token generation; defaults to 'csrf-token'
rest.security.csrf.parameter_name=
```

```
# Specifies the 'server' generated CSRF token length in bits; defaults to 256
rest.security.csrf.token.length=
```

The REST server uses the same random algorithm as the Client Token to produce the CSRF token for server-side CSRF token generation. It can be customized using the following runtime property, which affect the Client Token encryption.

```
rest.security.random.algorithm=
```

Token Validation

Here's a code sample that shows the CSRF protection token in the Request Headers:

Example 9-1. A Token Request in a Request Header

```
// Client request with the CSRF protection token in the Request headers. This sample
// assumes that 'DOCUMENTUM-CSRF-TOKEN' is set as the CSRF token header name
GET /dctm-rest/repositories/REPO HTTP/1.1
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58="
DOCUMENTUM-CSRF-TOKEN: {csrfTokenValue}

// Server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json

//...
```

Here's a code sample that shows a client request with CSRF protection in its query parameter:

Example 9-2. A Token Request in a Query Parameter

```
// Client request with the CSRF protection token in the query parameters
// this sample assumes 'csrf-token' is set as the CSRF token query parameter name
GET /dctm-rest/repositories/REPO?csrf-token={urlEncodedCsrfTokenValue} HTTP/1.1
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.3.1 (java 1.5)
```

```
Cookie: DOCUMENTUM-CLIENT-TOKEN="ACa/0u...W58="

// Server response
HTTP/1.1 200 OK
Content-Type: application/vnd.emc.documentum+json

//...
```

Errors for Validation

When the CSRF token is not validated, the REST server sends the HTTP 403 Response status shown here:

- Forbidden (403)
 - E_CSRF_TOKEN_NOT_VALID
 - E_CSRF_TOKEN_NOT_PROVIDED

HTTP Inspection Method

The REST server can determine which HTTP methods (GET, POST, PUT, DELETE) require CSRF token validation. By default, *GET* does not require CSRF token validation. This can be customized using runtime properties as shown in the following code sample:

```
// Used to specify which HTTP methods required CSRF protection; defaults to POST,PUT,DELETE.
rest.security.csrf.http_methods=
```

Cross-Origin Resource Sharing (CORS) Support

The Cross-Origin Resource Sharing (CORS) specification defines a mechanism to enable client-side cross-origin requests. Specifications that enable an API to make cross-origin requests to resources can use the algorithms defined by this specification.

For example, when an API is used on `http://example.org` resources, a resource on `http://hello-world.example` can be made to use the mechanism described by the CORS specification by setting *Access-Control-Allow-Origin: http://example.org* in the Response Header. This mechanism retrieves the resource cross-origin from `http://example.org`.

For more information, see [Cross-Origin Resource Sharing](#).

Enable REST Server CORS Support

You can enable REST server CORS support by configuring the REST API Runtime Properties file. Set the value of the `rest.cors.enabled=true` configuration property to enable CORS support.

Example 9-3. CORS Configurations

```

#Whether or not cross origin resource sharing support is enabled.
#The default value is false.
rest.cors.enabled=false

#The origins that are allowed to share.
#When you want enable an origin to share, set the value to *.
#When more than one origin exists, the value should be seperated by a comma.
#For example, http://emc.com, https://test.com:8443
#The default value *.
rest.cors.allowed.origins=*

#The HTTP methods allowed to be shared.
#The value should be in uppercase, and seperated by a comma.
#The default value is: *
rest.cors.allowed.methods=*

#The HTTP headers allowed to be sent.
#The value should be seperated by a comma.
#The default value is '*', which means that all headers are allowed.
rest.cors.allowed.headers=*

#Whether to allow user credentials
#The default value is true
rest.cors.allow.credentials=true

#The headers that are safe to expose to the API.
#The values are seperated by a comma.
#Simple headers are exposed by the CORS specification, and need not be set:
#Cache-Control, Content-Language, Content-Type, Expires, Last-Modified, Pragma.
#The Location header is added by default.
rest.cors.exposed.headers=Location

#The amount of time (in seconds) that the results of a preflight request
#can be cached.
#The value is in seconds and the default is 3600 (1 hour).
rest.cors.max.age=3600

```

Request Sanitization

Stored Cross-Site Scripting allows for the permanent injection of JavaScript code. This is a security vulnerability because this JavaScript code can result in the theft of user sessions. Request sanitization cleans up user input to secure against this vulnerability.

Request sanitization looks at two parts of the input:

- The input object metadata
- The input HTML content

Third Party Libraries

We used a third party library called OWASP AntiSamy. The OWASP AntiSamy library provides the *java api*, which is used to sanitize an input string. When found, a suspicious script is removed from the input string. The library also provides policies that can be used to configure sanitization.

Sanitize the Input Object Metadata

Here's a sample that shows you how a malicious script might be added while creating a document:

Example 9-4. XML Script Injection Sample

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <properties>
    <object_name>
      The<script>alert("I shouldn't be here!")</script>Document
    </object_name>
    <r_object_type>dm_document</r_object_type>
  </properties>
</document>
```

Example 9-5. JSON Script Injection Sample

```
{
  "name" : "document",
  "type" : "dm_document",
  "properties": {
    "object_name" : "The<script>alert('I shouldn't be here!')</script>Document",
  }
}
```

After sanitization, the metadata becomes:

Example 9-6. XML Sanitized Metadata

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <properties>
    <object_name>TheDocument</object_name>
    <r_object_type>dm_document</r_object_type>
  </properties>
</document>
```

Example 9-7. JSON Sanitized Metadata

```
{
  "name" : "document",
  "type" : "dm_document",
  "properties": {
    "object_name" : "TheDocument",
  }
}
```

The metadata sanitization is combined with the Documentum REST marshalling framework by Java annotations. When the input metadata is custom processed, the sanitization should be called manually.

You can use the `rest.sanitize.type.metadata` configuration setting, which is found in the runtime properties file, to control whether the metadata for all objects is sanitized .

```
# Determines whether to sanitize the metadata of an object.
# The sanitize will modify the metadata (remove the suspicious part).
# The default value is false.
#
rest.sanitize.type.metadata=false
```

When model field have special behavior against the global configuration, the `@SanitizeConfig` annotation can be used in the following two use cases:

- When metadata is supposed to be saved with XML, or a legitimate script, etc. After the metadata has been sanitized the script and other XML elements may be inadvertently removed. To avoid the removal of legitimate content, set the annotation `@SanitizeConfig` with `skipSanitize=true`. This value causes the annotated field to be skipped during the sanitization process while deserializing.

Here's an example that shows you how to use the `@SanitizeConfig` annotation:

```
@SerializableType(value...)
public class MyModel {
    @SerializableField
    @SanitizeConfig(skipSanitize=true)
    private String notSanitized;
    ...
}
```

- When a specific field always requires sanitizing in spite of the global configuration, that field must be annotated with the `@SanitizeConfig` annotation with the value `enforceSanitize=true`. Regardless of the configuration, that field is always sanitized.

Here's an example that shows you how to use this annotation:

```
@SerializableType(value...)
public class MyModel {
    @SerializableField
    @SanitizeConfig(enforceSanitize=true)
    private String mustBeSanitized;
    ...
}
```

Sanitize the Input HTML Content

When performing the following operations, the content may be sanitized according the configurations:

- While importing the content (folder child objects/documents resources)
- While creating the primary content/renditions (contents resource)
- While checking in the content (versions resource)

You can configure whether to sanitize content, and for what kind of content format (`dm_format`) . By default, only HTML and `put_html` are sanitized.

To sanitize content, it must be loaded into memory, and converted into a readable string. When loaded into memory, there is one configuration to set the maximum content size to be loaded. When the content is larger than the set size, the content is not sanitized.

Note: When converting the content into a readable string, the charset of the content is required.

There is one new query parameter called `content-charset` that is added to the resources mentioned. This parameter tells the server how to parse the content. When the `content-charset` is not provided, the Rest server tries to get metadata information from the content itself.

When the metadata has a valid charset, is used to sanitize the content. When neither `content-charset` parameter, nor metadata information can be found, the configuration default charset is used.

Note: When the charset and the content do not match, then the uploaded content may have incorrect encoded values.

Configurations

```
# Determines whether to sanitize the metadata of an object.
# The sanitize will modify the metadata (remove the suspicious part).
# The default value is false.
rest.sanitize.type.metadata=false

# Determines whether to sanitize the content of an html.
# The sanitize will modify the content (remove the suspicious part).
# The default value is false.
rest.sanitize.type.content=false

# Set the max size of the content which will be sanitized.
# If the content size is larger than the max size, it will not be sanitized.
# The value is in bytes.
# The default value is 500K bytes.
rest.sanitize.content.max.size=500000

# Set the default html content charset/encoding.
# When sanitizing the input html content, the charset is determined by the following sequence:
# 1. the request parameter content-charset
# 2. try to get the charset from html meta info
# 3. both 1 and 2 failed, then use this default charset to sanitize the html content
rest.sanitize.content.default.charset=UTF-8

# Specifies the content formats which will be sanitized.
# Only the formats in this list will be sanitized.
# The formats in the list should be separated by ','.
# The format names refer to dm format.
rest.sanitize.content.format=html,pub_html
```

Customize the *AntiSamy* Policy Files

When you want to use customized *AntiSamy* policy files instead of the default policy files, you can create policy files under the `WEB-INF\classes` directory. This is the same directory where the `rest-api-runtime.properties` file is located.

You can create a `rest-antisamy-html.xml` file to define the policy for sanitization of file content, and `rest-antisamy-string.xml` to customize the object metadata sanitization policy.

There are two sample policy files provided:

- `rest-antisamy-html.sample.xml`
- `rest-antisamy-string.sample.xml`

You can create one or both of these files to customize your sanitization policy. When a custom policy file is provided, the system default policy file that would normally be used is ignored.

Explore Documentum Platform REST Services

This section provides a sample that guides you through some of the basic concepts and tasks in Documentum Platform REST Services. The sample focuses on common tasks involving folders, documents, and contents. By exploring the sample, you will become 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 Platform 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 Platform 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",
      "hints": {
        "allow": [
          "GET"
        ],
        "representations": [
          "application/xml",
          "application/json",
          "application/atom+xml",
          "application/vnd.emc.documentum+json"
        ]
      }
    },
    "about": {
      "href": "http://localhost:8080/dctm-rest/product-information",
      "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"
  }
],
"entries": [
  {
    "id": "http://localhost:8080/dctm-rest/repositories/acme01",
    "title": "acme01",
    "content": {
      "content-type": "application/json",
      "src": "http://localhost:8080/dctm-rest/repositories/acme01"
    },
    "links": [
      {
        "rel": "edit",
        "href": "http://localhost:8080/dctm-rest/repositories/acme01"
      }
    ]
  },
  {
    "id": "http://localhost:8080/dctm-rest/repositories/acme02",
    "title": "acme02",
    "content": {
      "content-type": "application/json",
      "src": "http://localhost:8080/dctm-rest/repositories/acme02"
    },
    "links": [
      {
        "rel": "edit",
        "href": "http://localhost:8080/dctm-rest/repositories/acme02"
      }
    ]
  }
]
}

```

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
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": "7.2.0000.0000Win64.SQLServer",
"docbroker": "CS70_Main"
},
"links": [
{
"rel": "self",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01"},
{
"rel": "http://identifiers.emc.com/linkrel/users",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/users"},
{
"rel": "http://identifiers.emc.com/linkrel/current-user",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/currentuser"},
{
"rel": "http://identifiers.emc.com/linkrel/groups",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/groups"},
{
"rel": "http://identifiers.emc.com/linkrel/cabinets",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/cabinets"},
{
"rel": "http://identifiers.emc.com/linkrel/formats",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/formats"},
{
"rel": "http://identifiers.emc.com/linkrel/network-locations",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/
network-locations"},
{
"rel": "http://identifiers.emc.com/linkrel/relations",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/relations"},
{
"rel": "http://identifiers.emc.com/linkrel/relation-types",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/relation-types"},
{
"rel": "http://identifiers.emc.com/linkrel/checked-out-objects",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/
checked-out-objects"},
{
"rel": "http://identifiers.emc.com/linkrel/types",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/types"},
{
"rel": "http://identifiers.emc.com/linkrel/dql",
"hreftemplate":
"http://localhost:8080/dctm-rest/repositories/acme01
{?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"
    }
  ],
  "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"
      },
      "links": [
        {
          "rel": "edit",
          "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000d1f"
        }
      ],
      "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"},
"links": [
{
"rel": "edit",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000107"}
]
}
]
```

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"
},
{
"rel": "edit",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/objects/0c0004d280000107"
},
{
"rel": "http://identifiers.emc.com/linkrel/delete",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/objects/0c0004d280000107"
},
{
"rel": "canonical",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000107"
},
{
"rel": "http://identifiers.emc.com/linkrel/folders",
"href":
"http://localhost:8080/dctm-rest/repositories/acme01/folders/0c0004d280000107/folders"
},
{
"rel": "http://identifiers.emc.com/linkrel/documents",
"href":
```

```

"http://localhost:8080/dctm-rest/repositories/acme01/folders/
0c0004d280000107/documents"
},
{
  "rel": "http://identifiers.emc.com/linkrel/objects",
  "href":
"http://localhost:8080/dctm-rest/repositories/acme01/folders/
0c0004d280000107/objects"
},
{
  "rel": "http://identifiers.emc.com/linkrel/child-links",
  "href":
"http://localhost:8080/dctm-rest/repositories/acme01/folders/
0c0004d280000107/child-links"
},
{
  "rel": "http://identifiers.emc.com/linkrel/relations",
  "href":
"http://localhost:8080/dctm-rest/repositories/acme01/relations?
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" },
    { "rel": "next",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
0c0004d280000107/folders?items-per-page=100&page=2"},
    { "rel": "first",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
0c0004d280000107/folders?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"
},
"links": [{"rel": "edit", "href": "http://localhost:8080/dctm-rest/
repositories/acme01/folders/0b0004d280009646"}]
},
{
  "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"
  },
  "links": [ { "rel": "edit", "href": "http://localhost:8080/dctm-rest/
repositories/acme01/folders/0b0004d280009647" } ]
},
...
}
```

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"},
    {
      "rel": "edit",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
0b0004d280009646"},
    {
      "rel": "http://identifiers.emc.com/linkrel/delete",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
0b0004d280009646"},
    {
      "rel": "http://identifiers.emc.com/linkrel/parent-links",
      "href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
0b0004d280009646/parent-links"},
    {
      "rel": "parent",

```



```

"title": "0c0004d280000107",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0c0004d280000107"},
{
  "rel": "http://identifiers.emc.com/linkrel/folders",
  "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0b0004d280009646/folders"},
{
  "rel": "http://identifiers.emc.com/linkrel/documents",
  "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0b0004d280009646/documents"},
{
  "rel": "http://identifiers.emc.com/linkrel/objects",
  "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0b0004d280009646/objects"},
{
  "rel": "http://identifiers.emc.com/linkrel/child-links",
  "href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/0b0004d280009646/child-links"},
{
  "rel": "http://identifiers.emc.com/linkrel/cabinet",
  "href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/0c0004d280000107"},
{
  "rel": "http://identifiers.emc.com/linkrel/relations",
  "href": "http://localhost:8080/dctm-rest/repositories/acme01/relations?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"
    }
  ],
  "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"
},
"links": [
{
"rel": "edit",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
090004d280009651"
}
]
}
]
```

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
```

```
{
  "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"},
{"rel": "edit",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
090004d280009894"},
{"rel": "http://identifiers.emc.com/linkrel/delete",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/documents/
090004d280009894"},
{"rel": "http://identifiers.emc.com/linkrel/parent-links",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/parent-links"},
{"rel": "parent",
"title": "0b0004d280009646",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/folders/
0b0004d280009646"},
{"rel": "http://identifiers.emc.com/linkrel/cabinet",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/cabinets/
0c0004d280000107"},
{"rel": "contents",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/contents"},
{"rel": "http://identifiers.emc.com/linkrel/primary-content",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/contents/content"},
{"rel": "http://identifiers.emc.com/linkrel/checkout",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/lock"},
{"rel": "version-history",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/versions"},
{"rel": "http://identifiers.emc.com/linkrel/current-version",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/versions/current"},
{"rel": "http://identifiers.emc.com/linkrel/relations",
"href": "http://localhost:8080/dctm-rest/repositories/acme01/
relations?related-object-id=090004d280009894&related-object-role=any"}
]
}

```

9. Send a POST request to create a 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:

```

POSThttp://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894/contents
Authorization: Basic ZGhhZG1pbjpwYXNzd29yZ
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 anHTTP 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?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?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
nBPcR6cWlqmXW2SIuP%2FeIbtI4upEs3%2B4aMZVeac9njIJ6zRosgm8yBIYAgm038KhDVOLF
lBxrb8Wsx%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"
    }
  ]
}
```

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!

10. 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.

```
DELETEhttp://localhost:8080/dctm-rest/repositories/acme01/objects/
090004d280009894
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.

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 Platform REST Services is programming language independent. You can use other languages to consume REST Services as well.

Basic Navigation

Documentum Platform REST Services has a rich hierarchy as follows:

- Documentum Platform 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 Platform 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()['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 in more details, shows the JSON representation of resources, and discusses some REST design principles.

Service Entry Point

Documentum Platform 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-information"
    }
  }
}
```

Link Relations

In a hypermedia-driven Documentum Platform 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()['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 10, Explore Documentum Platform 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()['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()['entries'] ]
```

For collections within a given repository, you can specify conditions that are used to select results by using [Filter Expression, page 43](#). 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 375](#) 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()['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()['entries']:
    print ( e['title'])
```

Note: The code sample 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()['entries']:
        p = e['content']['properties']
        print ( p['object_name'], ' ', p['title'])
    try:
        documents = get_link(response(), '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 52](#).

A filter always returns an entire resource. However, you can use [Property View, page 52](#) 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 21](#).

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()['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()
{
  "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()
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()
>>> response()
{
  "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()  
>>>
```


Appendix A

Link Relations

Documentum Platform REST Services uses the following link relations:

Table 18. Public Link Relations

Link Relation	Description	Specification
about	Returns product information	RFC 6903
canonical	Designates an Internationalized Resource Identifier (IRI) as preferred over resources with duplicate content.	RFC 6596
child	Points to a hierarchical child, or subobject, of the current object.	http://www.w3.org/MarkUp/draft-ietf-html-relrev-00.txt
contents	Points to the contents metadata feed for a contentful object.	REC-html401-19991224
edit	Points to a resource that can be used to edit the link's context.	RFC 5023
enclosure	Identifies a related resource that is potentially large and might require special handling.	new-link-relation
first	An IRI that refers to the furthest preceding resource in a series of resources.	RFC 5005
icon	Points to the icon for an entry, a page or a site.	rel-icon
last	An IRI that refers to the furthest following resource in a series of resources.	RFC 5005
next	Indicates that the link's context is a part of a series, and that the next in the series is the link target.	RFC 5005

Link Relation	Description	Specification
parent	Refers to one of the following: <ul style="list-style-type: none"> • 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 19. Link Relations Introduced in Documentum Platform REST Services

Link Relation	Description
acl	Points to the ACL resource for a specific permission set object.
acls	Points to the ACLs feed under a repository.
as-search-template	Points to the save as search template behavior for {\$1} {\$2}aved search.
associations	Points to the associated Sysobjects feed for a specific ACL object.
assist-values	Points to the type assist values resource to obtain the value assistance of a type.
aspect-types	Points to the aspect types resource to show all aspect types of a repository.
aspect-type	Points to the single aspect type resource.

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-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.
checkin-branch	Points to the checkin as branch version behavior for an object that is checked out.
child-links	Points to the child links feed resource of a folder.
checkout	Points to the checkout behavior for a versionable object.
comments	Points to top level comments for a sysobject.
content-media	Indicates that the content can be downloaded.
current-user	Points to the current login user resource under a repository.
current-user-preferences	Points to the current user preferences resource to manipulate the user preference.
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.
dematerialize	Points to the link to dematerialize a lightweight 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.
lightweight-types	Points to the types resource which contains the collection of children lightweight types of a shareable type.
lightweight-objects	Points to the collection of lightweight objects shares one shareable object.

Link Relation	Description
logoff	Points to the log out action for single sign on user.
materialize	Points to the link to materialize a lightweight object.
objects	Points to the Lists folder contents.
object-aspects	Points to the object aspects resource to get attached aspects of an object, or attache an aspect to the object.
parent-links	Points to the parent links resource of a non-cabinet sysobject.
parent-shareable-type	Points to the parent shareable type of a lightweight type.
permissions	Points to the permissions resource for a Sysobject.
permission-set	Points to the permission set resource for a Sysobject or a user.
primary-content	Points to the primary content resource for a content sysobject type.
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.
replies	Points to the log out action for single sign on user.
repositories	Points to the repositories feed from the docbrokers.
saved-searches	Points to the saved searches resource under a repository.
search-templates	Points to the search templates resource under a repository.
search-execution	Points to the execution behavior for a saved search or search template.
saved-search-results	Points to the results resource of a saved search.
shared-parent	Points the parent shareable object of a lightweight object.

Link Relation	Description
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.
virtual-document-nodes	Points to the virtual document nodes resource to fetch all nodes of an virtual document.
The fully qualified Documentum link relation path is prefixed with the following string: <code>http://identifiers.emc.com/linkrel/</code>	

Appendix B

Resource Coding Index

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
ACLs	acls	AtomFeed	AclsFeedView
ACL	acl	AclObject	AclView
ACL Associations	acl-associations	AtomFeed	SysObjectsFeedView
Aspect types	aspect-types	AtomFeed	AspectTypesFeedView
Aspect type	aspect-type	AspectTypeObject	AspectTypeView
Batch capabilities	batch-capabilities	BatchCapabilities	NA
Batches	batches	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
Comment	comment	Comment	CommentView
Comments	comments	AtomFeed	CommentsFeedView
Comment Replies	comment-replies	AtomFeed	CommentsFeedView
Content media resource	content-media	NA	NA
Content	content	ContentMeta	ContentView
Contents	contents	AtomFeed	ContentsFeedView
Current user	current-user	UserObject	UserView
Current user preferences	current-user-preferences	AtomFeed	PreferencesFeedView
Current user preference	current-user-preference	PreferenceObject	PreferenceView
Current version	current-version	SysObject	SysObjectView

Resource	Code Name	Model Class	View Class
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
Home document	home-document	NA	NA
Lightweight objects	object-lightweight-objects	AtomFeed	SysObjectsFeedView
Lock	lock	SysObject	SysObjectView
Materialization	materialization	SysObject	SysObjectView
Network locations	network-locations	AtomFeed	NetworkLocationsFeedView
Network location	network-location	NetworkLocation	NetworkLocationView
Object aspects	object-aspects	ObjectAspects	ObjectAspectsView
Object parent	object-parent	SysObject	SysObjectView
Parent folder link	parent-folder-link	FolderLink	FolderLinkView
Parent folder links	parent-folder-links	AtomFeed	FolderLinksFeedView
Permissions	permissions	AccessorPermission	AccessorPermissionView
Permission Set	permission-set	PermissionSetObject	PermissionSetView
Product information	product-information	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
Saved searches	saved-searches	AtomFeed	SavedSearchFeedView
Saved search	saved-search	SavedSearch	SavedSearchView
Search templates	search-templates	AtomFeed	SearchTemplateFeedView

Resource	Code Name	Model Class	View Class
Search template	search-template	SearchTemplate	SearchTemplateView
Saved search results	saved-search-results	SearchAtomFeed	SearchFeedView
Saved search execution	saved-search -execution	SearchAtomFeed	SearchFeedView
Sub groups	group-member -groups	AtomFeed	GroupsFeedView
SysObject	object	SysObject	SysObjectView
Type	type	TypeObject	TypeView
Types	types	AtomFeed	TypesFeedView
Type assistance values	type-assist-values	ValueAssistances	ValueAssistancesView
User default folder	default-folder	FolderObject	FolderView
User	user	UserObject	UIView
Users	users	AtomFeed	UsersFeedView
User Permission Set	user-permission-set	PermissionSetObject	PermissionSetView
Virtual document nodes	virtual-document -nodes	AtomFeed	VirtualDocumentNodes FeedView
<ol style="list-style-type: none"> 1. Model Classes for Core resources are in the package <code>com.emc.documentum.rest.model</code>. 2. View Classes for Core resources are in the package <code>com.emc.documentum.rest.view.impl</code>. 			