Smart Assurance EMC Data Access API (EDAA) User Guide

VMware Smart Assurance 10.1.2

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

1 Preface 4

2 Overview 7

EDA API 7

EDAA for SMARTS 7

EDAA key Tenets 8

3 Using the EDA API 9

Overview 9

EDAA Specification 13

Use of HTTP Headers 13

Using REST to access Type information 16

Query Parameters in EDAA 20

Consumer choice 38

Preface

1

As part of an effort to improve its product lines, VMware periodically releases revisions of its software and hardware. Therefore, some functions described in this document might not be supported by all versions of the software or hardware currently in use. The product release notes provide the most up-to-date information on product features.

Contact your VMware technical support professional if a product does not function properly or does not function as described in this document.

Audience

The Smart Assurance EMC Data Access API (EDAA) implements the Application Programmer Interface (API) used to develop applications that interact with the Domain Manager (DM). It enables Software Development Kit (SDK) users to build systems that apply SMARTS functionality to unique user requirements. These applications can be adapters that make the states of external devices and systems available to the SMARTS system or remote client applications that extend DM services to other client applications, such as Network Management systems.

All of the information needed to design, build and run an application using the EDA API is presented in this document. First, a context for developing SMARTS applications is established, describing a Domain Manager system and the role of the EDA API. Then concepts needed to integrate and apply the API through the development lifecycle are explained. This document and the programming language-specific API reference manual provide the information needed to develop software that uses the EDA API.

The discussion in this document applies to both applications that use event notification to drive their logic and adapters that interface to external devices.

The EDA API complements other tools for interfacing with a Domain Manager, such as the Adapter Scripting Language (ASL) and command line tools. Unlike these proprietary interfaces, however, the EDA API is based on common, standardized programming languages. The current languages supported is Java. Language bindings are described in separate reference manuals for each language, whereas this document is generally programming language independent.

This document is intended for software developers implementing subscription and adapter applications that involve the core event correlation, repository, and notification services of a Domain Manager.

Related documentation

The following VMware publications provide additional information:

- *Service Assurance Suite 9.5 Alert EMC Data Access API (EDAA) Programmer Guide*
- *Smarts Foundation 9.5 EMC Data Access API (EDAA) Programmer Guide*

Special notice conventions used in this document

VMware uses the following conventions for special notices:

Caution Indicates a hazardous situation which, if not avoided, will result in death or serious injury.

Important Indicates a hazardous situation which, if not avoided, could result in death or serious injury.

Caution Indicates a hazardous situation which, if not avoided, could result in minor or moderate injury.

Note Addresses practices not related to personal injury.

Note Presents information that is important, but not hazard-related.

Typographical conventions

VMware uses the following type style conventions in this document:

Bold	Used for names of interface elements
Italic	Used for full titles of publications referenced in text
`Monospace`	Used for: ■ System code ■ System output, such as an error message or script ■ Pathnames, filenames, prompts, and syntax ■ Commands and options
Monospace italic	Used for variables
`Monospace bold`	Used for user input
[ ]	Square brackets enclose optional values
|	Vertical bar indicates alternate selections - the bar means or
{ }	Braces enclose content that the user must specify, such as x or y or z
...	Ellipses indicate nonessential information omitted from the example

Where to get help

VMware support, product, and licensing information can be obtained as follows:

Product information

For documentation, release notes, software updates, or information about VMware products, go to VMware Online Support.

Technical support

Go to VMware Online Support and click Service Center. You will see several options for contacting VMware Technical Support. Note that to open a service request, you must have a valid support agreement. Contact your VMware sales representative for details about obtaining a valid support agreement or with questions about your account.

Overview

2

This chapter includes the following topics:

This chapter includes the following topics:

- EDA API
- EDAA for SMARTS
- EDAA key Tenets

EDA API

The EDA API (EMC Data Access API) is the system component that provides access to the facilities of the SMARTS product. It allows adapter and application developers to write software that extends and adapts functionality to their specific application requirements.

Developers build adapters that forward external events and topology updates to the system. There are also adapters that receive event notifications and correlation results from the core system that integrate with other applications such as network management systems. Developers also use the EDA API to build remote applications, such as user interfaces, that utilize the core information directly. The developer designs and builds systems using these components and the information they exchange (events, topology updates and correlation results).

EDAA for SMARTS

This topics describes how does EDAA fit into the SMARTS product.

The idea with EDAA is to establish a consistent style of REST API design across SMARTS. EDAA embodies a set of industry best practices around REST and applies them to the domain of IT Infrastructure Resource Management.

REST is a very simple concept, so simple that it can be used to build a very large variety of APIs. If you give the task of designing a REST API to 10 different product teams, then you will likely end up with 14 different REST API styles being deployed in products. The idea with EDAA is that we profile and codify best practices and approaches to REST API design so that the REST APIs to products adopting EDAA look very similar and have very similar characteristics. By having consistent approach to REST API design in SMARTS, we simplify the task of any consumer using two or more VMware software products.

EDAA key Tenets

EDAA is a "style" of REST architecture. REST itself is very simple, there are multiple ways to approach the task of building a REST API to a product. The fundamental notion of EDAA was to identify and codify certain REST best practices with the goal that VMware product teams and others would approach the task of designing and building a REST API to their product in a consistent fashion. Without this body of work to inform and guide product teams, these teams would deliver REST APIs to their products that reflected a vast variety of possible approaches -- reducing the ease with which consumers could access the APIs from multiple products.

Why adopt the EDAA style of REST API design?

When a product team adopts the EDAA style of REST API design, the customers of that product realize several benefits:

- The data and functionality embodied in the product is available through a simple, web friendly REST API. Software consumers of this API can be built, either as Mashups, Javascript web clients, Operating system resident (fat clients) programs written in various programming languages or even command line scripts using CURL.
- The REST API to the product follows REST industry best practices.
- The REST API to the product has a style very similar to other VMware products, allowing developers to learn EDAA once and then be very familiar with the REST API produced by many VMware products.

Using the EDA API 3

This section includes instructions for building and running a client using the EDA API with the Java binding as well as an example.

The following information is included in this chapter:

This chapter includes the following topics:

- Overview
- EDAA Specification
- Query Parameters in EDAA
- Consumer choice

Overview

The Java API defines a set of client-side base classes for developing client applications that interact with Domain Managers.

This chapter provides guidelines for building a basic application that uses the Java API to interact with Domain Managers. These guidelines include setting up a development environment to build solutions with the Java API and the SMARTS.

The EDA API is distributed as part of the SMARTS. Installing the SMARTS will provide the development environment described in this section.

The Java language implementation of the Remote API is distributed as the skclient.jar file, which is located in:

```
BASEDIR\smarts\classes
```

File `skclient.jar` includes the Java classes that directly implement the API, along with supporting classes. The four packages are:

- `com.smarts.decs`, which includes classes supporting the event correlation interface
- `com.smarts.remote`, which includes the main interface classes for accessing Domain Manager functionality
- `com.smarts.repos`, which includes supporting classes for working with repository objects

- `com.smarts.sos`, which includes supporting classes related to the operating system.

Sample Programs

The SMARTS includes the following sample Java programs that you can build and run:

- CloneClass
- getInstance

CloneClass

The CloneClass program provides clones an instance of a class which are passed as parameters in the CloneClass Method and renames it with the name passed again as the parameter to the method.

```
package SmartsEDAATestingIP.com.org.edaa;

import static com.jayway.restassured.RestAssured.given;
import static com.jayway.restassured.RestAssured.when;

import com.jayway.restassured.http.ContentType;
import com.jayway.restassured.response.Response;

import Utilities.SmartsGeneric;

public class Clone {

       private String hostname;
       private String username;
       private String password;
       private String broker;  //holds broker:port value
       private String serverName;

       public Clone(String hostname,String username,String password,String broker,String
serverName) {

              this.hostname = hostname;
              this.username = username;
              this.password = password;
              this.broker = broker;
              this.serverName = serverName;

       }


       public int  CloneClass(String className, String newinstancename , String instancename)
{

              Response resp =given().
                            body(" { arguments : {\"clone_name\" :
\""+newinstancename+"\" } }").
                            when().
                            contentType(ContentType.JSON).
```

```
                           post("http://"+hostname+":8080/smarts-edaa/msa/"+serverName+"/
instances/"+className+"::"+instancename+"/action/clone?alt=json&pretty=true");

              return resp.getStatusCode();

       }

       public static void main (String args[]){
              Clone sg ;
              sg = new Clone("itops-dev-210", "admin" , "changeme" , "itops-dev-210:426",
"INCHARGE-AM-PM-EDAA") ;
              int i  = sg.CloneClass("InCharge_NAS_Host_Feature","abcd", "Feature-NAS-
Host") ;

              System.out.print(" Response code is "+ i);
       }



}
```

Output: Response code is 200OK.

getInstance

The getInstance program gets the cloned instance from the last execution.

```
package SmartsEDAATestingIP.com.org.edaa;

import static com.jayway.restassured.RestAssured.given;
import static com.jayway.restassured.RestAssured.when;

import com.jayway.restassured.http.ContentType;
import com.jayway.restassured.response.Response;

import Utilities.SmartsGeneric;

public class GetClass {

       private String hostname;
       private String username;
       private String password;
       private String broker;  //holds broker:port value
       private String serverName;

       public GetClass(String hostname,String username,String password,String broker,String
serverName) {

              this.hostname = hostname;
              this.username = username;
              this.password = password;
              this.broker = broker;
              this.serverName = serverName;

       }
```

```
        public int getInstance (String className, String clonedinstancename)
        {
                Response resp = when().
                               get("http://"+hostname+":8080/smarts-edaa/msa/"+serverName+"/
instances/"+className+"::"+clonedinstancename+"?alt=json&pretty=true");


                return resp.getStatusCode();
        }

        public static void main (String args[]){
                GetClass sg ;
                sg = new GetClass("itops-dev-210", "admin" , "changeme" , "itops-dev-210:426",
"INCHARGE-AM-PM-EDAA") ;
                int i  = sg.getInstance("InCharge_NAS_Host_Feature","abcd") ;

                System.out.print(" Response code is "+ i);
        }



}
```

Output: Response code is 200OK.

Post Request

The Post request has two parts, the URL to which you send the data and the arguments which form the payload of the post request.

The EDAA gives you the flexibility to perform all the operation which is done on an instance of a class through `dmctl` using a web call. For example, the URL to which the post request is sent is:

`localhost:8080/smarts-edaa/msa/servername/instances/className::instanceName/action/action_name`

For example, if you want to perform clone action on `Feature-NAS-Host` instance of `InCharge_NAS_HOST_Feature` class, which is hosted on a domain with name `INCHARGE-AM-PM` on the host `<ip address>`. The URL which is used to post the request using the above mentioned format must be:

`http://<host ip address>:8080/smarts-edaa/msa/INCHARGE-AM-PM-EDAA/instances/InCharge_NAS_Host_Feature::Feature-NAS-Host/action/clone?alt=json&pretty=true`

In the payload of this post request you must pass all the parameters needed for that particular action to execute. In this case the clone action expects `clone_name` as the parameter. So, the payload must be:

```
{
arguments : {
          "clone_name" : "new_name"
        }
}
```

EDAA Specification

This topic is a normative specification of EMC Data Access API (EDAA).

Use of HTTP Headers

This section describes how EDAA uses http headers, specifically eTags, and the Accept header for API version and language negotiation.

Language Negotiation

A consumer of an EDAA may use the http Accept-Language header to specify which languages/ locales it would prefer certain human-readable components of the response be translated into. Currently, EDAA defines the Message information item of an Error resource as the only normative aspect that is governed by the Accept-Language header. An EDAA MAY use language preference to alter other aspects of the response formatting.

Note also that EDAA supports an alternative form of expressing language preference. An language= optional query parameter may also be included on EDAA URIs to specify consumer preference for language/locale. If the client request uses both the Accept- Language header and an language= query parameter to express preference for the response format, then a response is given only if there is at least one language/locale that overlaps between the Accept-Language header and the languages= header. For example a request:

```
GET /types?language=da
Accept-Language: es, en-gb;q=0.8, en;q=0.7
```

contains a conflict between the language query parameter and the Accept-Language: header. There is no language/locale that matches both criteria. In this case, the server must return an error (406 not acceptable).

JSON Equivalent of Feed and Entry eTags

EDAA implementations are encouraged to support the notion of eTags for cache control and optimistic concurrency control.

EDAA implementations that implement eTags and also support the JSON serialization format described in this page MUST annotate those feeds and entries with eTags. Feed level eTags should be weak eTags and entry level etags should be strong eTags.

Here is an example of an eTag at the feed level:

```
{
"etag": 'W/"C0QBRXcycSp7ImA9WxRVFUk."'
... other metadata about the feed
"entries":[
{
entry ...
},
... etc for all the entries in the feed
],
...
}
```

Here is an example of an eTag at the entry level:

```
{
"etag": '"ADEEFO42drp7tKA7QxRDBIL."'
other metadata about the entry... ,
representation of a resource within the content object ... ,
...
}, ...
}
```

Using eTag in a GET

For this, we assume the EDAA is supporting eTags in the style described by EDAA. The consumer could issue a GET operation, for example:

```
GET /types/FileServer/instances
```

and the response might look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<atom:feed ...
xmlns:gd='http://schemas.google.com/g/2005'
gd:etag='W/"B9roqXcycSp7ImA9WxRVDdk."'>
...
<atom:link href="https://.../types/FileServer/instances" rel="self"/>
...
<atom:entry ...
xmlns:gd='http://schemas.google.com/g/2005'
gd:etag='"U822F2drp7tKA7QxRD2Ko."'>
...
<atom:link rel='self' href='http:.../instances/Fileserver::1234' />
...
</atom:entry>
<atom:entry>
...
</atom:feed>
```

At some point later in time, the consumer may want to "refresh" the feed to see if anything had changed. Clearly the consumer could re-issue the same GET and reprocess the response. However, if the consumer has a cached copy of the feed it previously retrieved and if it re-issued the operation as follows:

```
GET /types/FileServer/instances
If-None-Match: W/"B9roqXcycSp7ImA9WxRVDdk."
```

Then the EDAA could note that the eTag in the If-None-Match header matches the eTag associated with its version of the feed, and if they match, return an http 304 (not modified) status code with an empty response body. Saving processing time on the server side, saving network bandwidth, reducing the latency of the response to the client and saving the client from having to re-process a feed that is no different from the version it already has. Of course, if the eTags don't match then a full response is returned to the consumer with http 200 (ok) status code.

This also works at the entry level. The consumer could inspect entry level eTags, and using the @href in the rel="self" atom:link within the atom:entry, could issue a request to "refresh" the value of the entry if that entry had changed:

```
GET /instances/Fileserver::1234
If-None-Match: “U822F2drp7tKA7QxRD2Ko."
```

And the EDAA could check if the eTags match, and if they do, return the 304 response, but if they don't, send a full response with 200 (ok) status.

Using eTag in a PUT or PATCH operation

For this, we assume the EDAA is supporting eTags in the style described by this spec. The consumer could issue a GET operation, for example:

```
GET /types/FileServer/instances
and the response might look like:
<?xml version="1.0" encoding="UTF-8"?>
<atom:feed ...
xmlns:gd='http://schemas.google.com/g/2005'
gd:etag='W/"B9roqXcycSp7ImA9WxRVDdk."'>
...
<atom:link href="https://.../types/FileServer/instances" rel="self”/>
...
<atom:entry ...
xmlns:gd='http://schemas.google.com/g/2005'
gd:etag='“U822F2drp7tKA7QxRD2Ko."'>
...
<atom:link rel='self' href='http:.../instances/Fileserver::1234' />
...
</atom:entry>
<atom:entry>
...
</atom:feed>
```

At some point in the future, the consumer may wish to use PUT or PATCH to make modifications to the resource. When an EDAA decorates response feeds with eTag elements, then it MUST also require consumers to annotate any resource modification request with an http If-Match header containing the eTag of the resource being modified. For example, to modify the FileServer instance retrieved in the previous example, a PUT request as follows could be formed:

```
PUT /instances/Fileserver::1234
If-Match: "U822F2drp7tKA7QxRD2Ko."
... body of the PUT contains a partial representation (for update) of a FileServer instance
```

Note, the content of the If-Match: header is the eTag value contained in the atom:entry corresponding to the resource being updated. The EDAA MUST compare the value of the eTag given in the If-Match header with the current eTag value for the resource. If the values match, the modification operation may proceed. If the values do not match, the EDAA MUST respond with an http 412 (Precondition Failed) response.

If the EDAA provided an eTag with the resource representation, as shown above, and the consumer does not include an If-Match: header in a modification request, then the EDAA MUST reject the request with an http 412 (Precondition Failed) response.

Using REST to access Type information

EDAA defines the following URI patterns to help developers understand the resource model of the EDAA. Each of these URI patterns is read-only (GET) and return a feed of one or more resource type representations.

In fact the reason that all EDAA URI patterns start with /types or /instances is to support the idea of treating type resources as first class resources in the EDAA approach and thereby to distinguish /types resources from /instances resources.

The examples below show the atom/xml representation of the responses. EDAA has also defined a JSON equivalent of these feeds.

/types

GET operation to retrieve a paginated feed of resource representations. Each resource in the feed is a representation of a "type resource". Current format of a type resource is "VS-XML", an XSD-like syntax for defining types, attributes and relationships of the type and other type related metadata.

page and per_page query parameters are supported to shape the pagination of the feed.

An example response is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom" xmlns="http://www.w3.org/2005/Atom"
xmlns:vsc="http://schemas.emc.com/vs-xml/namespace/Common/1.0">
<title>/slm/msa/types</title>
<updated>2011-05-23T16:52:06-05:00</updated>
<author><name>msa framework</name></author>
<id>74a72d5a-de7c-4f82-a753-b4a5b00afe28</id>
```

```
<entry>
<title type='text'>vCenter</title>
<id>http://localhost:8080/slm/msa/types/vCenter</id>
<updated>2011-05-23T16:52:06-05:00</updated>
<link rel='edit' href='http://localhost:8080/slm/msa/types/vCenter/instances' />
<link rel='related' href='http://localhost:8080/slm/msa/types/vCenter/instances' />
<link rel='alternate' href='http://localhost:8080/slm/msa/types/vCenter' />
<link rel='self' href='http://localhost:8080/slm/msa/types/vCenter' />
<content type='application/xml'>
<vsc:Type xmlns:atom='http://www.w3.org/2005/Atom' xmlns:vsc='http://schemas.emc.com/vs-xml/
namespace/Common/1.0'
xmlns:inst='http://schemas.emc.com/msa/uim/1.0'>
<vsc:typeName namespace='http://schemas.emc.com/msa/uim/1.0'>vCenter</vsc:typeName>
<atom:link rel='http://schemas.emc.com/msa/common/reln/PR_Create'
href='http://localhost:8080/slm/msa/types/vCenter/PR_Create' />
<atom:link rel='self' href='http://localhost:8080/slm/msa/types/vCenter' />
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>displayName</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>ipAddress</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:long' minOccurs='1'
maxOccurs='1'>id</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>name</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>description</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>vCenterVersion</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>userName</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>port</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>password</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>connectionStatus</vsc:attribute>
<vsc:relationship relType="vCenterDatacenter" type="http://schemas.emc.com/msa/common/
contains" minOccurs="1"
maxOccurs="unbounded"
description="List of Datacenters">Datacenters</vsc:relationship>
<vsc:action rel='edit' description='Instances are mutable' />
<vsc:action rel='http://schemas.emc.com/msa/uim/1.0/vCenter/action/verifyConnection'
description='Verifies if vCenter
is reachable' />
</vsc:Type>
</content>
</entry>
<entry>
<title type='text'>vCenterDatacenter</title>
<id>http://localhost:8080/slm/msa/types/vCenterDatacenter</id>
<updated>2011-05-23T16:52:06-05:00</updated>
<link rel='related' href='http://localhost:8080/slm/msa/types/vCenterDatacenter/instances' />
<link rel='alternate' href='http://localhost:8080/slm/msa/types/vCenterDatacenter' />
<link rel='self' href='http://localhost:8080/slm/msa/types/vCenterDatacenter' />
<content type='application/xml'>
```

```
<vsc:Type xmlns:atom='http://www.w3.org/2005/Atom' xmlns:vsc='http://schemas.emc.com/vs-xml/
namespace/Common/1.0'
xmlns:inst='http://schemas.emc.com/msa/uim/1.0'>
<vsc:typeName namespace='http://schemas.emc.com/msa/uim/1.0'>vCenterDatacenter</vsc:typeName>
<atom:link rel='http://schemas.emc.com/msa/common/reln/PR_Create'
href='http://localhost:8080/slm/msa/types/vCenterDatacenter/PR_Create' />
<atom:link rel='self' href='http://localhost:8080/slm/msa/types/vCenterDatacenter' />
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>displayName</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:long' minOccurs='1'
maxOccurs='1'>id</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>name</vsc:attribute>
<vsc:attribute type='http://www.w3.org/2001/XMLSchema-datatypes:string' minOccurs='1'
maxOccurs='1'>managedObjectKey</vsc:attribute>
<vsc:relationship relType='vCenter' type='http://schemas.emc.com/msa/common/ownedBy'
minOccurs='1' maxOccurs='1'
description='vCenter this Datacenter belongs to'>vCenter</vsc:relationship>
</vsc:Type>
</content>
</entry>
...
</feed>
```

/types/{typeName}

GET operation to retrieve a representation of the type resource identified by {typeName}. Current format of a type resource is "VSXML", an XSD-like syntax for defining types, attributes and relationships of the type and other type related metadata.

An example response to GET /types/vCenter is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom" xmlns="http://www.w3.org/2005/Atom"
xmlns:vsc="http://schemas.emc.com/vs-xml/namespace/Common/1.0">
<title>/slm/msa/types/vCenter</title>
<updated>2011-06-02T16:24:27-05:00</updated>
<author><name>msa framework</name></author>
<id>8cc4c3fc-5821-47ad-9b65-0ed5b2809a3c</id>
<link rel="self" href="http://localhost:8080/slm/msa/types/vCenter"/>
<entry>
<title type="text">vCenter</title>
<id>http://localhost:8080/slm/msa/types/vCenter</id>
<updated>2011-06-02T16:24:27-05:00</updated>
<link rel="edit" href="http://localhost:8080/slm/msa/types/vCenter/instances"/>
<link rel="related" href="http://localhost:8080/slm/msa/types/vCenter/instances"/>
<link rel="alternate" href="http://localhost:8080/slm/msa/types/vCenter"/>
<link rel="self" href="http://localhost:8080/slm/msa/types/vCenter"/>
<content type="application/xml">
<vsc:Type xmlns:inst="http://schemas.emc.com/msa/uim/1.0">
<vsc:typeName namespace="http://schemas.emc.com/msa/uim/1.0">vCenter</vsc:typeName>
<atom:link rel="http://schemas.emc.com/msa/common/reln/PR_Create"
href="http://localhost:8080/slm/msa/types/vCenter/PR_Create"/>
<atom:link rel="self" href="http://localhost:8080/slm/msa/types/vCenter"/>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
```

```
maxOccurs="1">displayName</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">ipAddress</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:long" minOccurs="1"
maxOccurs="1">id</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">name</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">description</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">vCenterVersion</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">userName</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">port</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">password</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">connectionStatus</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">dataCentersList</vsc:attribute>
<vsc:relationship relType="vCenterDatacenter" type="http://schemas.emc.com/msa/common/
contains" minOccurs="1"
maxOccurs="unbounded"
description="List of Datacenters">Datacenters</vsc:relationship>
<vsc:action rel="edit" description="Instances are mutable"/>
<vsc:action rel="http://schemas.emc.com/msa/uim/1.0/vCenter/action/verifyConnection"
description="Verifies if vCenter
is reachable"/>
</vsc:Type>
</content>
</entry>
</feed>
```

/types/{typeName}/hierarchy

GET operation to retrieve a paginated feed of resource representations of each type resource in the type hierarchy starting with the type identified by {typeName}. Each resource in the feed is a representation of a "type resource". Current format of a type resource is "VS-XML", an XSD-like syntax for defining types, attributes and relationships of the type and other type related metadata.

The idea is that if a type named "ApplicationServer" is a subclass of "SoftwareService" which is a subclass of "SoftwareElement", then the response to:

GET /types/ApplicationServer/hierarchy

would be a feed containing series of entries, one entry for the ApplicationServer type, followed by an entry for the SoftwareService type followed by the SoftwareElement type.

/types/{typeName}/PR_Create

The /PR_Create URI pattern was introduced in support of the partial representations used to create new resources of a given type. As discussed in EDAA Read/Write spec, the challenge for consumers of the read-write portion of MSA is to determine, for any given type, what is the subset of attributes and relationships that a third party consumer must specify when a resource is created.

/types/{typeName}/PR_Create contains a VS-XML representation of the subset of attributes and relationships of the type identified by {typeName} that must appear in the body of a POST (create) operation. Note the cardinality constraints on each attribute and relationship as specified by the @minOccurs and @maxOccurs on the attribute declaration and relationship declaration elements.

Here is an example response from GET /types/vCenter/PR_Create:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom" xmlns="http://www.w3.org/2005/Atom"
xmlns:vsc="http://schemas.emc.com/vsxml/
namespace/Common/1.0">
<title>/slm/msa/types/vCenter/PR_Create</title>
<updated>2011-06-03T09:17:05-05:00</updated>
<author><name>msa framework</name></author>
<id>b28dd0d9-60a6-4204-bf7f-d7b6d8ca2740</id>
<link rel="self" href="http://localhost:8080/slm/msa/types/vCenter/PR_Create"/>
<entry>
<title type="text">vCenter - PR_Create</title>
<id>http://localhost:8080/slm/msa/types/vCenter/PR_Create</id>
<updated>2011-06-03T09:17:05-05:00</updated>
<link rel="related" href="http://localhost:8080/slm/msa/types/vCenter"/>
<link rel="self" href="http://localhost:8080/slm/msa/types/vCenter/PR_Create"/>
<content type="application/xml">
<vsc:Type xmlns:inst="http://schemas.emc.com/msa/uim/1.0">
<vsc:typeName namespace="http://schemas.emc.com/msa/uim/1.0">vCenter</vsc:typeName>
<atom:link rel="self" href="http://localhost:8080/slm/msa/types/vCenter/PR_Create"/>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">displayName</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">ipAddress</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">name</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">description</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">userName</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">password</vsc:attribute>
</vsc:Type>
</content>
</entry>
```

Query Parameters in EDAA

The following table is the set of query parameters defined by EDAA specification:

Parameter	Meaning
page and per_page	Pagination related
alt	Alternative format of the resource representation (eg atom or JSON)
fields	Specify the subset of fields (properties) of the resource that should be returned in the response (like SELECT in SQL)
expand	Augment relationship representations with an in line feed of related resources
orderby	comma separated list of properties the response should be sorted on. The response MUST present an atom:feed with the entries sorted by the values of the indicated fields (like ORDERBY in SQL)
filter	simple boolean predicate expression to describe a filter, or subset, of the resources to return in a GET operation (like WHERE in SQL).
languages	a string, in the format defined for http Accept-Languages header, containing a comma separated list of languages/locales (with quality weightings) that expresses the consumer's preference for localizing responses.

This table summarizes which query parameter is applicable for the various URI patterns:

URI Pattern	page	per_page	alt	fields	expand	orderby	filter	languages
/types	Y	Y	Y			Y	Y	Y
types/{typeName}			Y					Y
/types/{typeName}/hierarchy	Y	Y	Y					Y
/types/{typeName}/PR_Create			Y					Y
/types/{typeName}/instances	Y	Y	Y	Y	Y	Y	Y	Y
/instances	Y	Y	Y	Y	Y	Y		Y
/instances/{id}			Y	Y	Y			Y
/instances/{id}/relationships			Y	Y				Y
/instances/{id}/relationships/{relName}	Y	Y	Y	Y	Y	Y	Y	Y

Note, query parameters can be combined in a single request, for example:

```
GET /types/vCenter/
instancespage=1&per_page=20&alt=atom&fields=displayName,id,connectionStatus&orderby=id
OR
GET /types/vCenter/
instancespage=1&per_page=20&alt=atom&fields=displayName%7Cid%7CconnectionStatus&orderby=id
```

Note In the above query fields are separated by the OR(%7C) condition.

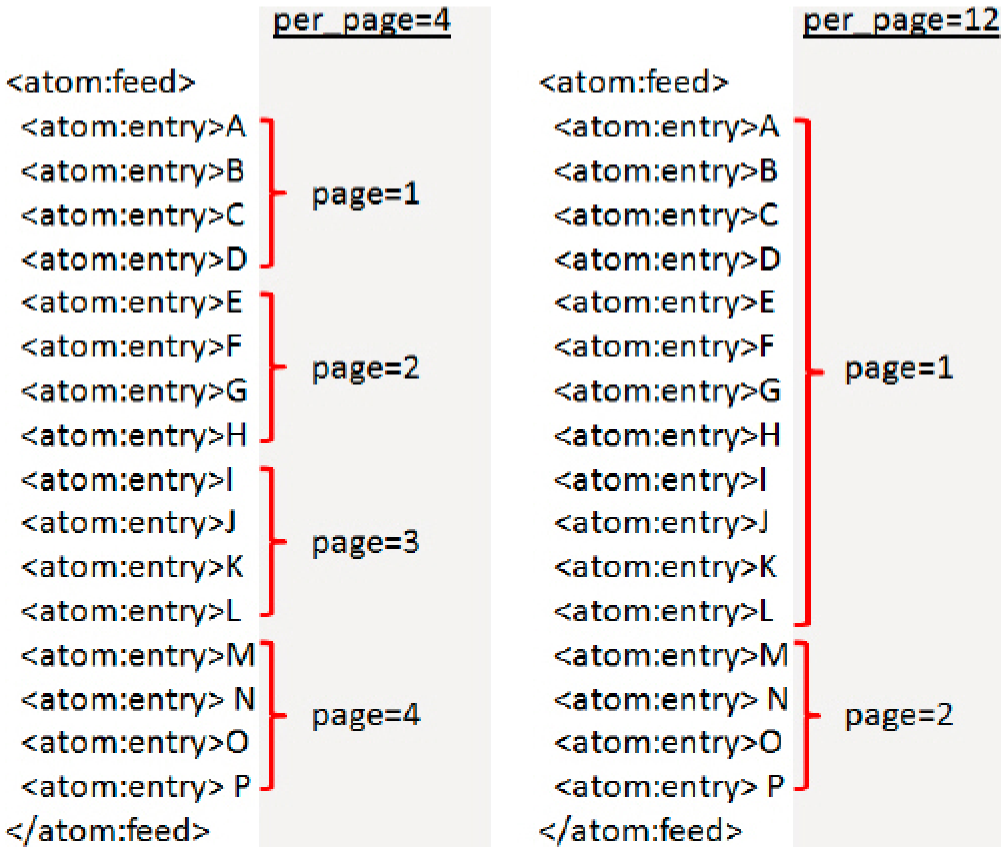
may be an appropriate (but complicated) request.

Note also that filter and orderby affect the interpretation of page because those filter changes which atom:entry elements are included in the "conceptual" atom:feed over which the request operations and orderby changes the order in which those atom:entry elements appear.

page and per_page

These query parameters define a "chunking" or "paging" view over a large collection of resources in an atom:feed.

An atom:feed is really just a collection of atom:entry elements, potentially a large number of them, conceptually an infinite number. The page and per_page query parameters work together to define a sequence of "chunks" called "pages" over this collection of atom:entries, each page containing at most the number of atom:entry elements as defined by per_page.



In the chunking on the left, the "conceptual" atom:feed is divided into pages of 4 each, resulting in the entire feed being "chunked" into 4 pages.

```
GET /{URL to conceptual feed}per_page=4&page=2
```

would return an atom:feed containing the 3rd page of the "conceptual" atom:feed, containing the following 4 atom:entry elements:

```
<atom:feed>
<atom:entry>E
<atom:entry>F
<atom:entry>G
<atom:entry
```

Whereas, using the chunking on the right, which divides the "conceptual" atom:feed into pages of 12, there are only 2 pages. The same request, but specifying a different value of per_page:

```
GET /{URL to conceptual feed}per_page=12&page=2
```

would result in a different response:

```
<atom:feed>
<atom:entry>M
<atom:entry>N
<atom:entry>O
<atom:entry>P
</atom:feed>
```

Note that the two queries return different responses. Note also, in the second response that even though per_page=12 it did not guarantee there would be 12 atom:entry elements in the response.

Both page and per_page have integer values. If a value of page or per_page is not an integer, the EDAA MUST respond with an http error code 400 (bad request).

It is possible to specify ridiculous values of per_page. If per_page is not specified in the URL, or the value of per_page is less than 1, the EDAA implementation will substitute some default value of per_page, usually 20). If the value of per_page is some huge integer, like 10000, that exceeds the size of the "conceptual" atom:feed, then the request is accepted and 1 page containing the entire atom:feed is returned. For large values of per_page, the response time of the query and the processing requirements on the client and server may be large. It is not recommended that the client use large values of per_page unless they are prepared to wait for some time to receive the response and are prepared to consume/process a large response.

The values of page are a bit more constrained. If page is not specified in the URL, or the value of page is less than or equal to 1, then the first page (page=1) is used as the default value. If the value of page exceeds the number of pages in the conceptual feed, an error response is returned with http code 400 (bad request).

It should be noted that the semantics of paging is altered by the filter and orderby query parameters. If two queries are exactly the same, except for the value of orderby, then the response from each query will represent different orderings of the atom:entry elements in the "conceptual" atom:feed and therefore the contents of page=1 will likely be different between the two queries. Similarly, filter changes which atom:entry elements are in the "conceptual" atom:feed and will therefore change which atom:entry elements appear in any of the pages.

Interpretation by URI Pattern

The following table describes how page and per_page are interpreted for each URI pattern. For those URI patterns that page and per_page apply, the "Default" value of page is 1. "Default" value of per_page is server determined, usually 20. For those URI patterns where page and per_page don't apply, those query parms are silently ignored if present in the URL.

URI Pattern	Applicable?	Comments
/types	Yes	"conceptual" atom:feed is all the type resources known to the EDAA
/types/{typeName}	No	Response is a single (type) resource, no "conceptual" atom:feed is associated with the response.
/types/{typeName}/hierarchy	Yes	"conceptual" atom:feed is all the type resources within the type hierarchy of the type identified by {typeName}
/types/{typeName}/PR_Create	No	Response is a single (type) resource, no "conceptual" atom:feed is associated with the response.
/types/{typeName}/instances	Yes	"conceptual" atom:feed is all the instance resources of the type identified by {typeName}
/instances	Yes	"conceptual" atom:feed is all the instance resources known to the EDAA
/instances/{id}	No	Response is a single instance resource, no "conceptual" atom:feed is associated with the response.
/instances/{id}/relationships	No	Response is a single instances resource, no "conceptual" atom:feed is associated with the response.
/instances/{id}/relationships/{relName}	Yes	"conceptual" atom:feed is all the instance resources related to the resource identified by {id} through the relationship named {relName}

alt

This query parameter allows a URL-level mechanism to control which format (Atom/XML or JSON) is used to serialize the resource representation in response to the request.

This query parameter is a convenience mechanism, useful for experimenting in a browser window. The functionality is duplicated with Content Negotiation in EDAA using the http Accept: header to specify consumer preference of format. It is preferred that the consumer use http Accept: headers to express format preference.

The range of value for alt is a string enumeration. Currently the only valid values of alt are "atom" and "json". More values may be added to this enumeration in the future, as additional formats are supported by EDAA (such as, perhaps csv for a "comma separated variable" serialization). If an invalid value of alt is specified in a URL, the EDAA MUST respond with an error, http code 400 (bad request).

Note, not all EDAA implementations are expected to support both Atom/XML and JSON serialization formats. Ideally, an EDAA should support both, but it is not strictly required. If a consumer specifies a value of alt that is valid, but not one of the serialization formats supported by the EDAA (for example the consumer uses alt=json on an EDAA that supports only Atom/XML), then the EDAA MUST respond with an error, http code 400 (bad request).

If no alt is specified in the URL, the EDAA implementation is free to choose which supported serialization format to use in the response.

As mentioned previously, the consumer can express serialization format preference using alt or an http Accept: header. In the case where the consumer uses both approaches, AND the approaches conflict (eg alt="atom" and the http Accept: specifies JSON), then the EDAA MUST return an error, 406 (not acceptable).

If a valid value of alt is specified in the URL, there is no conflict with an http Accept: header in the request and the corresponding serialization format is supported by the EDAA, then the EDAA MUST format the response to the request using the serialization format specified in alt.

For the value of alt=json, the serialization of the response is governed by the JSON conventions in EDAA also the rules to represent type resources and instance resources specified in EDAA.

If no value of alt is given in the URL and no Content type is specified in an http Accept: header, then the EDAA is free to choose which of serialization format to use for the response. If the EDAA supports alt=atom, it MUST use that as the default.

Interpretation by URI Pattern

The alt query parameter may appear on any URI pattern specified in EDAA.

fields

The idea of a fields query parameter is to allow the consumer to specify partial representations of the resource to be returned in responses. This provides a more succinct and fit for purpose representation to be specified by consumers, and avoids the overhead of server-side and client-side processing of attributes and relationships that are not of interest to the consumer.

With the fields query parameter, the consumer specifies a comma separated list of attribute names and relationship names. The resource representation(s) returned in the response will contain only those attributes and relationships specified in the fields query parameter.

For example, examine the type information for the "vCenter" type:

```
...
<vsc:typeName namespace="http://schemas.emc.com/msa/uim/1.0">vCenter</vsc:typeName>
<atom:link rel="http://schemas.emc.com/msa/common/reln/PR_Create" href="http://
localhost:8080/slm/msa/types/vCenter/PR_Create"/>
<atom:link rel="self" href="http://localhost:8080/slm/msa/types/vCenter"/>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">displayName</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">ipAddress</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:long" minOccurs="1"
maxOccurs="1">id</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">name</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">description</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">vCenterVersion</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">userName</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">port</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">password</vsc:attribute>
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">connectionStatus</vsc:attribute>
```

```
<vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:string" minOccurs="1"
maxOccurs="1">dataCentersList</vsc:attribute>
<vsc:relationship relType="vCenterDatacenter" type="http://schemas.emc.com/msa/common/
contains" minOccurs="1" maxOccurs="unbounded"
description="List of Datacenters">Datacenters</vsc:relationship>
...
```

If the following request is made:

```
GET /types/vCenter/instancesfields=displayName,id,Datacenters
OR
GET /types/vCenter/instancesfields=displayName%7Cid%7CDatacenters%7C
```

Note In the above query fields are separated by the OR(%7C) condition.

The response would contain partial resource representations for each resource, containing only the displayName and id attributes and the Datacenters relationship:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom" xmlns="http://www.w3.org/2005/Atom"
xmlns:vsc="http://schemas.emc.com/vs-xml/namespace/Common/1.0">
<title>/slm/msa/types/vCenter/instances</title>
<updated>2011-05-24T08:20:55-05:00</updated>
<author><name>msa framework</name></author>
<id>5aabb570-340e-4e6d-adaf-91c38c5e743a</id>
<link rel="self" href="http://localhost:8080/slm/msa/types/vCenter/
instancesfields=displayName,id,Datacenters"/>
<entry xmlns:gd='http://schemas.google.com/g/2005'
xmlns:atom="http://www.w3.org/2005/Atom"
gd:etag='“ADEEFO42drp7tKA7QxRDBIL."'>
<atom:title type='text'>vCenter - lglan195</atom:title>
<atom:id>http://localhost:8080/slm/msa/instances/vCenter::1</atom:id>
<atom:updated>2011-05-24T08:20:55-05:00</atom:updated>
<atom:link rel='self' href='http://localhost:8080/slm/msa/instances/
vCenter::1fields=displayName,id,Datacenters' />
<atom:link rel='alternate' href='http://localhost:8080/slm/msa/instances/vCenter::1' />
<atom:content type='application/xml'>
<inst:vCenter xmlns:atom='http://www.w3.org/2005/Atom'
xmlns:vsc='http://schemas.emc.com/vs-xml/namespace/Common/1.0'
xmlns:inst='http://schemas.emc.com/msa/uim/1.0'>
<inst:displayName>lglan195</inst:displayName>
<inst:id>1</inst:id>
<atom:link rel='http://schemas.emc.com/msa/uim/1.0/vCenter/relationship/Datacenters'
href='http://localhost:8080/slm/msa/instances/vCenter::1/relationships/Datacenters' />
</inst:vCenter>
</atom:content>
<entry>
<atom:title type='text'>vCenter - matt 111</title>
<atom:content type='application/xml'>
<inst:vCenter ...
<inst:displayName>matt 111</inst:displayName>
<inst:id>9</inst:id>
<atom:link rel='http://schemas.emc.com/msa/uim/1.0/vCenter/relationship/Datacenters'
href='http://localhost:8080/slm/msa/instances/vCenter::9/relationships/Datacenters' />
```

```
</inst:vCenter>
</atom:content>
</entry>
... etc. for every vCenter instance
</feed>
```

Note how the representation of each vCenter resource is a partial representation, containing only the properties of the resource specified in fields.

When an EDAA is processing a partial representation formed by a fields query parameter, the EDAA reduces the content of each representation to include only those attributes and relationships whose name appears in the list of names in the value of fields. If a resource does not have an attribute or relationship corresponding to one of the names in the list, that name is silently ignored. The net result is that the representation of any resource appearing in the response contains only the properties named in the fields query parameter. It is possible that a resource type could have an attribute and a relationship with the same name. In this case, if a name is specified in the value of fields that corresponds to both an attribute and a relationship then the resulting partial representation will contain both properties.

Interpretation by URI Pattern

The following table describes how fields is interpreted for each URI pattern. For those URI patterns for which fields applies, the "Default" value of fields is an empty string, meaning to include all properties (eg a full representation of each resource). For those URI patterns where fields doesn't apply, it is silently ignored if present in the URL.

URI Pattern	Applicable?	Comments
/types	No	
/types/{typeName}	No	
/types/{typeName}/hierarchy	No	
/types/{typeName}/PR_Create	No	
/types/{typeName}/instances	Yes	Since all the resources in the response are of the same type, then this use of fields is very useful to generate focused partial representations. Since a large number of resources may be returned in the response, specifying a very compact representation for each resource can be extremely beneficial to request/response latency and client and server side resource use.
/instances	Yes	Not terribly useful because there will be few property names that are shared amongst all types of resource instance. Many of the representations will be empty, even for common property names like "name" or "displayName" or "id". However, the partial representation filter implied by fields will be applied to each resource in the response.
/instances/{id}	Yes	Questionably useful, given that atom:feeds containing a single resource, as expected from this URI pattern, aren't often very large, and therefore the partial representation technique really doesn't reduce the response size significantly from the full representation of the resource.

URI Pattern	Applicable?	Comments
/instances/{id}/ relationships	Yes	As with /instances/{id}, above
/instances/{id}/ relationships/{relName}	Yes	Applies to the related resource representations contained in the response. Since this request is focused on a single relationship, the resources in the response are all of the same type, and therefore a reasonable partial representation can be specified by fields.

expand

The idea with the expand query parameter is to give the consumer some control of the representation of relationships that appear in a resource's representation. The consumer uses expand on a GET operation to specify which relationships should be expanded. An expanded relationship augments the normal relationship representation, an atom:link element, with a child (or inline) feed of related resources. If a consumer wants to avoid having to do an additional GET operation to retrieve the resources related to a particular resource, it would use the expand query parameter to add additional information about related resources across one or more relationships.

Note, the in line feed of related resources may also be paginated, at the control of the server. The use of page and per_page does not compose with expand. The page and per_page does not alter the pagination properties of the in-line feeds. The pagination of in-line feeds is under the control of the server.

Interpretation by URI Pattern

The following table describes how expand is interpreted for each URI pattern. For those URI patterns where fields doesn't apply, it is silently ignored if present in the URL.

URI Pattern	Applicable?	Comments
/types	No	
/types/{typeName}	No	
/types/{typeName}/ hierarchy	No	
/types/{typeName}/ PR_Create	No	
/types/{typeName}/ instances	Yes	Since all the resources in the response are of the same type, then this use of expand is very useful to generate consistency in the way relationships are represented in the response. Since a large number of resources may be returned in the response, specifying expand=* or a large list of named relationships in the value of expand may cause the response to become very large.
/instances	Yes	Although expand applies to this URI pattern, the heterogeneity of resource type returned means that for many resources, the relationships named in the query parameter may not match relationships defined for many instances. It is not a problem if a relationship named in the expand does not appear in any given resource, it simply means that a relationship with that name is unavailable to expand with an in-line feed.

URI Pattern	Applicable?	Comments
/instances/{id}	Yes	Most useful to avoid round trip of an additional GET operation to retrieve a feed of related resources, in addition, because this operation returns a singleton, the chances of the response becoming very large is not as great as with URIs that return collections, like /types/{typeName}/instances
/instances/{id}/ relationships	No	
/instances/{id}/ relationships/{relName}	Yes	Expansion of relationship representations applies to the representation of the related resources

orderby

The orderby query parameter allows the consumer to control the order of appearance of atom:entries within a "conceptual feed".

Consider the following "conceptual feed":

```
<atom:feed>
<atom:entry>
<attr1>A
<attr2>10
<atom:entry>
<attr1>B
<attr2>9
<atom:entry>
<attr1>C
<attr2>8
<atom:entry>
<attr1>D
<attr2>7
<atom:entry>
<attr1>E
<attr2>6
<atom:entry>
<attr1>F
<attr2>5
</atom:feed>
```

The orderby query parameter allows the consumer to specify the "sort order" of the entries. For example:

```
GET /{URL to conceptual feed}orderby=attr1%20DESC
```

would result in the following response:

```
<atom:feed>
<atom:entry>
<attr1>F
<attr2>5
<atom:entry>
<attr1>E
<attr2>6
```

```
<atom:entry>
<attr1>D
<attr2>7
<atom:entry>
<attr1>C
<attr2>8
<atom:entry>
<attr1>B
<attr2>9
<atom:entry>
<attr1>A
<attr2>10
</atom:feed>
```

Note that the atom:entries appear in sorted order by the value of the "attr1" property in descending order.

The orderby query parameter takes as value a comma separated list of "sort specifiers". Each "sort specifier" is composed of a string name followed optionally by a direction indicator. The string name identifies an attribute name property of a resource. The direction indicator is either "ASC" or "DESC". If there is no direction indicator within a "sort specifier", the default value is "ASC".

Because the value of orderby is a comma separated list of these sort specifiers, it is possible to specify nested collating sequences.

For example, orderby=attr1%20ASC,%20attr2%20DESC,attr3,attr4, (note the URL encoding of the whitespace) would cause the atom:entry elements to be first sorted by the value of the resource property "attr1" (in ascending sequence) and within that, sorted by the value of att2 (in descending sequence), and within that sorted by attr3 and then by attr4.

If a "sort specifier" contains a "direction indicator" with a value other than "ASC" or "DESC" (or their lowercase equivalents) then the EDAA must return an error response, with http code 400 (bad request).

If a "sort specifier" contains a name of an attribute that does not appear within a given resource, then, for the purposes of sorting, value should be considered NULL, and the atom:entry corresponding to that resource should appear in the collation sequence as if the resource represented by the atom:entry had an attribute with the given name and the value of that attribute was NULL.

If an orderby query parameter is not specified in a request, the EDAA implementation is free to return the atom:entry elements in whatever sequence it chooses, but it MUST be consistent in the collation sequence applied to atom:entry elements in absence of an orderby query parameter.

The orderby query parameter is very similar to a SQL "ORDERBY" clause.

Interpretation by URI Pattern

The following table describes how orderby is interpreted for each URI pattern. If the orderby query parameter appears on a URI patterns for which orderby is not applicable, then the EDAA MUST return an error with http code 400 (bad request).

URI Pattern	Applicable?	Comments
/types	Yes*	The attribute model is fixed. The set of attributes upon which types feeds can be sorted is "typeName".
/types/{typeName}	No	
/types/{typeName}/hierarchy	No	
/types/{typeName}/PR_Create	No	
/types/{typeName}/instances	Yes	Since all the resources in the response are of the same type, it is easy for the consumer to specify useful sorting order with orderby.
/instances	Yes	Not terribly useful because there will be few property names that are shared amongst all types of resource instance, for any value of orderby, there will be many resources that do not contain one or more of the attributes specified in the "sort specifiers" defined in the value of orderby, resulting in many atom:entries being sorted by NULL values for those attributes.
/instances/{id}	No	
/instances/{id}/relationships	No	As with /instances/{id}, above
/instances/{id}/relationships/{relName}	Yes	Applies to the related resource representations contained in the response. Since this request is focused on a single relationship, the resources in the response are all of the same type, and therefore a reasonable sorting of those atom:entries can be specified by orderby.

filter

The filter query parameter has functionaly analogous to a SQL WHERE clause. The idea with filter is to allow the consumer to specify a filter expression, composed of boolean predicates that are applied against potential resources and acting as a filter so that only those resources that cause the filter expression to evaluate true are represented in the response.

Consider the following "conceptual" atom:feed:

```
<atom:feed>
<atom:entry>
<attr1>A
<attr2>10
<atom:entry>
<attr1>B
<attr2>9
<atom:entry>
<attr1>C
<attr2>8
<atom:entry>
<attr1>D
<attr2>7
<atom:entry>
<attr1>E
<attr2>6
```

```
<atom:entry>
<attr1>F
<attr2>5
</atom:feed>
```

The filter query parameter allows the consumer to select a subset of the atom:entry elements. For example:

```
GET /{URL to conceptual feed}filter=attr2 LT 8
```

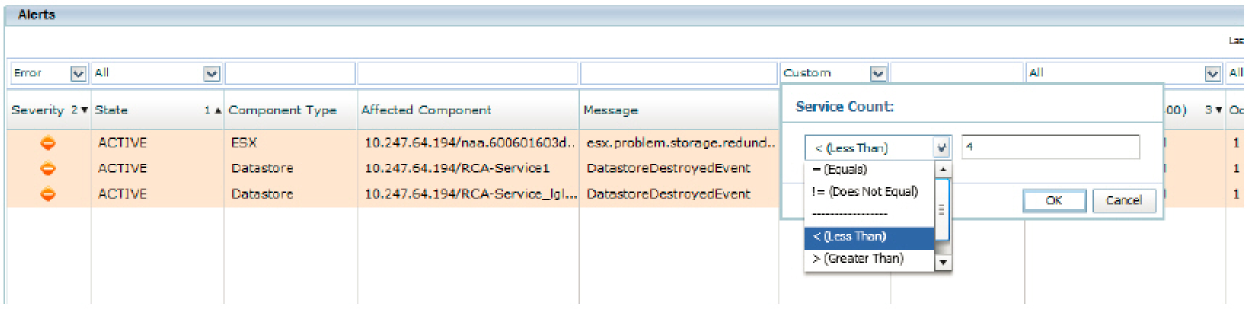
would return a subset of the atom:entry elements for which the expression "attr2 less than 8" evaluates true:

```
<atom:feed>
<atom:entry>
<attr1>D
<attr2>7
<atom:entry>
<attr1>E
<attr2>6
<atom:entry>
<attr1>F
<attr2>5
</atom:feed>
```

Note, the filter expressions presented in this section MUST be url encoded in practice. They are presented here using un-encoded syntax for readability. For example, the filter expression shown above would properly appear as

```
GET /{URL to conceptual feed}filter=attr2%20LT%208
```

This mechanism is useful, for example, in building UIs that allow the end-user to choose expression(s) on properties to be displayed in a table. Smarts, for example, provides a mechanism to filter a table of Alert resources by providing UI widgets allowing the user to form boolean expressions on any/all columns. An example screen shot is shown below:



Alerts

Severity 2 ▾	State 1 ▴	Component Type	Affected Component	Message		00) 3 ▾	Oc
Error	All				Custom	All	All
⊖	ACTIVE	ESX	10.247.64.194/naa.600601603d..	esx.problem.storage.redund..			1
⊖	ACTIVE	Datastore	10.247.64.194/RCA-Service1	DatastoreDestroyedEvent			1
⊖	ACTIVE	Datastore	10.247.64.194/RCA-Service_lgl...	DatastoreDestroyedEvent			1

Service Count: < (Less Than) 4

With the filter query parm feature of EDAA, the UI shown above could form predicates and use the server to do the resource filtering.

If the consumer submits a request containing a filter query parameter with value that does not conform to the filter expressions described below, then the EDAA implementation MUST reject the request, returning an http error code 400 (bad request).

Filter Expressions

The value of a filter query parameter is a filter expression as described in this section.

Filter expressions are boolean predicates expressed against the attribute properties of a resource. For example, consider the following VS-XML definition of a FileServer type (/types/FileServer):

```
...
<link href=".../types/FileServer" rel="self"></link>
...
<entry><title type="text">FileServer</title>
...
<content type="application/xml">
<type:Type xmlns:type="http://schemas.emc.com/vs-xml/namespace/Common/1.0" xmlns:atom="http://
www.w3.org/2005/Atom">
<type:typeName namespace="http://schemas.emc.com/vs-xml/namespace/ip/1.0">FileServer</
type:typeName>
...
<type:attribute type="xs:string" minOccurs="0" maxOccurs="1">IsManaged</type:attribute>
...
```

If a consumer wanted a collection of only FileServer instances that have value of IsManaged as true, then they could use the filter query parameter to express this constraint:

```
GET /types/FileServer/instancesfilter=IsManaged eq true
```

The idea with filter is that the type definition for a resource type (as returned by /types/{typeName}) defines a collection of attribute properties. Those properties that:

1 have @type as a simple type (eg xs:string, etc.), and

2 have @maxOccurs as "1"

can participate in a filter expression.

The syntax of the filter query parameter is a "filter_expr" as defined in the following(semi-formal) BNF:

```
filter_expr ::= bool_expr | filter_expr 'or' bool_expr
bool_expr ::= pexpr | bool_expr 'and' pexpr
pexpr ::= bool_pred | ‘(‘ filter_expr ‘)‘
bool_pred ::= simple_pred | 'not' pexpr
simple_pred ::= property_name rel_op term | property_name ‘in’ ‘(‘ in_list ‘)’ |
property_name 'lk' like_term
rel_op ::= 'eq' | 'ne' | 'gt' | 'ge' | 'lt' | 'le' //equals, not equals, greater than,
greater than or equal, less than and less than or equal to
in_list ::= string_lit | in_list ‘,’ string_lit
like_term ::= string_lit
The “like_term” is a string literal used with the lk operator; it can include a leading or
trailing % wildcard to match zero or more characters. % is encoded in a URI as
```

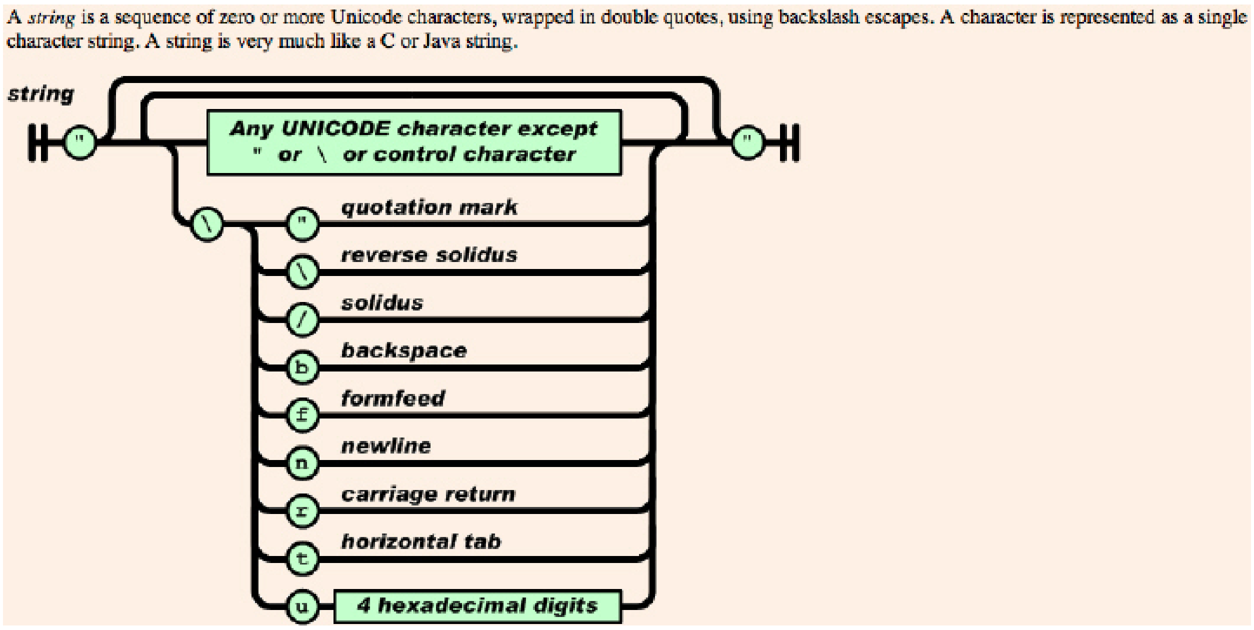
```
“%25”.
and property_name is a string literal corresponding to an attribute property of a resource
type meeting the constraints described above
and term is a valid string serialization of a value within the range of the simple type
associated with the property defined by the property_name in
the simple_pred expression. See terms below.
```

filter expression terms

Filter expressions are built by combining simple predicates of the form "property name" "operator" "term". The set of valid values for "property name" are defined by the resource model associated with the EDAA implementation. The set of operators is defined above ('eq' , 'ne' , 'gt' , 'ge' , 'lt' , 'le', 'in' , 'lk'). The valid value for "term" depends somewhat on the operator.

Terms are values with simple type. In the filter syntax, we use the simple syntax for primitive terms defined by JSON . Specifically, a term can be a string, number, or any of the following literal values: true, false, null.

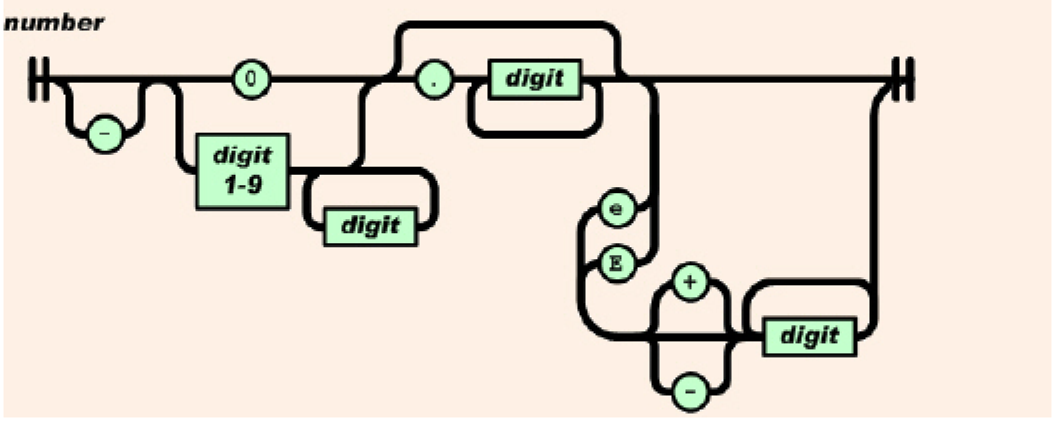
A string is defined by JSON as:



A *string* is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.

A number is defined by JSON as:

A *number* is very much like a C or Java number, except that the octal and hexadecimal formats are not used.



Dates, date/time and timestamps should be represented syntactically using ISO 8601.

For the 'in' operator, the "term" is a list of string terms enclosed by parenthesis. The semantic of this operation is that a predicate evaluates to true for a given resource, if the value of the referenced "property name" is equal to (string match) any of the string literals listed in the "term" of the predicate. For example, if a resource has the property "foo" with value "bar", then the following expression:

```
filter=foo in ("baz", "bar", "blee")
```

would evaluate to true.

For the 'lk' operator, the term is essentially a string term, however the string term can be prepended or appended with a '%' character. The '%' matches any number of any characters. The predicate with the 'lk' operator evaluates to true for a given resource using a string pattern match evaluation on the "property name" value for the resource. For example, if a resource has the property "foo" with value "aaaaabarbbbbbbb", then the following expression:

```
filter=foo lk "%bar%"
```

would evaluate to true.

filter expression operator precedence

In a filter expression, operators can be combined using 'AND', 'OR', 'NOT' and parentheses to yield complex expressions. For example, a resource type T defines 3 properties (p1, p2 and p3). A consumer wishes to filter a collection of resource instances of type T, specifically retrieving only those instances where p1 has value 'a', p2 has value 'b' and p3 has the value either 8 or 9. In order to achieve this, the consumer would make the following request:

```
GET /types/T/instancesfilter=p1 eq "a" AND p2 eq "b" AND (p3 eq 8 OR p3 eq 9)
```

Note the use of the parentheses for the expression predicate involving p3. The AND expression has higher precedence than OR. Had the consumer not used parenthesis here, the filter would not return the desired subset of instances.

Operation ordering in the filter expression is similar to most programming languages and query languages such as SQL. The operator precedence is defined as:

Level	Operators
1	expressions in parentheses (pexpr)
2	'eq' , 'ne' , 'gt' , 'ge' , 'lt' , 'le', 'in' , 'lk'
3	NOT
4	AND
5	OR

Note that of the following requests:

1

```
GET /types/T/instancesfilter=p1 eq "a" AND p2 eq "b" AND p3 eq 8 OR p3 eq 9
```

2

```
GET /types/T/instancesfilter=(p1 eq "a" AND p2 eq "b" AND p3 eq 8) OR p3 eq 9
```

3

```
GET /types/T/instancesfilter=p1 eq "a" AND p2 eq "b" AND (p3 eq 8 OR p3 eq 9)
```

The first two requests produce the same subset of instances, whereas the third request produces a different subset.

Missing Properties and Filter Expressions

Note that some resources may not contain values for properties defined in their type. For example, if a property is declared as minOccurs=0, any given instance of that type may or may not contain a value. What is the semantic of a filter expression referencing that property?

Consider the following situation, involving a type T and two attributes, one of which is defined with minOccurs=0:

```
<vsc:Type ...
<vsc:typeName ...>T</vsc:typeName>
...
<vsc:attribute minOccurs="1" ...>p1</vsc:attribute>
<vsc:attribute minOccurs="0" ...>p2</vsc:attribute>
...
</vsc:Type>
```

A GET on /types/T/instances may result in a collection that contains resources of type "T", some of which may not have a value for property p2.

A GET on /types/T/instancesfilter=p2 eq "foo" poses an interesting challenge. For those instances that contain a value for p2, the semantic is clear, evaluate the predicate against the value of p2 and include that instance in the response collection if the predicate evaluates to true. For an instance that does not contain a value for p2, the predicate MUST evaluate to false.

In general, if a filter expression contains a reference to a property not present in a given instance, that predicate MUST evaluate to false.

Interpretation by URI Pattern

The following table describes how filter is interpreted for each URI pattern. If the filter query parameter appears on a URI patterns for which filter is not applicable, then the MSA MUST return an error with http code 400 (bad request).

URI Pattern	Applicable?	Comments
/types	Yes*	The property model is fixed. The set of properties upon which types feeds can be filtered is "typeName".
/types/{typeName}	No	
/types/{typeName}/hierarchy	No	
/types/{typeName}/ PR_Create	No	
/types/{typeName}/instances	Yes	Since all the resources in the response are of the same type, it is easy for the consumer to specify useful filter expressions.
/instances	No	Not terribly useful because there will be few property names that are shared amongst all types of resource instance, for any filter expression appearing in filter will, for many resources, result in an illegal filter expression, thereby making the probability of having a request actually return successfully very small.
/instances/{id}	No	
/instances/{id}/relationships	No	As with /instances/{id}, above
/instances/{id}/relationships/ {relName}	Yes	Applies to the related resource representations contained in the response. Since this request is focused on a single relationship, the resources in the response are all of the same type, and therefore a reasonable filter expression can be formed.

languages

This query parameter allows a URL-level mechanism to control which language/locale the consumer would prefer the EDAA to use when localizing responses.

This query parameter is a convenience mechanism, useful for experimenting in a browser window or when the client technology (such as Flex/Flash) makes it difficult to manipulate http headers. The functionality is duplicated with Language Negotiation in EDAA using the http Accept-Language: header to specify consumer preference of localization. It is preferred that the consumer use http Accept-Language: headers to express language/locale preference.

The range of value for languages is a string. The format of the string is exactly that specified for http Accept-Language header. The value of languages is a comma separated set of language/ locale tags with an optional quality (preference) value. For example, languages="da, en-gb;q=0.8, en;q=0.7", expresses the consumer's preference to have responses localized to Danish, and if that is not possible, will accept responses localized to British English, or (with slightly less preference) any English dialect.

If an invalid value of languages is specified in a URL, the EDAA MUST respond with an error, http code 400 (bad request).

If none of the acceptable languages specified by the consumer are supported by the EDAA, then the EDAA MUST respond with an error, http code 406 (Not Acceptable).

As mentioned previously, the consumer can express localization preference using languages or an http Accept-Languages: header. In the case where the consumer uses both approaches, AND the approaches conflict (eg there is no language/locale that appears in both lists of preference), then the EDAA MUST return an error, 406 (not acceptable).

If no form of localization preference is expressed by the consumer, an EDAA implementation is free to choose which language/locale to use.

If a valid value of languages is specified in the URL, there is no conflict with an http Accept-Language: header in the request and at least one language/locale is supported by the EDAA, then the EDAA MUST format the response to the request using a supported language/locale format specified as being most preferred by the consumer.

If no form of localization preference is expressed by the consumer, an EDAA implementation is free to choose which language/locale to use.

Interpretation by URI Pattern

The languages query parameter may appear on any URI pattern specified in EDAA.

Consumer choice

When a consumer is creating requests, there are two mechanisms it can use to specify a preference for which data serialization format to use:

1. the alt query parameter on a request URL, or
2. an Accept: header of the http request.

For example, if a consumer wishes to process responses using JSON, then it could issue the following request:

```
GET /instances?alt=json
```

If the EDAA receiving the above request supports the JSON serialization format, it would return a collection of instance representations in JSON. EDAA conventions on JSON describes how collections of resource instances are represented in JSON in EDAA.

If the EDAA doesn't support JSON, it will return an error, http code 400 (bad request).

An equivalent request can be made using an http header:

```
GET /instances
Accept: application/json
```