opentext™

OpenText™ Documentum™ Content Management

Version 25.2

Foundation REST API Data Access API

OpenText™ Documentum™ Content Management Foundation REST API
Data Access API

EDCPKRST250200-ADA-EN-01
Rev.: 2025-Mar-19

This documentation has been created for OpenText™ Documentum™ Content Management Foundation REST API CE 25.2.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product, on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677
Support: https://support.opentext.com
For more information, visit https://www.opentext.com

Data Access API Specification Root

This topic is the Data Access API Specification. This information is available for general publication.

Contents

1 (Non-Normative) Introduction
2 (Non-Normative) Audience
3 Normative statement
4 EDAA - Data Access API - Common Style of REST API to access Data from OpenText Products

(Non-Normative) Introduction

See EDAA Primer, specifically the first chapter (Introduction to EDAA)

(Non-Normative) Audience

The audience for the EDAA specification is software developers who wish to understand more about EDAA. This document is primarily targeted at developers building EDAA implementations. Although this document also provides some value to developers on product teams building software consuming EDAA implementations, a related document, the Primer, was composed with that audience in mind.

This document assumes the reader is familiar with REST. If you are not familiar with REST, then please examine the myriad descriptions of REST that can be found on the Web.

Normative statement

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119, as scoped to those conformance targets.

Certain sections of this document are explicitly marked as "non-normative". These non-normative sections of the document provide illustrations of the specification and if there is any contradiction with normative sections of this document, the normative sections are authoritative.

EDAA - Data Access API - Common Style of REST API to access Data from OpenText Products

This section contains a collection of related topics that in aggregate represent the EDAA specification.

1. Base EDAA
2. EDAA Read/Write

Base EDAA Spec

This topic is a normative specification of "base" Data Access API.

Contents

1 See Also
2 Resource Metamodel
  2.1 Modeling Resource Types using VS-XML
  2.2 Representing Relationships
    2.2.1 Expanded Relationships
3 URI Patterns
4 Optional Query Parameters
  4.1 Query Parameters and URI Patterns
5 Responses
  5.1 Error Resource
  5.2 Example XML Error Response
  5.3 Example JSON Response
6 Use of HTTP Headers
  6.1 Language Negotiation
  6.2 Content Negotiation
  6.3 Version Negotiation
  6.4 ETag
  6.5 ETag in feeds and entries
    6.5.1 JSON Equivalent of Feed and Entry eTags
    6.5.2 Using eTag in a GET
    6.5.3 Using eTag in a PUT or PATCH operation
7 EDAA and Type Namespaces
8 JSON Format

See Also

Related links:

- Primer may be a good place to begin
- reading. EDAA Read/Write

Resource Metamodel

Resources in EDAA have the following aspects:

Resource Identifier

A resource MUST be identified by a unique identifier that distinguishes a resource from all other resources known to an EDAA.

Resource type information

A resource MUST be associated with a resource type resource. The resource type resource defines a set of potential actions, events, attributes and relationships that MAY appear with any resource associated with that type.

Actions

A resource MAY be associated with 1 or more actions. An action describes some sort of function or operation that can be performed on that resource. Note at any given point in time, a resource may expose a different set of actions than what is described in the resource's type

Attributes

A resource MAY be associated with 1 or more attributes. An attribute is an information item containing a piece of information about a resource.

Relationships

A resource MAY be associated with 1 or more relationships. A relationship represents a collection of resources related to a given resource by some named relationship.

Modeling Resource Types using VS-XML

Resource models are described using a syntax called VS-XML. Type information is available using the /types URI pattern and related patterns. The type model supported by EDAA is described here.

Representing Relationships

Relationships are represented by atom:link elements, or their JSON equivalent.

To assist tools and other consumers to determine the "kind" of atom:link being used, the following uri patterns MUST be used when the atom:link represents a relationship or an action (see see verb-style actions).

1. Relationship @rel URI pattern:
   - rel = "{common}/{domain-name-space}/{type}/relationship/{rel_name}"
2. Actions @rel URI pattern:
   - rel = "{common}/{domain-name-space}/{type}/action/{action_name}"

We will be reserving the URI prefix http://schemas.emc.com/msa for namespaces related to common, cross-domain EDAA names.

Examples:

<atom:link rel="http://schemas.emc.com/msa/UIM/ZoneSet/relationship/hasZones"

href=".../instances/ZoneSet::123/relationships/hasZones"></atom:link>

<atom:link rel="http://schemas.emc.com/msa/UIM/ZoneSet/action/activate" href=".../instances/ZoneSet:789/action/activate"></atom:link>

The URI used for the value of @rel may be derived from one of several sources:

- Common relationships
  - e.g. relationships defined by the EDAA metamodel, such as TypeHierarchy

- For relationships defined in a Type resource, the URI to the relationship is formed by concatenating the fully qualified type of the resource with the name of the relationship.
  - For example, the FileServer type is defined in VS-XML by:
    - <type:typeName namespace="http://schemas.emc.com/vs-xml/namespace/ip/1.0">FileServer</type:typeName>
    - Within that type definition, there are many relationships defined, including
      - <type:relationship relType="ICIM_Connection">ConnectedVia</type:relationship>
    - Within a representation of a FileServer instance, the ConnectedVia relationship would appear as the following atom:link:
      - <atom:link rel="http://schemas.emc.com/vs-xml/namespace/ip/1.0/FileServer/ConnectedVia" href="https://.../instances/FileServer::nasp02.lss.emc.com/relationships/ConnectedVia/>

Expanded Relationships

Relationships may be extended to include an in-line representation of the collection of related resources. This mechanism is detailed in another section of the base EDAA specification.

URI Patterns

One way to think about EDAA is in terms of a set of URI patterns. We note that, due to Hypermedia As The Engine of Application State or HATEOAS, these patterns should not be considered a contract with the consumer of the EDAA. Rather, we articulate these URI patterns in terms of forming a structure to help a product developer adapt the EDAA style to the REST API he/she is building for a product.

The "base" EDAA describes these URI patterns in terms of read-only (GET) operations. The relationship of these URI patterns to other operations in the REST uniform interface (PUT, POST, DELETE, etc.) is described in EDAA Read/Write.

The following table summarizes the URI patterns in EDAA and outlines which patterns MUST or SHOULD be implemented.

Note, the URI patterns described here use the URI template notation.

EDAA	Required	Description
/	SHOULD	A resource describing the capabilities of the EDAA. The resource SHOULD correspond to an Atom Publishing Protocol Service Document.
/types	MUST	A collection of resource type resources, one resource type resource for each type supported by the EDAA. See VS-XML description for an explanation of the convention for representing resource types.
/types/{typeName}	MUST	A single resource type resource. The value of {typeName} is a string and it MUST correspond to the name of a type resource supported by the EDAA.
/types/{typeName}/hierarchy	MUST	A collection of resource type resources, one for each type in a type hierarchy starting with the type resource identified by {typeName}. The value of {typeName} is a string and it MUST correspond to the name of a type resource supported by the EDAA.
		A resource describing the partial representation of the type resource identified by {typeName} for the purposes of creating a new instance of that resource type. The

/types/{typeName}/PR_Create	SHOULD	value of {typeName} is a string and it MUST correspond to the name of a type resource supported by the EDAA. See Partial Representation for Create.
/instances	SHOULD	A collection of resources representing all resources known to the EDAA implementation.
/types/{typeName}/instances	MUST	A collection of resources containing a representation of each resource instance of a given type, where the type is identified by {typeName}. The value of {typeName} is a string and it MUST correspond to the name of a type resource supported by the EDAA.
/instances/{id}	MUST	An individual resource identified by {id}, where {id} corresponds to an identifier of an individual resource known to the EDAA. Note that {id} corresponds to whatever identifier scheme defined by the EDAA's resource model. See the Primer for some details on what considerations apply to choice of {id}.
/instances/{id}/relationships	MUST	An individual resource identified by {id}, where {id} corresponds to an identifier of an individual resource known to the EDAA. The semantics of this pattern are exactly the same as /instances/{id}. This URI pattern is documented to maintain consistency with the ROA.
/instances/{id}/relationships/{relnName}	MUST	A collection of resources containing those resources related to an individual resource identified by {id}, where {id} corresponds to an identifier of an individual resource known to the EDAA, via the relationship identified by {relnName}. The type definition associated with the resource identified by {id} MUST contain a relationship definition defining a relationship named {relnName}.

For purposes of backwards compatibility with MSA 1.0 (a precursor REST API style), implementations SHOULD treat URIs with the pattern "/instances/byKey/{id}" and "/instances/byType/{typeName}" by using HTTP 301 redirect to the corresponding "/instances/{id}" and "/types/{typeName}/instances" URIs. EDAA implementations MUST NOT design implementations that allow "byKey" or "byType" to be used as identifiers for resources.

Optional Query Parameters

Here are the optional Query Parameters defined in EDAA. See also the Primer - Query parms and why they are useful for a description of how these query parameters can be used.

EDAA implementations SHOULD support these query parameters:

Parameter	Meaning	Notes
		▪ These query parameters define a "chunking" or "paging" view over a large collection of resources in an atom:feed. ▪ Along with RFC 5005 Atom Pagination style atom:link elements, a consumer can understand related resources that contain other "pages" or "chunks" of the entire collection of resources.

	page and per_page	Pagination related	■ The per_page query parameter allows the consumer to control the maximum number of resources that may appear on any page or chunk of the collection. ■ The page query parameter allows the consumer, typically via RFC 5005 style atom:link elements to retrieve a specific page or chunk. ■ These query parameters were supported in MSA 1.0. ■ See Primer for more explanation.
	alt	Alternative format of the resource representation	■ If you don't specify an alt parameter 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. ■ alt=json returns a JSON representation of the feed, see EDAA JSON ■ Note: gdata also allows alt=rss, alt=json-in-script, alt=atom-in-script, alt=rss-in-script and alt=atom-service ■ The EDAA spec does NOT require a compliant implementation to support these values for the alt query parameter. ■ alt=csv returns a comma separated variable representation of the equivalent Atom feed. ■ Note: EDAA also supports the notion of using an HTTP Accept header to specify the expected content type of the response (see below). ■ See Primer for more explanation.
	fields	Specify the subset of fields (properties) of the resource that should be returned in the response	■ See see Gdata Partial response ■ The value of fields is a comma separated collection of field specifiers (attribute names or relationship names). ■ If a field specifier is given that does not correspond to a property name, relationship name or atom feed component, then it is ignored ■ See Primer for more explanation.
	expand	Specify 1 or more relationships to be "expanded" so that the representation of the resource includes a feed of related resources as child content of the relationship's link element	■ See Details on the expand query parameter ■ The value of the query parameter is a comma separated list of relationship names ■ For each relationship named in the query parameter value, the representation of that relationship is "expanded" to include an in line feed of related resources ■ This allows a consumer to avoid an additional round trip to retrieve related resources ■ See Primer for more explanation.
		comma separated list of properties the	■ inspired by odata

orderby	response should be sorted on. The response MUST present an atom:feed with the entries sorted by the values of the indicated fields.	■ The value of orderby is a comma separated list of property name and optional sort direction (ASC or DESC) tokens ■ See Primer for more explanation.
filter	simple boolean predicate expression to describe a filter, or subset, of the resources to return in a GET operation.	■ inspired by SQL "where" clause concept ■ similar to the odata $filter concept ■ see filter expressions below. ■ See Primer for more explanation.

languages	Alternative format of the language/locale preference	■ If you don't specify a language query parameter and no language preference is specified in an http Accept-Language: header, then the EDAA is free to choose which language/locale to use when localizing responses.
		■ The value space of this query parameter is exactly the same as described for the http Accept-Language header
		  ■ That is, a comma separated list of language preferences with optional quality value (q=...)
		  ■ 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.
		■ Note: EDAA also supports the notion of using an HTTP Accept-Language header to specify the expected content type of the response (see below).
		■ See Primer for more explanation.

Query Parameters and URI Patterns

This table summarizes which query parameter is applicable for the various URI patterns. If a cell in the table contains "Y", then the query parameter MAY appear in a request conformant to the URI pattern, if the cell is blank, then the query parameter MUST NOT appear in a request conformant to the URI pattern.:

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

Responses

EDAA implementations MUST use http status codes in responses to consumer requests, as specified in RFC 2616.

In addition, for those responses that correspond to an error (eg status codes in the 4xx and 5xx ranges) the content of the response message SHOULD contain a representation of a common Error resource as specified below. Note that the error representation need not appear within an atom entry or feed element.

Note, some EDAA implementations may additionally choose to surface Error as a resource, eg support GET /types/Error/instances. EDAA does not require nor does it prohibit EDAAs from doing this.

Error Resource

EDAA defines a common type, called Error, that contains a standard set of information items that describe and error condition. This type is defined within the http://schemas.emc.com/msa/common/ type namespace and has the name "Error". EDAA implementations MUST not define a type with the name equal to "Error" in any other namespace.

An Error needs to be associated with an identity. The identifier scheme is essentially opaque to the consumer and the EDAA implementation is free to choose any identifier scheme, as long as Error resource instances are uniquely defined amongst all instances known to the EDAA.

An EDAA declaring a type named "Error" within the http://schemas.emc.com/msa/common/ MUST define the type with the following properties:

Information model for Error

Property Name	Range/Type	Comment
Severity	integer	■ The range of values is a fixed set, the set is the 8 levels defined by RFC 5424 ■ Uis may be free to map this set into a different set ■ For severities coming from other systems that do not match the RFC 5424 severity level set, it is the responsibility of the creator of the Error resource to map those idiosyncratic severities into one of the levels defined by RFC 5424.
Type	URI	■ Identifies a domain-specific error type, useful for classifying an Error resource into a category of Error with a given semantic. ■ This URI SHOULD be resolvable to a web resource describing the type of error in more detail. ■ Each product MAY define error types in its product-specific namespace for errors specific to the product

ErrorCode	string	A brief code, typically an integer, that is associated with the type of Error. This is typically a product-specific code.
HTTPStatusCode	integer	The RFC 2616 HTTP Status Code returned with the response.
Message	string	■ brief human readable synopsis of the Error, to be displayed in UI to describe the Error to human consumers ■ A representation MAY contain one or more of these Message properties, each additional Message property MUST be associated with an additional lang attribute, as described in xml:lang tag or equivalent (for non-XML markup like JSON). ■ If the request associated with an Error resource specifies an Accept-Language HTTP header, the Error resource representation should include only those Message properties that correspond to the language(s) requested in the Accept-Language header. If no Accept-Language header was specified, then the EDAA is free to include as many (or as few) Message properties of the Error resource as it chooses.
Created	timestamp	time the Error resource was created, the timestamp must be generated according to RFC3339 and MUST include information (such as timezone offset) to allow the consumer to unambiguously map the property into a UTC form.
Request	string	Describes the request operation that caused the Error. For example "GET /instances/Foo::123" is an example value of a Request property.
RequestorAddress	string	IP address of the client that made the request.
RequestorIdentity	string	Username of the entity that issued the request, as authenticated by the security mechanism protecting access to the EDAA. ■ E.g. Identify the entity that issued the request with respect to the authentication authority, or the identity provider, i.e. a CAS referencing an LDAP server. ■ E.g. the username from the CAS ticket ■ Note, in the case where the error condition is one in which the consumer MUST authenticate before performing an action (ie the RequestorIdentity of the request is not known), then an EDAA MUST use a null value for this property.
Extensibility	any content outside of the http://schemas.emc.com/msa/common/ namespace	The mechanism by which additional name/value properties can be added to the Error information model

Note, with respect to the information in an Error representation, specifically in the case of authentication and authorization failures, the content is purely informational -- consumers MUST NOT use the information in an Error representation found in response to facilitate a

security handshake, or to convey security policy information in-band.

An EDAA MAY persist Error resource instances, but is NOT required to do so. If an EDAA chooses not to persist Error resources at all, or choose to persist Error resources for a brief period of time, then the EDAA MUST return an HTTP 410 (Gone) to any request addressing the given Error URL.

For example, if the consumer receives a representation of an Error, with identity Error::d57c9800-9b56-11e0-aa82-0800200c9a66. The consumer MAY choose to issue the following request:

```
GET /instances/Error::d57c9800-9b56-11e0-aa82-0800200c9a66
```

In the case where the EDAA does not persist Error instances, it MUST respond with an HTTP Status code of 410 (Gone). The EDAA SHOULD NOT generate another Error resource in the body of the 410 (Gone) response.

There are several serialization mechanisms available in EDAA, such as atom/xml or JSON. The data serialization format used to serialize an Error resource representation MUST correspond to the format requested by the user following the mechanisms outlined in Content Negotiation. If the consumer asked for the response to be atom/XML, then if an Error resource is created in response to the request, it MUST appear in the response payload in XML format. If the consumer specified JSON for the response, then if an Error resource is created in response to the request, it MUST appear in the response payload in JSON format.

The response MAY contain style-sheet information to render the Error response representation in a more human friendly format.

Example XML Error Response

Consider the situation where the following request is issued by an authorized identity labeled "sgg":

```
GET /types/Foo/instances
```

But the typeName "Foo" is unknown to the EDAA, generating a 404 not found style error condition. The following XML payload would be delivered in the response:

```
<Error xmlns="http://schemas.emc.com/msa/common/">
  <Severity>3<Severity>
  <Type>http://schemas.emc.com/msa/common/error/resource_not_found</Type>
  <ErrorCode>404</ErrorCode>
  <HTTPStatusCode>404<HTTPStatusCode>
  <Message xml:lang="en">The resource you are looking for is not found.</Message>
  <Message xml:lang="fr">La ressource que vous recherchez n'est pas trouvee.</Message>
  <Message xml:lang="es">El recurso que usted esta buscando no se encuentra.</Message>
  <Created>2011-06-20T11:59:29-05:00</Created>
  <Request>GET /types/Foo/instances</Request>
  <RequestorAddress>127.0.0.0</RequestorAddress>
  <RequestorIdentity>sgg</RequestorIdentity>
</Error>
```

Example JSON Response

The JSON encoding of the XML response above would look like:

```
{
```

```
    "Severity" : 3,
    "Type" : "http://schemas.emc.com/msa/common/error/resource_not_found",
    "ErrorCode" : 404,
    "HTTPStatusCode" : 404,
    "Messages": [
        {"en" : "The resource you are looking for is not found."},
        {"fr" : "La ressource que vous recherchez n'est pas trouvee."},
        {"es" : "El recurso que usted esta buscando no se encuentra."}
    ],
    "Created" : "2011-06-20T11:59:29-05:00",
    "Request" : "GET /types/Foo/instances" ,
    "RequestorAddress" : "127.0.0.0",
    "RequestorIdentity" : "sgg"
}
```

Use of HTTP Headers

This topic describes how EDAA uses http headers, specifically eTags, and the Accept header for API version and content 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).

Content Negotiation

EDAA implementations MUST process client use of the http Accept header.

EDAA SHOULD support application/atom+xml and application/json as alternative formats. The ECAA implementation MAY accept other mime types.

If the client request uses both the Accept header and an ?alt query parameter to express preference for the response format, then a response is given only if the value of the Accept header matches the value of the ?alt query parameter. For example a request:

```
GET /types?alt=atom
Accept: application/json
```

contains a conflict between the ?alt query parameter and the Accept: header. In this case, the server must return an error (406 not acceptable).

Note, it is possible, using "accept parameters" for the consumer to specify a set of acceptable content types ordered by preference. For example, the following request

```
GET ...
Accept: application/json; q=0.2, application/atom+xml; q=0.1, text/*
```

encodes the consumer's preference for a JSON serialization format, but indicates that atom/xml is acceptable (but less preferable) and that a plain text format is acceptable, but the least preferable.

In cases where multiple content types appear in an Accept: header, the EDAA implementation MUST attempt to use the format most preferred by the consumer. If a request like the one shown above is received by an EDAA that only supports the application/atom+xml format, then the EDAA MUST attempt to return a response formatted as atom/xml.

If a valid content type is requested in the Accept: header, there is no conflict with an ?alt query parameter in the URL 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 Accept: header.

For the value in the Accept: header of "application/atom+xml", the serialization of the response is governed by the RFC 4287 - Atom Syndication Format and rules to represent type resources and instance resources specified in EDAA.

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

An EDAA implementation is free to support other Content types as serialization format, for example GZIP, or CSV, or even plain text (text/*).

Version Negotiation

Some REST APIs encode version number in the URI scheme. This is generally considered a bad idea. See here for a discussion.

The preferred approach is to have consumers specify version scheme in an HTTP Accept: header, if version makes a difference. Specifically, using a version={version id} accept-parameter. For example, if the user specifically wanted the legacy MSA 1.0 Atom feed, it would specify:

```
GET /...
Accept: application/atom+xml;version=1.0
```

For many consumers, the version is not relevant, so no version= accept-parameter need be specified. This is generally a bad idea, since consumers SHOULD care about the service version. Consumer-facing documentation (user guides, programmer guides, examples, etc.) should STRONGLY RECOMMEND that consumers specify Accept: header and version information.

When an EDAA implementation receives a request with no Accept: header, or an Accept: header that does not specify version, then the EDAA implementation is free to serve whatever version of the service it wants. Basically if the consumer indicates it doesn't care

about the version, than the EDAA implementation is free to choose.

Note that HTTP Accept: header is also used for content negotiation, so the actual value used by the consumer may contain a combination of version preference and serialization format preference. The following table indicates the possible consumer preferences and how they would be encoded in the Accept: header

	Any Format	atom/XML	JSON	Foo Format (fictitious)
Any Version	NO Accept: header	Accept: application/atom+xml	Accept: application/json	Accept: application/foo
MSA 1.0	Accept: application/atom+xml;version=1.0 (A)	Accept: application/atom+xml;version=1.0	Accept: N/A (A)	Accept: N/A (A)
EDAA	Accept: application/atom+xml;version=2.0 (B)	Accept: application/atom+xml;version=2.0	Accept: application/json;version=2.0	Accept: application/foo;version=2.
Vn	Accept: application/atom+xml;version=n (B)	Accept: application/atom+xml;version=n	Accept: application/json;version=n	Accept: application/foo;version=n

Notes:

- (A) - MSA 1.0 supports only atom+xml
- (B) - Default format is atom+xml, unless otherwise specified

ETag

The ETag header is a commonly used feature of http. Two major benefits: cache control and optimistic locking. Although cache control is a "nice to have" feature, and is likely very important performance enhancement for mashups, the optimistic locking feature is critical for read/write support in EDAA.

From the ETag wikipedia entry:

> An ETag, or entity tag, is part of HTTP, the protocol for the World Wide Web. It is one of several mechanisms that HTTP provides for cache validation, and which allows a client to make conditional requests. This allows caches to be more efficient, and saves bandwidth, as a web server does not need to send a full response if the content has not changed. ETags can also be used for optimistic concurrency control as a way to help prevent simultaneous updates of a resource from overwriting each other.
>
> An ETag is an opaque identifier assigned by a web server to a specific version of a resource found at a URL. If the resource content at that URL ever changes, a new and different ETag is assigned. Used in this manner ETags are similar to fingerprints, and they can be quickly compared to determine if two versions of a resource are the same or are different. Comparing ETags only makes sense with respect to one URL—ETags for resources obtained from different URLs may or may not be equal and no meaning can be inferred from their comparison.

Therefore EDAA implementations SHOULD support ETag. The EDAA implementation is responsible for generating the eTag "fingerprint" for a resource and regenerating a different fingerprint when the underlying data about the resource changes in way that is visible to the consumer. The EDAA is free to use whatever algorithm makes sense for generating and regenerating the eTag.

Common approaches are to use a "last modified" timestamp of when the resource was last modified, or a hash of the property values of the resource.

Client code SHOULD utilize this ETag value in subsequent GET requests to the resource (using an If-None-Match header), and be prepared to accept an HTTP 304 Not Modified status.

Before any PUT/POST or DELETE operations, clients SHOULD refetch a representation of the resource, and proceed only if the response is an HTTP 304 value.

ETag in feeds and entries

Since EDAA responses are formatted as an atom:feed (or its JSON equivalent), containing collections of resource representations in atom:entry elements, it would be even more useful if eTags were associated with feeds and entries.

Google's gdata has a particular approach to decorating feeds and entries with etags, described here.

For those EDAA implementations that support eTags, eTags MUST appear in the the feed and entry level as well as in the http headers.

JSON Equivalent of Feed and Entry eTags

See eTags in EDAA JSON.

Using eTag in a GET

For this discussion, 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 discussion, 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.

EDAA and Type Namespaces

The following statements are normative with respect to the relationship between types namespaces and an EDAA implementation

- An EDAA may surface types from one or more namespaces
- The developer of the EDAA MUST ensure that there are no two type resources with the same name across all types surfaced by the EDAA (ie ensure {typeName} is unambiguous)
  - If a URI pattern production {typeName} resolves to more than 1 type resource, an http 500 (server error) MUST be returned
- Types in the common namespace (Task, Error, etc.) supersede types from any other namespace

JSON Format

Please see Public_EDAA_JSON.

Base EDAA Spec - VS-XML

Contents

1 Using VS-XML to Represent a Resource Type
2 VS-XML Syntax
  2.1 type declaration element
  2.2 typeName declaration element
  2.3 link element
  2.4 attribute declaration element
  2.5 relationship declaration element
  2.6 action declaration
  2.7 other attributes and elements
3 Other possible Metadata formats

Using VS-XML to Represent a Resource Type

The VS-XML language was developed in 2008 as part of the MSA 1.0 effort (a pre-cursor to EDAA). The name, VS-XML, stands for "View Services XML", where "View Services" was the name of the project under which MSA 1.0 was developed.

VS-XML is an XML Schema Definition (XSD) like language for representing the important properties of a resource type:

- name of the type
- namespace of the type
- parent type (eg super class)
- attributes of a type
- relationships of a type
- other information, such as what actions may be done to instances of that type.

Here is an example VS-XML representation of a resource type:

The idea is that the product team would define a resource model and then define the properties of each attribute. Based on some description of these resource types (eg in UML, SQL database or something else) and then generate the VS-XML from that basic representation. Of course, teams could simply handcraft the VS-XML and store it in a database somewhere.

VS-XML Syntax

VS-XML syntax is defined in the VS-XML namespace: http://schemas.emc.com/vs-xml/namespace/Common/1.0.

VS-XML MAY appear as XML markup or as JSON as described in subsections below.

A VS-XML type declaration consists of exactly 1 type declaration element.

type declaration element

The type declaration element in VS-XML is a relatively empty tag that serves as a common parent element for the entire type declaration. This tag also is a convenient spot for declaring namespaces used in the remainder of the markup.

Example

```
<type:Type xmlns:type="http://schemas.emc.com/vs-xml/namespace/Common/1.0" xmlns:atom="http://www.w3.org/2005/Atom"
     description="A brief description of the type"  documentation="http://some-url-to-documentation">
```

A type declaration has the following attributes:

- zero or one description attributes
- zero or one documentation attributes

A type also has a name and the name is scoped within a declared namespace.

A type declaration element has as child elements:

- exactly 1 typeName declaration element
- zero or more link elements
- zero or more attribute declaration elements
- zero or more relationship declaration elements
- zero or more action declarations
- zero or more elements or attributes from a namespace different than the VS-XML namespace

In JSON, a type is serialized as:

```
{
  "name": <<string representing the name of the type>>,
  "namespace": <<URI identifier of the namespace in which this type is being defined>>,
  "description": <<OPTIONAL brief sentence or two describing the type>>,
  "documentation": <<OPTIONAL URL to a web resource providing more documentation on the type>>
  "links: [
      <<link objects for "self" and the parent type and the collection of types associated within the type hierarchy>>
  ],
  "attributes": [... ],
  "relationships":[ ... ],
  "actions": [...],
  <<other attributes, if any, that the author wishes to add>>
}
```

typeName declaration element

The typeName declaration has two major components, a namespace declaration that defines the namespace of the type being defined and element content that defines the name of the type.

```
<type:typeName namespace="http://schemas.emc.com/vs-xml/namespace/ip/1.0">ICIM_RelayDevice</type:typeName>
```

The namespace declaration appears in the value of @namespace, it is a URI identifier of the namespace.

The name of the type is the content of the element, it must be an NCName and unique within the declared namespace.

or in JSON, the information is already covered by the type declaration.

link element

Link elements are used within a type declaration to associate the type with other (related resources).

Here is an example of some links that may appear in a type declaration:

```
 <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">ICIM_RelayDevice</type:typeName>
      <atom:link rel="self" href="https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/ICIM_RelayDevice"/>
      <atom:link rel="http://schemas.emc.com/msa/common/reln/hierarchy"
                 href="https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/ICIM_RelayDevice/hierarchy" />
      <atom:link rel="http://schemas.emc.com/msa/common/reln/parent"
                 href="https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/UnitaryComputerSystem"/>
      <atom:link rel="edit" href="https:.../types/ICIM_RelayDevice/instances"  />
```

or JSON:

```
{
  ...
  "links: [
        {
          "rel": "self",
          "href": "https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/ICIM_RelayDevice"
        },{
          "rel": "http://schemas.emc.com/msa/common/reln/hierarchy",
          "href": "https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/ICIM_RelayDevice/hierarchy"
        },{
          "rel": "http://schemas.emc.com/msa/common/reln/parent",
          "href": "https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/UnitaryComputerSystem"
        },{
          "rel": "edit",
          "href": "https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/ICIM_RelayDevice/instances"
        }
  ],
...
```

Typical uses of link include:

type's superclass

rel = "http://schemas.emc.com/msa/common/reln/parent"

value of href is the URI to the type that is the immediate super type (parent type) of the resource

type's type hierarchy

rel = "http://schemas.emc.com/msa/common/reln/hierarchy"

value of href is the URI to the feed of types in the resource's type hierarchy

type's Create Partial Representation (PR_Create)

rel = "http://schemas.emc.com/msa/common/reln/PR_Create"

value of href is the URI to the partial type definition defining the subset of attributes and relationships that MUST appear in create operations

type's feed of Instances

rel = "http://schemas.emc.com/msa/common/reln/instances"

value of href is a URI to the list of instances of the given type

new instances can be created by the consumer (instantiability)

rel = "edit"

if present, suggests that it is *possible* (but no guarantee) that instances of this type can be created by the consumer.

attribute declaration element

A VS-XML attribute declaration declares that an attribute with a given set of properties is part of a type definition.

Example:

```
<type:type ...
  <type:attribute type="xs:string" minOccurs="0" maxOccurs="1">OSVersion</type:attribute>
...
```

or in JSON

```
{
  ...
  "attributes": [
    {
      "name": "OSVersion",
      "type": "xs:string",
      "minOccurs": "0",
      "maxOccurs": "1",
      "default": "",
```

```
      "description": "optional sentence or two describing the type",
      "documentation": "http://optional-url-to-web-resource-describing-the-attribute"
    }, ... repeated once for each attribute in the type
  ],
...
}
```

An attribute declaration has seven properties:

name

an NCName that must be unique with in the type declaration. In XML, it appears as the content of the attribute declaration element.

type

a QName or URI defining the base type of the attribute.

minOccurs

Similar to the XSD @minOccurs, a string value that resolves to a non-negative integer. Defines the minimum number of times the attribute must appear in an instance resource of the type

maxOccurs

Similar to the XSD @maxOccurs, a string value that resolves to a non-negative integer or the literal "unbounded". Defines the maximum number of times the attribute must appear in an instance resource of the type, if the value of @maxOccurs is unbounded, there is no limit to the number of times the attribute may occur.

default

An optional property, that describes the value a consumer should assume the attribute takes if the attribute is absent from a representation. This property may appear only on attribute declarations that have @minOccurs="0" and that have @type corresponding to a simple type.

description

An optional property, a string containing a sentence or two description of the attribute.

documentation

An optional property, a URL to a web resource that contains more details about the attribute.

relationship declaration element

A VS-XML relationship declaration declares that a relationship with a given set of properties is part of a type definition.

Example:

```
<type:type ...
  <type:relationship relType="ICIM_Connection" type="http://schemas.emc.com/msa/common/reln/contains"
                     minOccurs="0" maxOccurs="unbounded"
                     description="Instances of ICIM_Connection are connected to ...">ConnectedVia</type:relationship>
...
```

or in JSON

```
{
  ...
  "relationships":[
    {
      "name": "ConnectedVia",
      "relType": "ICIM_Connection",
      "type": "http://schemas.emc.com/msa/common/reln/contains",
      "minOccurs": "0",
      "maxOccurs": "unbounded",
      "description": "Instances of ICIM_Connection are connected to ...",
      "documentation": "http://example.com/docs/ICIM_Connection.html"
    }, ... repeated once for each relationship in the type
  ],
}
```

A relationship declaration has six properties:

name

an NCName that must be unique with in the type declaration. In XML, it appears as the content of the attribute declaration element.

relType

an NCName that corresponds to the name of a type known to the EDAA. This defines the type of resource related via the relationship.

type

a URI declaring that the relationship follows some common semantic associated with relationships of this type. If this attribute is missing, or has an empty or "" value, then consumers should conclude that there is no "common" semantic associated with this relationship.

minOccurs

Similar to the XSD @minOccurs, a string value that resolves to a non-negative integer. Defines the minimum number of resources must be related via this relationship. If not specified, the default value for @minOccurs is "1".

maxOccurs

Similar to the XSD @maxOccurs, a string value that resolves to a non-negative integer or the literal "unbounded". Defines the minimum number of resources must be related via this relationship, if the value of @maxOccurs is unbounded, there is no limit to the number of resources that can be related via this relationship. If not specified, the default value for @maxOccurs is "1".

description

A string containing a human readable description or synopsis of the semantics of the relationship.

documentation

An optional property, a URL to a web resource that contains more details about the relationship.

action declaration

Per EDAA Read-Write spec an action declaration was added to VS-XML to allow the designer to indicate that a given "action" is typically associated with instances of the type. This is no guarantee that for any given instance of that type, these actions will be available, as the set of available links and actions may change depending on the authorization of the given user, the state of the resource, or other circumstances.

Example:

```
<type:type ...
  <type:action rel="edit" description="instances are mutable"/>
  <type:action rel="http://schemas.emc.com/msa/uim/VCenter/action/provision" description="provision the given instance"
               documentation="http://someWebServer.org/docs/VCenter/action/provision.html" />



...
```

or in JSON

```
{
  ...
  "actions" : [
      {"rel" : "edit",  "description" : "instances are mutable" },
      {"rel" : "http://schemas.emc.com/msa/uim/VCenter/action/provision", "description" : "provision the given instance",
       "documentation" : "http://someWebServer.org/docs/VCenter/action/provision.html"}
  ],
...
```

An action declaration contains a @rel value, that suggests that, for any given instance of the declared type, it is possible that an atom:link with that value of @rel may appear in the resource's representation.

Note that the values of @rel are either ncnames or URIs, following the semantics defined for atom:link element by Atom Syndication Format RFC4287.

The action optionally may include a human readable text description giving a human readable short synopsis of the semantics of the action.

An optional documentation attribute contains a URL to a more in-depth description of the action, including potentially description of the input and outputs of the action, detailed behavior description, WADL document, etc.

other attributes and elements

A VS-XML type declaration may contain attributes and elements from a namespace other than the VS-XML (http://schemas.emc.com/vs-xml/namespace/Common/1.0) namespace.

Other possible Metadata formats

At some point in the future, we will consider supporting other forms of type metadata other than VS-XML. We could consider an RDF form or perhaps a straight XML Schema Definition form. At that time in the future, we will add additional mechanism to allow the consumer to request which form it wants in the /types request.

Base EDAA Spec - Expand Relationships

Contents

1 Expanding Relationships
2 Problem Statement
3 Discussion
4 Normative Statements
    4.1 JSON representation of the inline feed construct

Expanding Relationships

EDAA represents relationships using atom:link elements. In certain situations, it is convenient to have relationship information represented with additional information as described in this section.

Problem Statement

1. A concern has been raised about the use of the atom:link to represent relationships. This causes too many round trips to retrieve a resource's representation and the identities of the related resources.
2. As it stands now, when one retrieves a resource's representation, the response is a set of atom:link elements that give the URI to a relationship resource (an href of the form: {base}/instances/{id}/relationships/{relnName}), this href can then be requested, returning an atom feed containing representations of the related resource.
3. Frequently what is desired (for purposes of displaying a table of related resource identifiers) is the list of the related resources. The current approach, to use the atom:link requires two round trips to the server and returns more information than is immediately necessary for these type of use cases
4. This section describes a mechanism in EDAA that allows a user to specify a representation of a given relationship that contains a list of related resources.

Discussion

The problem boils down to how to represent the information about what resources are related to a given resource via a set of relationships.

EDAA uses the notion of an atom:link, referencing the relationship itself as a resource. Each resource representation contains a set of atom:link elements, one for each relationship defined by the resource's type. Each relationship is represented as a resource itself, essentially the collection of related resources via a given named relationship. The pairwise "connection" between a resource and another resource via a relationship is called an "association". The collection of associations is embodied by a relationship resource.

For many use cases, a resource representation modelling relationships with just an atom:link referring to the collection of associations (ie the resource found via the /instances/{id}/relationships/{relnName} pattern) is sufficient. This is a terse way to identify the collection

of related resources.

For example in the EDAA representation of a Switch resource, as found in the UIM product:

```
GET types/Switch/instances
```

The EDAA response has each relationship represented as an atom:link element referring to the relationship resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<atom:feed xmlns=...
  <atom:entry>
  ...
     <atom:link rel='self' href='https://.../instances/Switch::10001' />
...
    <atom:content type='application/xml'>
      <Switch ...>
        <Model>MDS 9506</Model>

        ... etc. for the other properties of the Switch resource

        <atom:link rel='http://schemas.emc.com/msa/uim/1.0/Switch/relationship/PortGroup'
                        href='https:.../instances/Switch::10001/relationships/PortGroup' />
        <atom:link rel='http://schemas.emc.com/msa/uim/1.0/Switch/relationship/VSAN'
                        href='https://.../instances/Switch::10001/relationships/VSAN' />

        ... etc. for each relationship in the Switch resource type
      </Switch>
    </atom:content>
  </atom:entry>
```

Another approach is to "inline" the contents of the related resource collection as additional content in the atom:link element representing the relationship. An inline feed of related resources then provides additional detail on the representation of the relationship, and allows the consumer to avoid doing another request to the EDAA to examine the contents of the relationship. The cost of this representation is additional size of the original resource representation.

Using this option, a relationship now appears "expanded", with an inline feed of atom:entry elements each of which contains a representation of a related resource via the given relationship.

For example, if the Switch resource identified above (Switch::10001) was associated with 9 resources via the VSAN relationship, then each of those 9 resources would appear in its own atom:entry element as a child of the inline feed content of the relationship's atom:link element.

The consumer expresses its intention to retrieve this additional "expanded" form of a relationship, by naming the relationship to be expanded in an ?expand query parameter, such as:

```
GET types/Switch/instances?expand=VSAN
```

The EDAA would respond to this request with a slightly different representation of the Switch resource, expanding the relationship named "VSAN":

```
<?xml version="1.0" encoding="UTF-8"?>
<atom:feed xmlns=...
  <atom:entry>
  ...
     <atom:link rel='self' href='https://.../instances/Switch::10001' />
```

```
...
    <atom:content type='application/xml'>
      <Switch ...>
        <Model>MDS 9506</Model>

        ... etc. for the other properties of the Switch resource

        <atom:link rel='http://schemas.emc.com/msa/uim/1.0/Switch/relationship/VSAN'
                        href='https://.../instances/Switch::10001/relationships/VSAN' >
            <ae:inline xmlns:ae="http://schemas.emc.com/atom/ext/">
              <atom:feed xmlns = ...
                <atom:link rel="self" href="https://.../instances/Switch::10001/relationships/VSAN?page=1" />
                <atom:link rel="next" href="https://.../instances/Switch::10001/relationships/VSAN?page=2" />
                <atom:link rel="last" ... />
                <!-- uses RFC5005 for paging (if there are too many entries) -->
                 ...
                <atom:entry>                  <!-- Each related resource appears as its own atom:entry -->
                  <atom:title type='text'>VSAN - 1</atom:title>
                   ...
                  <atom:content type='application/xml'>
                    <inst:VSAN ...'>
                      <inst:VSANOperationalState>up</inst:VSANOperationalState>
                      ...

                     <atom:link rel='http://schemas.emc.com/msa/uim/1.0/VSAN/relationship/FCInterface'
                                    href='https://.../instances/VSAN::10001::RDN~2F~2F~2FVSAN~2F1/relationships/FCInterface'
/>
                    </inst:VSAN>
                 </atom:content>
            </atom:entry>

            ... etc for the other resources related via the VSAN relationship
        </ae:inline>
        </atom:link>

        ... etc. for each relationship in the Switch resource type, expanded or not, depending on what the consumer expressed
in the GET request URL

      </Switch>
    </atom:content>
  </atom:entry>
```

We note that this approach of expanding relationships with an inline feed of related resources is good for many relationships in practice especially those relationships that have a small handful of associations, but for those relationships that encapsulate hundreds of associations, this approach can cause a resource representation to get too large for consumers to tolerate. This is why this approach was not chosen as the "default" way relationships are represented in EDAA.

Normative Statements

A consumer would use an optional "expand" query parameter to request the expanded representation of relationships for resources returned for a given GET request.

The form of the ?expand query parameter is:

{some-url}?expand=[* | {relnName} [,{relnName}]*]

The value of the "expand" query parameter can take two forms:

- the wildcard, or "*" value, essentially matching all relationship names

or

- a collection of one or more {relnName} tokens, each of which SHOULD correspond to the name of a relationship contained in the

resource type of the resource (or collection) identified by {some-url}. A {relnName} can appear multiple times in the list, however a {relnName} appearing multiple times in the list is the same as it appearing once.

The "expand" query parameter can be added to any most requests (see Base EDAA spec for details). If the ?expand query parameter is included on a URL associated with an operation other than GET, then the EDAA MUST respond with an error, 400 (Bad Request).

If the resource referred to by {some-url} is homogeneous with respect to type, and the value of {relnName} does not correspond to any relationship name defined by that type, then the EDAA implementation MAY reject the request, returning and error, 400 (Bad Request).

For each relationship identified in the ?expand query parameter any appearance of that relationship in the response is represented in an an "expanded form" of the relationship. The "expanded form" of the relationship is the "normal" mechanism of using an atom:link to represent the relationship, but further annotating it with an inline atom:feed of related resources.

For each relationship NOT named in the ?expand query parameter, the relationship is represented in using the "default form" of an atom:link with no child element.

For example, if a FileServer resource type defined three relationships "ConnectedVia", "MountsTo" and "HostsServices", then the following request:

```
GET /instances/FileServer::nasp02.lss.emc.com?expand=ConnectedVia,MountsTo
...
```

Would return a representation of the single resource identified by FileServer::123. The representation of that resource would include the 3 relationships, as defined by the FileServer type, but the relationships "ConnectedVia", "MountsTo" would be represented using the "expanded form" (ie these relationship representations would include an in-line feed containing the related resources) and the relationship "HostsServices" would be represented in the default form.

An example response is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns=...
  <entry><title type="text">Representation of nasp02.lss.emc.com [Celerra Data Mover]</
title>   ...
    <content type="application/xml">
    <inst:FileServer ...
      <inst:OSVersion>Product: EMC Celerra File Server   Project: SNAS   Version: T5.5.28.1</

inst:OSVersion>       ... etc for the other attributes of the resource

      <link href="https:.../instances/FileServer::nasp02.lss.emc.com/relationships/ConnectedVia"
            rel="http://schemas.emc.com/ip/FileServer/relationships/ConnectedVia">  <!-- the relationship atom:link is the same
-->
        <ae:inline xmlns:ae="http://schemas.emc.com/atom/ext/">
                <!-- in this case, the link has child content using atom-inlining with an emc namespace -->
          <feed xmlns = ...
            <link rel="self" href="https:.../instances/FileServer::nasp02.lss.emc.com/relationships/ConnectedVia?
page=1&fields=DisplayName" />
            <link rel="next" href="https:.../instances/FileServer::nasp02.lss.emc.com/relationships/ConnectedVia?
page=2&fields=DisplayName" />
            <link rel="last" ... />
                <!-- uses RFC5005 for paging (if there are too many entries) -->
            ...
            <entry>                  <!-- Each related resource appears as its own atom:entry -->
              ...
              <content type="application/xml">
                 <!-- content element MAY appear, and it MAY contain a partial representation of the related resource -->
                 <!-- Service implementation decides on what attributes and relationships appear in the partial representation
-->
```

```
                <inst:IPNetwork xmlns:inst="http://schemas.emc.com/vs-xml/namespace/ip/1.0"
xmlns:atom="http://www.w3.org/2005/Atom">
                  <inst:DisplayName>10.9.20.240</inst:DisplayName>
                </inst:IPNetwork>
              </content>
            </entry>
            <entry>
              <content type="application/xml">
                 ...
              </content>
            </entry>

            ... etc for the other resources related via the ConnectedVia relationship
          </ae:inline>
        </link>

        <link href="https:.../instances/FileServer::nasp02.lss.emc.com/relationships/MountsTo"
              rel="http://schemas.emc.com/ip/FileServer/relationships/MountsTo">
          <ae:inline xmlns:ae="http://schemas.emc.com/atom/ext/">
            <feed xmlns = ...
          ...
        </link>

        <link href="https:.../instances/FileServer::nasp02.lss.emc.com/relationships/HostsServices"
              rel="http://schemas.emc.com/ip/FileServer/relationships/HostsServices" />
              <!-- Note: no child element, therefore this is a "default form" representation of the relationship -->

         ... etc. for the other relationships of the resource

        </inst:FileServer>
      </content>
    </entry>
</feed>
```

The use of atom:inline extension allows us to represent the relationship in terms of a subordinate, or in-line atom:feed where the entries contain links to the actual related resources.

For those relationships that are empty (no related resources), an empty inline element would be the child of the atom:link element.

RFC5005 pagination allows the server to govern or limit the size of these subordinate feeds in a way that the consumer can reason about and traverse. The server can apply RFC5005 for any relationship (or none of them) where the number of related resources exceeds some server determined limit. The server is free to handle pagination in anyway that provides the consumer clear access to the entire collection of related resources. In the example above, the first "page" of resources related via the "ConnectedVia" relationships appears in the ae:inline feed child of the relationship, subsequent pages reference similar partial representations of the related resources through the use of the ?fields query parameter on the href for the atom:link corresponding to the "next" page. The server is free to link full or partial representations as it sees fit. The server also chooses how big the page size is of the inline feeds; the ?per_page query parameter applies to the feed returned in the GET response, not to the individual inline feeds that may appear in the response.

Note that as of time the EDAA specification was written, the atom:inline extension is a proposal and requires modification to atom to allow the element as child of the atom:link element. We use the syntax as proposed by the extension, but, as a temporary measure until atom is changed, we put the inline element in a namespace. If the extension is ratified and atom is updated, we will revert to using the inline element from the (updated) atom namespace.

The resource representation appearing in the atom:entry children of the inline feed is up to the server. Many EDAA implementations will choose a very small subset (2 or 3) properties to appear in the inline feed. Returning full resource representations in the inline feeds should be done carefully, as they increase the size of the response substantially. This technique is useful for relationships of small cardinality (eg 0, 1 or a small number) of related resources, as it reduces yet another round trip to the server to traverse

relationships.

JSON representation of the inline feed construct

JSON formatted inline feeds follow a very similar pattern to their atom/XML

counterpart. From JSON in EDAA, a feed is represented as:

```
{
  ... metadata about the feed
  "entries":[
    {
      ... metadata related to the content of the entry ,
      ... representation of a resource
    },
    {
      ... metadata related to the content of the entry ,
      ... representation of a resource
    },

    ... etc for all the entries in the feed
  ],
  ...
}
```

And in particular, the "default" form of an example relationship appears as:

```
{
  ... entry related metdata
  "content" : {
    "@base": "https://example.com/msa/instances/FileServer::nasp02.lss.emc.com",

"OSVersion": "Product: EMC Celerra File Server   Project: SNAS   Version: T5.5.28.1",

    ... etc for the other attribute properties, represented as "property-name": <<value>> pairs.

    "links": [
      { "rel": "http://schemas.emc.com/msa/example/FileServer/reln/ConnectedVia",
        "href": "/relationships/ConnectedVia"
      },
      ... etc. for the other relationships, one link attribute per relationship, action links are also contained in the links
array
    ]
  }
}
```

For the "expanded" form of the relationship in JSON, we add an optional "inline" field to the "link object":

```
{
...
   "links": [
      { "rel": "http://schemas.emc.com/msa/example/FileServer/reln/ConnectedVia",
        "href": "/relationships/ConnectedVia",
        "inline": {
             <<<JSON representation of an atom feed appears as the value of the inline field >>>
        }
      },
...
```

A JSON equivalent of the atom feed XML example above is:

```
{
  "id": ...
  "updated": ...
  "links":[
      ...
  ],

  "entries":[
    {
      "updated": ...
      "content-type": "application/json",
      "content": {
        "@base": "https://example.com/msa/instances/FileServer::nasp02.lss.emc.com",

        "OSVersion": "Product: EMC Celerra File Server   Project: SNAS   Version: T5.5.28.1",

          ... etc for the other attribute properties

        "links": [
            {
              "rel": "http://schemas.emc.com/msa/example/FileServer/relationships/ConnectedVia",
              "href": "/relationships/ConnectedVia",
              "inline: {
                   "links": [
                     { "rel": "self",  "href": "/relationships/ConnectedVia?page=1&fields=DisplayName" },
                     { "rel": "next",  "href": "/relationships/ConnectedVia?page=2&fields=DisplayName" },
                     { "rel": "last",  "href": ... }
                   ],
                   "entries": [
                       {
                        "links": [ ... ],
                        "updated": ...,
                        "content-type": "application/json",
                        "content": {
                           "DisplayName" : "10.9.20.240"
                       },
                       {
                        "links": [ ... ],
                          …
                       },
                       ... etc. for the other resources related via the ConnectedVia relationship
                   ]
                }
            },
           {
               "rel": "http://schemas.emc.com/msa/example/FileServer/relationships/MountsTo",
               "href": "/relationships/MountsTo",
               "inline: {
               ...
               }
           },
           {
               "rel": "http://schemas.emc.com/msa/example/FileServer/relationships/HostsServices",
               "href": "/relationships/HostsServices"
           }
           … etc. for the other relationships of the resource
        ]
      }
    }
  ]
}
```

Base EDAA - JSON

This topic describes conventions on how EDAA uses JSON.

Contents

1 Preamble
2 Feeds and Entries
3 Links in JSON
    3.1 @base
4 Feed Metadata
5 Entry Objects
6 Instance Resource as a JSON Entry
7 Feed of Instance Resources
8 Type Resource as a JSON Entry
9 Feed of Type Resources
10 Note on coping with XML Namespaces
11 Error Response in JSON
12 JSON Format for body of PUT, POST and PATCH
    12.1 Create a new Alert Instance
    12.2 Update an Existing Alert Instance
    12.3 Update an Existing Alert Instance using PATCH
13 eTags

Preamble

Many REST APIs use Atom XML as the format for responses. This is a very good idea.

However, many Javascript clients prefer to process JSON formatted requests and responses, as the parsing is much simpler. The Primer has a section on JSON that describes pros/cons vis a vis atom/xml.

This portion of the proposal outlines how key portions of EDAA responses MUST appear when the consumer requests a JSON serialized response.

The EDAA authors considered using the mapping from ATOM into JSON developed by Google's Gdata map Atom into JSON. However, it seems that Google might be deprecating that approach, as the JSON created by this mapping is very verbose.

The EDAA spec contains examples of JSON objects, using a substitution or placeholder notation. So syntax like:

```
{
  <<SubsitutionName>> : { ... }
}
```

Should be read such that the literal bracketed by '<<' and '>>' should be substituted for a real string value.

It is a general recommendation that JSON object "names" use lowerCamelCase.

Feeds and Entries

The responses to EDAA operations are either feeds (equivalent of atom:feed elements) or entries (equivalent of atom:entry elements).

An entry MAY be a stand alone JSON object:

```
{
    ... metadata related to the content of the entry ,
    ... representation of a resource
}
```

An entry may appear in a feed:

```
{
  ... metadata about the feed
  "entries":[
    {
      ... metadata related to the content of the entry ,
      ... representation of a resource
    },
    {
      ... metadata related to the content of the entry ,
      ... representation of a resource
    },

    ... etc for all the entries in the feed
  ],
  ...
}
```

Links in JSON

In EDAA, links are used for several purposes:

1. metadata about feeds and entries, such as rel="self", rel="alternate" style links
2. metadata about pagination, rel="first", "prev", "next and "last" links
3. in a resource, a link is used to model each of the resource's relationships
4. in a type, links are used to model parent (super type) and an entire hierarchy for that type

A link contains several pieces of information:

1. rel, containing a name or URI describing the nature of the link relationship
   - is this a "next" link in pagination? Is this a URI to "self"? etc.
2. href, contains a URI to the related resource
3. type, we could use, but do not use in EDAA, a type field of a link to suggest the content-type of the resource identified by the value of href.

The general form of a link object in JSON is:

```
...
  {
    "rel": <<NCName or absolute URI describing the nature of the link>>,
    "href": <<URI which can be either an absolute URI, or a relative URI to some base (see below)>>,
    "type": <<OPTIONAL: Internet Media Identifier, such as would be found in an HTTP Accept: header>>
  }
...
```

For example, the following atom+xml link:

```
<atom:link rel="self" href="https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/FileServer"/>
```

would look like this in JSON:

```
  {
    "rel": "self",
    "href": "https://lglom041.lss.emc.com/msa/mss/man-lglom157/types/FileServer"
  }
```

In general, links are commonly grouped together. We recommend using a "links" array object to contain a collection of link objects:

```
...
  "links": [
     { <<link object>> },
     { <<link object>> },
  ]
...
```

For example, the following collection pagination related atom:links:

```
...
<link href="https://example.com/msa/types/FileServer/instances" rel="self"></link>
<link href="https://example.com/msa/types/FileServer/instances?page=1" rel="first"></link>
<link href="https://example.com/msa/types/FileServer/instances?page=3" rel="last"></link>
<link href="https://example.com/msa/types/FileServer/instances?page=2" rel="next"></link>
...
```

would look like this in JSON:

```
...
  "links":[
    {
      "rel": "self",
      "href": "https://example.com/msa/types/FileServer/instances"
    },
    {
      "rel": "first",
      "href": "https://example.com/msa/types/FileServer/instances?page=1"
    },
    {
      "rel": "last",
      "href": "https://example.com/msa/types/FileServer/instances?page=3"
    },
    {
      "rel": "next",
      "href": "https://example.com/msa/types/FileServer/instances?page=2"
    }
  ]
```

```
...
```

Note, in the above example, there is no "prev" link, since the "self" link indicates (by lack of ?page parameter) that there is no previous page.

@base

Continuing with the example in the previous section, we introduce a convenience called "@base". Using "@base" property could make this even shorter, by making certain URIs relative to a base URL:

```
...
  "@base": "https://example.com/msa/types/FileServer/instances",
  "links":[
    {
      "rel": "self",
      "href": ""
    },
    {
      "rel": "first",
      "href": "?page=1"
    },
    {
      "rel": "last",
      "href": "?page=3"
    },
    {
      "rel": "next",
      "href": "?page=2"
    }
  ]
...
```

If the "@base" property appears in a sibling or parent scope to the "links" resource, then any "href" child properties of link objects MAY be interpreted as relative URIs to the base URI. If there is no "@base" property available within the scope of the property, then any href properties MUST be absolute URIs.

Note that the "rel" property sometimes contains URIs as well, but the URIs in the "rel" property are NOT effected by the "@base" property, they are always absolute URIs.

Feed Metadata

The preamble to a "typical" atom:feed in EDAA looks like this:

```
<feed xmlns="http://www.w3.org/2005/Atom" ...>
  <updated>2011-03-21T16:15:11.937Z</updated>
  <title type="text">Instances of FileServer</title>
  <link href="https://example.com/msa/types/FileServer/instances" rel="self"></link>
  <link href="https://example.com/msa/types/FileServer/instances?page=1" rel="first"></link>
  <link href="https://example.com/msa/types/FileServer/instances?page=3" rel="last"></link>
  <link href="https://example.com/msa/types/FileServer/instances?page=2" rel="next"></link>
  <author>
    <name>MSA deployed at https://example.com/msa</name>
  </author>
  <id>urn:uuid:3089ad3a-eb8e-435e-a8e2-4fcc2c1f545a</id>

... entries
</feed>
```

Certain properties of the feed are very useful in the EDAA context:

- last updated property
- the "self" link
- the pagination links ("first", "prev", "next" and "last")
- any "alternate" links to the feed
- the feed id

Certain properties are required by Atom, and are more appropriate for blog feeds than EDAA work:

- /feed/title
- /feed/author

Therefore, to support the "useful" properties of a feed, JSON feed objects MUST contain:

```
{
    "id": <<<UUID unique to this feed>>>
    "updated": <<<date time the feed was generated or last changed>>>
    "links":[
        <<<see definition of "links": above, MUST contain a "self" link, MAY contain pagination links and other links>>>
    ],
    <<< other properties not defined by this specification, allowing extensibility >>>
    "entries":[
    {
      ... metadata related to the content of the entry ,
      ... representation of a resource
    },

    ... etc for all the entries in the feed
  ]
}
```

The JSON equivalent to the atom:feed preamble example given above is as follows:

```
{
  "@base": "https://example.com/msa/types/FileServer/instances",
  "id": "3089ad3a-eb8e-435e-a8e2-4fcc2c1f545a",
  "updated": 2011-03-21T16:15:11.937Z,
  "links":[
      { "rel": "self",  "href": ""},
      { "rel": "first", "href": "?page=1"},
      { "rel": "last",  "href": "?page=3"},
      { "rel": "next",  "href": "?page=2"}
  ],
  "entries":[
    ...
  ]
}
```

Note that two properties found in the atom:feed are not found in the JSON feed:

```
   <title type="text">Instances of FileServer</title>
  <author><name>EDAA deployed at https://example.com/msa</
name></author>
```

These properties are required by the atom specification but they serve no useful purpose in EDAA.

Entry Objects

An example atom:entry representing an individual resource representation in an atom:feed of resources is shown below:

```
...
<entry>
  <title type="text">Representation of nasp02.lss.emc.com [Celerra Data Mover]</title>
  <updated>2011-03-21T18:22:07.347Z</updated>
  <id>https://example.com/msa/instances/FileServer::nasp02.lss.emc.com</id>
  <link href="https://example.com/msa/instances/FileServer::nasp02.lss.emc.com" rel="alternate"></link>
  <link href="https://example.com/msa/types/FileServer" rel="http://schemas.emc.com/msa/common/reln/type"></link>
  <content type="application/xml">
    <inst:FileServer xmlns:inst="http://schemas.emc.com/msa/example" xmlns:atom="http://www.w3.org/2005/Atom">
    ... resource representation
    </inst:FileServer>
  </content>
</entry>
...
```

Certain properties are useful to include as metadata for an entry (the JSON equivalent to an atom:entry in EDAA):

- "self" link
- relationship to the type definition
- alternative representations (eg "alternate" links)
- updated (when the resource associated with the entry was last changed)
- /content/@content-type

Certain properties are required by Atom, but not immediately useful to EDAA:

- title
- id, since it is not a unique identifier to this particular entry in this particular feed

The general pattern is to divide the representation of a entry into two parts: entry metadata and a content object that contains a JSON representation of the resource in the content field:

```
...
{
  "links": [
    ... the various "self" and "alternate" and other links associated with the entry
    ... note, the actual "type" of the resource contained in the "content" object is defined by a link to the type resource
  ],
  "updated": <<date/time the resource was last changed>>,
  "content-type": <<<string representing a valid Internet Media Type as would be found in an HTTP Accept:
header>>, <<other attributes that can be added by the EDAA implementation>>,
  "content": {
    ... representation of a resource
  }
}
```

The JSON equivalent of the atom:entry example given above is as follows:

```
{
  "@base": "https://example.com/msa"
  "links": [
    { "rel": "self",  "href": "/instances/FileServer::nasp02.lss.emc.com"},
    { "rel": "http://schemas.emc.com/msa/common/reln/type", "href": "/types/FileServer"},
  ],
```

```
  "updated": 2011-03-21T18:22:07.347Z,
  "content-type": "application/JSON;version=1.0",
  "content": {
    ... representation of a the attributes and relationships of the FileServer instance identified by
"FileServer::nasp02.lss.emc.com"
  }
}
```

Note the following elements in the atom:entry have no corresponding property in the JSON representation of an entry:

```
  <title type="text">Representation of nasp02.lss.emc.com [Celerra Data Mover]</title>
<id>https://example.com/msa/instances/FileServer::nasp02.lss.emc.com</id>
```

Instance Resource as a JSON Entry

A representation of an instance resource has the following types of components:

- the type of the resource
- attribute properties
- relationship properties as link elements
- action links

The following is an example content element containing a representation of a resource:

```
...
<entry>
...
  <content type="application/xml">
     <inst:FileServer xmlns:inst="http://schemas.emc.com/msa/example" xmlns:atom="http://www.w3.org/2005/Atom">
    <inst:OSVersion>Product: EMC Celerra File Server   Project: SNAS   Version: T5.5.28.1</inst:OSVersion>

      ... etc for other properties

      <atom:link rel="http://schemas.emc.com/msa/example/FileServer/reln/ConnectedVia"
                 href="https://example.com/msa/instances/FileServer::nasp02.lss.emc.com/relationships/ConnectedVia"/>

      ... etc. for the other relationships
    </inst:FileServer>
  </content>
</entry>
...
```

As we have seen previously, the content type (value of /content/@type) is an attribute of the entry. The actual resource representation is a JSON object that is the value of the "content" object. The JSON representation for the resource depicted above in the atom:content element is:

```
{
  ... entry related metdata
  "content" : {
    "@base": "https://example.com/msa/instances/FileServer::nasp02.lss.emc.com",

"OSVersion": "Product: EMC Celerra File Server   Project: SNAS   Version: T5.5.28.1",

    ... etc for the other attribute properties, represented as "property-name": <<value>> pairs.

    "links": [
      { "rel": "http://schemas.emc.com/msa/example/FileServer/reln/ConnectedVia",
```

```
        "href": "/relationships/ConnectedVia"
      },
      ... etc. for the other relationships, one link attribute per relationship, action links are also contained in the links
array
    ]
  }
}
```

Note, if the information model of the resource contained within the "content" object is composed of more than one namespace, then see below for more information on how to cope with multiple namespaces in the JSON representation.

Feed of Instance Resources

A feed of resources representing instances is a combination of the entry pattern and the representation of an instance resource in JSON. The general form of a feed of instance resources in JSON, that would correspond to the response to a GET on /instances, for example, would look like:

```
{
  "id": <<<UUID unique to this collection>>>
  "updated": <<<date time the feed was generated or last changed>>>
  "links":[
      <<<see definition of "links": above, MUST contain a "self" link, MAY contain pagination links and other links>>>
  ],
  <<<other feed properties>>>
  "entries":[
    {
      "links": [ ... ],
      "updated": <<some date/time>>,
      "content-type": <<some media type>>,
      "content": {
         ... representation of a the attributes and relationships of a resource
      },
    },
    ... etc. for each entry in the feed.
  ]
}
```

Type Resource as a JSON Entry

In EDAA, types are also represented as resources, using a specialized XML syntax called VS-XML. Here is an example type resource represented within an atom:entry:

```
...
<entry>
  ...
  <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/msa/example">ICIM_RelayDevice</type:typeName>
      <atom:link rel="self" href="https://example.com/msa/types/ICIM_RelayDevice"/>
      <atom:link rel="http://schemas.emc.com/msa/common/reln/hierarchy"
                 href="https://example.com/msa/types/ICIM_RelayDevice/hierarchy" />
      <atom:link rel="http://schemas.emc.com/msa/common/reln/parent"
                 href="https://example.com/msa/types/UnitaryComputerSystem"/>

      <type:attribute type="xs:string" minOccurs="0" maxOccurs="1">OSVersion</type:attribute>
      ... etc. for each attribute in the type

      <type:relationship relType="ICIM_Connection">ConnectedVia</type:relationship>
```

```
      ... etc. for each relationship associated with the type
    </type:Type>
  </content>
</entry>
...
```

The representation of a type resource has four kinds of property:

- metadata including the name of the type and links (like "self" and related links like hierarchy and parent relationships) related to the type itself
- a list of attribute declarations
- a list of relationship declarations
- a list of action declarations

The general format of a Type object in the JSON representation is:

```
{
  "name": <<string representing the name of the type>>,
  "namespace": <<URI identifier of the namespace in which this type is being defined>>,
  "description": <<OPTIONAL brief sentence or two describing the type>>,
  "documentation": <<OPTIONAL URL to a web resource providing more documentation on the type>>
  "links: [
      <<link objects for "self" and the parent type and the collection of types associated within the type hierarchy>>
  ],
  "attributes": [
    {
      "name": <<name of an attribute>>,
      "type": <<type of the attribute, often a reference to a datatype within the XML Schema Datatyping system>>,
      "minOccurs": <<OPTIONAL string to indicate the minimum number of times this attribute must appear in an instance of the
type being defined,
                    default is "0" >>,
      "maxOccurs": <<OPTIONAL string to indicate the maximum number of times this attribute can appear in an instance of the
type being defined,
                    default is "unbounded">>
      "default": <<OPTIONAL simple value expressing what value the attribute can be assumed to take if it is absent from a
resource representation>>,
      "description": <<OPTIONAL brief sentence or two describing the attribute>>,
      "documentation": <<OPTIONAL URL to a web resource providing more documentation on the attribute>>
    }, ... repeated once for each attribute in the type
  ],
  "relationships":[
    {
      "name": <<name of a relationship>>,
      "relType": <<this is the type of resource associated via this relationship>>,
      "type": <<OPTIONAL URI to a common relationship type that defines a set of semantics this relationship shares>>,
      "minOccurs": <<OPTIONAL string to indicate the minimum cardinality of this relationship, default is "0">>,
      "maxOccurs": <<OPTIONAL string to indicate the maximum cardinality of this relationship, default is "unbounded">>,
      "description": <<OPTIONAL string with a brief description of the relationship>>
      "documentation": <<OPTIONAL URI to a web resource providing more documentation on the relationship>>
    }, ... repeated once for each relationship in the type
  ],
  "actions": [
    {
      "rel": <<an NCName or URI of as defined for atom:link elements in Atom Syndication Format RFC4287>>,
      "description": <<OPTIONAL human readable text description of the action>>,
      "documentation": <<OPTIONAL URL to human readable documentation that gives more detail about the inputs, outputs and
behavior of the action>>
    }
    ... repeated once for each kind of action that "typically" or "possibly" appears in representations of instances of the
type
  ],
  <<other attributes, if any, that the author wishes to add>>
}
```

For the type represented above, the JSON representation is:

```
{
  "@base":"https://example.com/msa",
  "name": "ICIM_RelayDevice",
  "namespace": "http://schemas.emc.com/msa/example",
  "links":[
    {
      "rel": "self",
      "href": "/types/ICIM_RelayDevice"
    },
    {
      "rel": "http://schemas.emc.com/msa/common/reln/hierarchy",
      "href": "/types/ICIM_RelayDevice/hierarchy"
    },
    {
      "rel": "http://schemas.emc.com/msa/common/reln/parent",
      "href": "/types/UnitaryComputerSystem"
    }
  ],
  "attributes": [
    {
      "name": "OSVersion",
      "type": "xs:string",
      "minOccurs": "0",
      "maxOccurs": "1"
    },
    ... etc for each attribute
  ],
  "relationships":[
    {
      "name": "ConnectedVia",
      "relType": "ICIM_Connection"
    },
    ... etc. for each relationship
  ],
  "actions": []
}
...
```

Note the use of "xs:" in "xs:string" should match a namespace declaration for "xs:", see coping with namespaces, below.

Feed of Type Resources

A feed of resources representing types is a combination of the entry pattern and the representation of a type resource in JSON within the content object of the entry. The general form of a feed of type resources in JSON, that would correspond to the response to a GET on /types, for example, would look like:

```
{
    "id": <<<UUID unique to this collection>>>
    "updated": <<<date time the feed was generated or last changed>>>
    "links":[
        <<<see definition of "links": above, MUST contain a "self" link, MAY contain pagination links and other links>>>
    ],
    <<<other feed level properties>>>
    "entries":[
    {
      "links": [ ... ],
      "updated": <<some date/time>>,
      "content-type": <<some media type>>,
      "content": {
        "name": "type_name_1",
        "namespace": ...
```

```
        "links: [
          ...
        ],
        "attributes": [
          {
            "name": "attr_1",
            "type": ...,
            "minOccurs": ...,
            "maxOccurs": ...,
          }, ... etc. for each attr
        ],
        "relationships":[
          {
            "name": "reln_name",
            "relType": ...,
            ...
          }, ... etc. for each relationship in the type
        ],
        "actions": [
          {
            "rel": "rel value",
            "description": "...",
            "documentation": "..."
          },  ... etc. for each action in the type

      },

  ... etc. each type is in a content object of a separate entry object.

  ]
}
```

Note on coping with XML Namespaces

In certain cases, for example in type declarations, the information model is inherently XML centric and it is required to clarify or namespace information items in the JSON that derive from different namespaces.

Following the example set by GData, we can define a QName like syntax for namespace qualified variables in JSON.

The steps are similar to those within an XML instance document. First declare the namespace and prefix mapping at an appropriate outer scope within the JSON representation:

```
{
  "xmlns$atom": "http://www.w3.org/2005/Atom"
  ...
}
```

Note the use of the $ syntax to subdivide the variable name in the namespace declaration.

And then within the scope of the declaration, use the prefix to scope the variable to the namespace:

```
{
   "xmlns$atom" ...
   {
   ...
     "atom$feed": ...
}
```

The variable above, called "atom$feed" should be interpreted as a named variable associated with the scope assigned to the prefix

"atom".

At some point in the future, when JSON Linked Data spec matures and settles out, we should consider using JSON-LinkedData to represent the mapping to XML namespaces, and in general use its RDFa-style notation to represent types in a future version of EDAA.

Error Response in JSON

EDAA also defines a standard approach to representing Error Resources in feeds and entry content of responses. Example JSON encoding of the Error resource can be found here.

Consumers can distinguish success responses from error responses by examining the http error code. A response in the 2xx range indicates success. A response in the 4xx or 5xx indicates some sort of failure condition.

Success responses, be they singleton style responses like the response to GET /types/{typeName}, or list style responses like the response to GET /instances, have the form

```
200 OK
...
{
  ... <<<collection metadata>>>
  "entries": [
     {
       ... <<<entry metadata>>>
       "content": {
           ... <<<JSON representation of a resource>>>
       }
     },
     ... etc. for other entries, if any
  ],
  ...
}
```

Error responses appear like:

```
4xx or 5xx
{
    "Severity" : 3,
    "Type" : "http://schemas.emc.com/msa/common/error/resource_not_found",
    "ErrorCode" : 404,
    ... etc for other properties of the error
}
```

JSON Format for body of PUT, POST and PATCH

When a consumer places JSON content in the body of a PUT or POST operation, the consumer MUST also include an HTTP Content-Type: header with the value application/JSON.

EDAA implementations MUST examine the Content-Type header of PUT, POST and PATCH operations to understand whether XML or JSON (or something else) is used to format the body.

EDAA implementations MUST support the Content-Type of application/XML. EDAA implementations SHOULD support Content-Type of application/JSON. EDAA implementations MAY support other Content-Types.

If a PUT, POST or PATCH operation is received by an EDAA implementation, but the Content-Type specified in the request is one

that is NOT supported by the implementation, then the implementation must return an HTTP 400 (Bad Request) error code with a standard error message indicating the body of the request is formatted using an unsupported Content-Type.

The following sections show example JSON payloads for simple operations (PUT, POST, PATCH) on an Alert resource.

Create a new Alert Instance

To create a new instance of the Alert type, using JSON, the message would look like:

```
POST /types/Alert/instances

{
    "Type": "Alert",
    "Severity": 1,
    "State": "Open",
    "Message": "some string, lorem ipsum",
    "Description": "some longer string than the Message, bla bla bla",
    "Tags": [
      "string for first Tag",
      "another Tag"
    ],
    "Resource": "https://example.com/msa/instances/Foo::1234",
    "ResourceType": "https://example.com/msa/types/Foo"
}
```

Update an Existing Alert Instance

Assuming the POST operation in the previous example succeeded, an Alert resource available at https://example.com/msa/instances/Alert::D7A9CF0F-F81C-8E37-74DDBE1C16ED0B18. The consumer can use a PUT operation to modify it.

```
PUT /instances/Alert::D7A9CF0F-F81C-8E37-74DDBE1C16ED0B18

{
    "Severity": 1,
    "State": "Closed",
    "Message": "some string, lorem ipsum",
    "Description": "some longer string than the Message, bla bla bla",
    "Tags": [
      "string for first Tag",
      "another Tag"
    ]
}
```

Because the PUT operation MUST be idempotent, the contents of the PUT must include the following properties of the Alert: Severity, State, Message, Description, Tags.

Update an Existing Alert Instance using PATCH

Again, examining the Alert resource available at https://example.com/msa/instances/Alert::D7A9CF0F-F81C-8E37-74DDBE1C16ED0B18. The consumer can use a PATCH operation to modify a subset of the resource's properties.

```
PATCH /instances/Alert::D7A9CF0F-F81C-8E37-74DDBE1C16ED0B18
```

```
{
    "State": "Closed",
}
```

Note unlike in the PUT operation described above, the PATCH operation is not required to be idempotent, and therefore can contain only the properties that need to be changed. Unfortunately, as discussed in PUT vs PATCH most tools and http libraries do not support PATCH.

eTags

As discussed in the eTag section of the EDAA spec, EDAA implementations are encouraged to support the notion of eTags for cache control and optimistic concurrency control.

Google's gdata defines an approach to placing eTags placing eTags in atom:feed and atom:entry elements.

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 ... ,
    ...
  }, ...
}
```

EDAA Read/Write

This topic is describes various techniques to address the needs of third party applications (consumers) to make modifications to the resources exposed via an EDAA.

Contents

1 See Also
2 Goal
3 EDAA Read/Write Patterns
4 Normative Aspects of EDAA Read/Write
4.1 Creating New Resources
4.2 Decorating Resource Representations with Capabilities
4.2.1 HTTP OPTIONS Verb
4.2.2 Using atom:links
4.3 Task Resource
4.3.1 Task Resource Model
4.3.2 Example
4.3.3 Knowing when the Task Completes
4.3.4 /types/Task
4.3.5 Multiple Resources
4.4 Supporting Partial Representations in VS-XML
4.4.1 Extending the /types/{typeName} URI pattern to support Partial Representations
4.5 Supporting "action" verb REST-hybrid
4.5.1 "action" links in Resource Representations
4.5.2 "action" verb example
4.6 Extending the Relationship Representation with Association Resources
4.7 Alias URIs
4.8 Using PATCH vs PUT

See Also

Related links:

- Primer may be a good place to begin reading.
- The base EDAA spec, focuses on read-only information

Goal

Base EDAA is a section of the EDAA specification dealing with read-only access to data, specifically describing how consumers can

issue HTTP GET operations against a set of well known URI patterns or more typically, URIs contained in the @href of atom:link elements.

The goal of this section of the EDAA specification is to standardize how EDAA supports consumer applications that create new resources, modify existing resources, delete existing resources and otherwise modify data through an EDAA interface.

EDAA Read/Write Patterns

The Primer reviews a set of "patterns" that developers should consider when they attempt to enhance the EDAA capabilities of their product to support operations that create/update and delete resources or otherwise allow consumers to directly modify data within their product through an EDAA interface.

A good understanding of the read/write patterns described in the Primer would be helpful to understand the normative aspects of EDAA read/write specification.

Normative Aspects of EDAA Read/Write

As discussed in the Primer, a lot of read/write use cases can be solved by basic applications of REST (eg HTTP POST/PUT/PATCH or DELETE) on the URI patterns in EDAA such as /types/{typeName}/instances and /instances/{id}.

There are, however, a few parts of the EDAA read/write approach that extend this basic use of REST, as discussed below.

Creating New Resources

A common way to create new instances of a resource type is to send a POST message to /types/{typeName}/instances. If an EDAA implementation permits resource creation by consumers, then it MUST include an atom:link element in the representation of the type resource to indicate that ability.

For example, if an EDAA implementation allows consumers to create new instances of the VCenter type, then the representation of /types/VCenter would need to contain <link href="https:.../types/VCenter/instances" rel="edit" /> as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="...
  <entry><title type="text">VCenter</title>
    <id>https:.../types/VCenter</id>
    ...
    <link href="https:.../types/VCenter" rel="self"></link>
    ...
    <link href="https:.../types/VCenter/instances" rel="edit" />
    <content type="application/xml">
      <type:Type xmlns:type="http://schemas.emc.com/vs-xml/namespace/Common/2.0" xmlns:atom="http://www.w3.org/2005/Atom">
        <type:typeName namespace="http://schemas.emc.com/vs-xml/namespace/uim/1.0">VCenter</type:typeName>
      ...
    </content>
  </entry>
...
</feed>
```

If an EDAA permits consumers to create new instances of a given type, it MUST support POST operation on /types/{typeName}/instances for those values of {typeName} for which instantiation is applicable. Furthermore, any representation of the type identified by {typeName} MUST include an atom:link with @rel="edit" indicating the type is instantiable.

The response to a POST operation creating a new resource is:

- http status code 201
- optionally an http-location header containing a URL to the newly created resource (as in Atom Publishing protocol)
- a representation of the newly created resource as an entry within a feed

Decorating Resource Representations with Capabilities

Not all resources support all of the possible operations in the uniform interface (GET/PUT/POST/DELETE/OPTIONS/HEAD and sometimes PATCH). In addition, several resources support "verb-like" or "action" functionality that may be performed on them like "provision" or "activate".

How does a consumer tell which operations are available on which resources? There are three mechanisms available:

1. HTTP OPTIONS verb
2. Examining atom:link elements in the resource's representation
3. Examining a representation of the type see EDAA types and VS-XML

HTTP OPTIONS Verb

For specifying which subset of the uniform interface is available for a particular resource, the EDAA SHOULD support the OPTIONS operation on resources it implements.

For example, if a particular resource of type Alert is implemented to allow GET operations, PUT operations and DELETE operations, then a request:

```
OPTIONS /instances/Alert::1234 HTTP/1.1
```

Should be implemented to respond with:

```
HTTP/1.1 200 OK
Allow: GET, PUT, OPTIONS, DELETE
```

But note, this does not indicate that additional action verbs are supported.

Using atom:links

In any representation of a resource, the EDAA MUST include an atom:link for each "modifying" operation supported by that resource. Modifying operations include PUT,POST,PATCH,DELETE and any "action" verb operation, like "provision", "activate", etc.

For PUT,POST,DELETE (and even PATCH), we will use the rel="edit" value, per the Atom Publishing Protocol. This means that the URI associated with the @href can be used to PUT (update), POST (create new) and DELETE (delete), per Atom Publishing Protocol. We will also use this URI for PATCH. The consumer SHOULD use an OPTIONS message on the URI to determine what, if any, verbs are supported at that URI.

For other operations, the value of @rel MUST correspond to a URI that identifies the kind of operation being represented. For example, if the @rel is http://schemas.emc.com/rel/uim/VCenter/action/activate, then the @href might be https://.../instances/VCenter::1234/action/activate. The consumer should reference human consumable documentation for further semantics of the operation, and for details on which HTTP verb to use (normally POST) and constraints on the request headers and

body.

Note, it is tempting to simplify the @rel for action verbs by using just the name of the action, ie use **activate**. This, however is not legal atom:link syntax. Instead use **http://schemas.emc.com/rel/uim/VCenter/action/activate**.

The atom:Link element's @rel (according to its definition in RFC 4287) MUST be either an IRI (ie URI) or one of the IANA assigned REL names. Although the wording in the RFC could be clearer, it is reasonable to assume this semantic. Therefore the use of "{action name}" for @rel is not legal -- we use URIs. The URI scheme, although a bit more syntax, is clearer. Besides, this detail is not necessarily intended to be human readable, most of the time, Javascript or other programming logic will be reasoning about and pattern matching over these URIs.

We did consider putting these types of links only in the types feeds and not on the actual instances themselves. As in many cases this is just metadata and will be static. However, while *sometimes* the ability to, say, perform an operation, or be modified is at the type level, this may not be true in general. There may be certain instances of a given type that cannot be deleted, perhaps due to some current situation with the resource, the capabilities of the consumer, etc. Having this information at the instance level provides the right level of association with the resource itself. It follows a HATEOAS principle.

Task Resource

All operations invoked through the EDAA SHOULD be considered long-running and therefore respond with an error (if the request is invalid) or, if the request is syntactically valid, an HTTP 202 (Accepted) response code and a Task resource. Consumers of EDAA operations MUST examine the HTTP response code to determine if an error occurred. And if a 2XX error code was returned, determine if the operation completed synchronously (200 code returned) or asynchronously (202 code returned). In the case of a 202 code returned, the consumer should expect that some representation of the Task resource is associated with the response.

Many PUT/POST/PATCH or DELETE operations may require modification of the underlying IT Resources and may require a long running task to be orchestrated to implement the semantic suggested by the operation. In these cases, an asynchronous operation should be built and a Task resource returned to the consumer in response to the request. Some of these operations are straight forward and can be completed with little latency, in this case, implementing the operation as a synchronous operation is the simplest approach.

Many GET operations are straight forward to perform and can be completed synchronously with respect to the request. Some GET operations (e.g. a complex query) MAY respond with an HTTP 202 and a Task resource, if the service implementation deems it may take a long time to construct the response.

If the response does contain a Task, there are two places the URI to the Task resource may be communicated in the response:

1. The response MUST contain an HTTP Location: header, containing the URI to the Task resource created by the request message
2. The response MAY contain a representation of the Task resource in the body of the response message.

Task Resource Model

- The Task schema is modeled in the namespace identified by: http://schemas.emc.com/msa/common
- All Task resources are identified by an opaque identifier, preferably a UUID generated by the EDAA implementation.
- A Task resource has the following URI: {base-URI}/instances/Task:{ID}, where {base-URI} identifies the EDAA implementation.
- All task resources have the following properties:

Property Name	Type	(min,max)	Description
ID	String (preferably a UUID)	(1,1)	A unique identifier for the Task instance that distinguishes a Task from all other Tasks known to an EDAA implementation.
Description	String	(1,1)	A brief textual description of the circumstances that caused this Task to be created.
Initiator	URI	(1,1)	A copy of the URI used in the request that caused this Task to be created.
InitiatorType	URI	(1,1)	A copy of the type of the initiator. This is derived from the @rel attribute of the atom:link element associated with the Initiator.
StartTime	DateTime	(1,1)	The time the processing associated with the Task was started.
EndTime	DateTime	(1,1)	The time the processing associated with the Task ended. This element may be empty, indicating that the processing associated with the Task is still in progress.
ExpireTime	DateTime	(1,1)	The time the Task will be permanently removed from the EDAA.
ExecutionState	String	(1,1)	The property contains a value, from an extensible enumeration of possible states the Task may be in. The initial enumeration includes "Executing", "Completed", "Failed".
Progress	Float	(0,1) - Not all implementations of the Task resource have the ability to estimate or monitor the progress of processing	An optional representation of percent completion of the task.
AffectedResources	Relationship to zero or more Resources	(0,n) - Assigned by the EDAA when the task is complete	This relationship MUST be empty in a Task's representation until the processing associated with the Task completes. After processing completes successfully, the relationship will reference a collection of zero or more resources acted upon by the processing associated with the task. The consumer SHOULD retrieve this resource when the task signals its successful completion. If the process ends in failure, this relationship MUST remain empty.
Errors	Relationship to zero or more Error resources	(0,n) - Assigned by the EDAA when the task ends in failure	This relationship will be empty in a Task's representation until the processing associated with the Task ends in failure. When the processing completes unsuccessfully, the relationship will contain a feed of one or more Error resources.
Other Properties	Any	(0,*)	Specializations of the Task type may include additional properties.

- A Task MAY include zero or more atom:link elements indicating the mechanisms (if any) for the consumer to register for

asynchronous event notification for change in status of the Task resource. See Below.

- The EDAA implementation MUST support the GET operation on Task resources, per the EDAA requirements and options for GET operations.
- The EDAA MAY support the DELETE operation (this is something an EDAA implementation chooses to support or not support), allowing third party consumers to terminate the processing of a request and removing the Task resource from the EDAA. If DELETE is supported for a task, then the representation of the Task MUST include an atom:link element with the value of @rel as "edit" and the value of @href as the URI to the Task resource. An OPTIONS message on this URI would indicate that DELETE is the only modification operation supported (not PUT or POST or PATCH).

Example

Consider the following request:

```
PUT /instances/Alert:1234 HTTP/1.1
...
{body containing an update partial representation of the Alert resource}
```

The EDAA responds with a Task resource:

```
HTTP/1.1 202 Accepted
...
Location: http://.../instances/Task:0c9e3e50-4365-11e0-9207-0800200c9a66
...
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:vsc="http://schemas.emc.com/msa/common"><updated>2011-02-
28T18:04:00.390Z</updated>
<title type="text">Representation of Task 0c9e3e50-4365-11e0-9207-0800200c9a66</title>
<link href="http://.../instances/Task:0c9e3e50-4365-11e0-9207-0800200c9a66" rel="self"></link>
...
<content type="application/xml">
<inst:Task xmlns:inst="http://schemas.emc.com/vs-xml/namespace/Common/1.0"
  xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://.../instances/Task:0c9e3e50-4365-11e0-9207-0800200c9a66"/>
  <atom:link rel="edit" href="http://.../instances/Task:0c9e3e50-4365-11e0-9207-0800200c9a66"/>
  <atom:link rel="http://schemas.emc.com/msa/eventing/Task/progress" href="...{exchange_name}/queue/Task_0c9e3e50-4365-11e0-
9207-0800200c9a66">
        … optional further markup to describe the event, the subscription protocol or the content delivery protocol
  </atom:link>

  <inst:ID>0c9e3e50-4365-11e0-9207-0800200c9a66</inst:ID>
  <inst:Description>Task related to PUT /instances/Alert:1234</inst:Description>
  <inst:Initiator>http://.../instances/Alert:1234</inst:Initiator>
  <inst:Type>http://schemas.emc.com/msa/rel/Update</inst:Type>
  <inst:StartTime>Mon Feb 28 13:14:18 EST 2011</inst:StartTime>
  <inst:EndTime></inst:EndTime>
  <inst:ExpiryTime>Tue Mar 1 13:14:18 EST 2011</inst:ExpiryTime>
  <inst:ExecutionState>Executing</inst:ExecutionState>
</inst:Task>
</content>
</entry>
</feed>
```

Knowing when the Task Completes

A consumer can monitor the progress of its request by monitoring the status of the corresponding Task resource. There may be several strategies available to the consumer to monitor status of the Task:

1. Polling
   - The consumer periodically sends a GET message to the Task URI returned in the response to the initial request. The consumer MAY also send an ETag to make the polling operation more efficient. If the GET returns an updated representation of the Task, the consumer may examine the Status property to see if the processing has completed.
2. Asynchronous Notification
   - The Task resource representation MAY contain atom:link entries corresponding to event notification. In this case, the consumer may use one of those links to subscribe to receive event notification for changes in the Task resource.

When processing for the Task is complete, the status property of the Task is updated, along with the EndTime property and either the Resource property (if the processing was successful) or the ErrorMessage and (usually) the ErrorCode properties if the processing was unsuccessful.

In the case where processing was successful, the Resource property contains a URI to an atom:entry for a single resource, or a URI to an atom:feed of multiple resources, depending on the number of resources changed as a result of processing the Task.

/types/Task

All EDAA implementations MUST provide a description of the Task resource by implementing the GET operation on /types/Task.

Multiple Resources

A Task might result in the creation or modification of multiple resources, in which case, the Resource property of the Task resource will, upon completion of processing, contain a URI referring to an atom:feed containing multiple atom:entry elements, one for each resource created/modified by the processing of the request.

Supporting Partial Representations in VS-XML

Note, the entire notion of VS-XML may be augmented to support different representations, such as an XML Schema (XSD) representation or an RDF representation. At the time this is written, those discussions are not sufficiently mature. We will propose normative changes to VS-XML in this section and, when the time comes, comment on how these changes may impact alternative formats such as XSD or RDF.

Extending the /types/{typeName} URI pattern to support Partial Representations

In order for clients to understand the partial representations that may be required on PUT/POST operations associated with a given resource type, we extend the URI patterns for /type/{typeName}.

The resource /type/{typeName}/PR_Create will contain a VS-XML representation of that subset of attributes and relationships that MUST/MAY be included in the body of a POST (or PUT message) when the consumer attempts to create a new resource instance of that type. Items with minOccurs > 0 MUST appear, those with minOccurs = 0 MAY appear.

For any Type resource that supports a create Partial Representation, an atom:link element must be included in the representation of that Type to indicate availability of these partial representation resources. The value of @rel MUST be http://schemas.emc.com/msa/rel/PR_Create.

For example, imagine a VCenter type and imagine that there are many properties, of which two properties, ("Name" and "ipAddress"), must be specified when a consumer creates a VCenter resource. In this case, the resource at

/types/VCenter/PR_Create would be an atom:feed containing one resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:vsc="http://schemas.emc.com/vs-xml/namespace/Common/1.0"><updated>2011-02-
28T21:55:46.910Z</updated>
<title type="text">Partial Representation (Create) for to https://.../types/VCenter</title>
<link href="https://.../types/VCenter/PR_Create" rel="self"></link>
<author>...</author>
<id>urn:uuid:...</id>
<entry><title type="text">VCenter Create Partial Representation</title>
  <id>https://.../types/VCenter/PR_Create</id>
  <updated>2011-02-28T21:55:46.910Z</updated>
  <link href="https://.../types/VCenter/PR_Create" rel="self"></link>
  <link href="https:.../types/VCenter" rel="related"></link>
  <content type="application/xml">
    <type:Type xmlns:type="http://schemas.emc.com/vs-xml/namespace/Common/2.0" xmlns:atom="http://www.w3.org/2005/Atom">
      <type:typeName namespace="http://schemas.emc.com/vs-xml/namespace/uim/1.0">VCenter_PR_CREATE</type:typeName>
      <atom:link rel="self" href="https://.../types/VCenter/PR_Create"/>
      ...
      <type:attribute type="xs:string" minOccurs="1" maxOccurs="1">Name</type:attribute>
      <type:attribute type="xs:string" minOccurs="1" maxOccurs="1">ipAddress</type:attribute>
    </type:Type>
  </content>
</entry>
</feed>
```

Supporting "action" verb REST-hybrid

As mentioned in the Primer it is difficult to support operations like "reboot" or "provision" directly on resources using the REST uniform interface. It is pragmatic to support the notion of a "REST-hybrid" interface to accommodate these kinds of use cases. This approach should be used by developers only when a simpler, direct use of the REST uniform interface is not an appropriate solution.

There are several aspects of the "REST-hybrid" approach that are standardized in EDAA

"action" links in Resource Representations

For those resources that support one or more "action" verbs, one atom:link element for each "action" SHOULD appear within the representations of the resource. Note, in situations where the state of the resource or the permissions of the consumer do not allow an "action" to occur, the atom:link MUST not appear in the resource representation.

The "action" atom:link MUST contain a value of @rel conforming to the following pattern: {domain-identifier}/{typeName}/action/{actionName}.

The "action" atom:link MUST contain a value of @href, that corresponds to a resource that, when invoked with a POST operation, will cause the "action" identified by the atom:link to be initiated.

Any information about the required input parameters (eg body of the POST operation) or output results (eg the expected content of the result) is NOT specified by any metadata convention defined by EDAA. It is recommended that developers provide human readable documentation that describe these details to consumers. As noted below there is metadata at the EDAA type level that MAY contain information about the action.

"action" verb example

For **action-style verbs**, consider the task of cloning a vAppTemplate. The consumer could observe from a representation of a vAppTemplate resource that there was a cloning action available, by examining the set of atom:link elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="...
  <entry>...
    <link href="https://.../instances/vAppTemplate:1234" rel="self"></link>
    ...
    <content type="application/xml">
      <inst:vAppTemplate ...>
        <atom:link rel="self" href="https://.../instances/vAppTemplate:1234"/>
        <atom:link rel="http://schemas.emc.com/rel/uim/vAppTemplate/action/clone"
href="https://.../instances/vAppTemplate:1234/action/clone" />
    ...
```

From this resource representation, we can see that the resource "vAppTemplate:1234" supports the notion of a "clone" action, because its representation contains an atom:link with @rel = "http://schemas.emc.com/rel/uim/vAppTemplate/action/clone" . If the consumer wishes to invoke this operation, it would issue a POST operation to the URL contained in the value of @href (https://.../instances/vAppTemplate:1234/action/clone).

Extending the Relationship Representation with Association Resources

If an EDAA implementation supports deletion of individual members (associations) in a relationship, then it MUST generate a corresponding atom:link element in each atom:entry contained in the relationship's representation containing the URL to which the DELETE can be sent.

Consider an example "ConnectedVia" relationship on a FileServer type. A GET on .../instances/FileServer::{id}/relationships/ConnectedVia returns an atom:feed with an atom:entry for each of the three resources associated with that FileServer resource vie the ConnectedVia relationship -- ie each association.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed ...
  <entry><updated>2011-02-28T23:10:42.580Z</updated>
    <title type="text">Representation of 10.254.131.0</title>
    <id>https:.../instances/IPNetwork::IPNET-10.254.131.0</id>
    <link href="https:.../instances/IPNetwork::IPNET-10.254.131.0" rel="alternate"></link>
    <link href="https:.../instances/a848c3ca-77dc-414f-970f-7ad7406c4913" rel="edit"></link>
    <content type="application/xml">
...
    </content>
  </entry>
  <entry><updated>2011-02-28T23:10:42.580Z</updated>
...
    <link href="https:.../instances/61096aa0-0b8b-40db-89a3-283871efbfbc" rel="edit"></link>
    <content type="application/xml">
...
    </content>
  </entry>
  <entry><updated>2011-02-28T23:10:42.580Z</updated>
...
    <link href="https:.../instances/f31ad8d7-22d8-49fd-9cdb-98ee348efdc7" rel="edit"></link>
    <content type="application/xml">
...
    </content>
  </entry>
```

The additional atom:link element in each entry indicates the URI the consumer should use in order to DELETE the individual associations from the relationship.

Alias URIs

As discussed in the Primer and in particular the Factory Pattern Example there are situations facing the plan author that require him/her to construct URIs to resources that are about to be created by the plan. Clearly the plan author should not anticipate what URI will be generated for those resources, and instead, we provide a mechanism to allow these resources to be aliased.

We define a URI scheme http://schemas.emc.com/msa/Alias/{some value unique to the plan} that allows the plan author to tag various parts of the plan with identifiers that can be referenced elsewhere in the plan. The implementation of the Factory Resource processing the plan will use these Aliased identifiers as placeholders for the final URIs the Factory constructs.

This pattern is similar to what Spring Framework uses for bean identifiers.

An EDAA that supports the notion of a factory pattern style of creating resourcs MUST support the Alias URI convention.

Using PATCH vs PUT

In general, PUT is used to modify the entire state of a resource. Each property that can be modified by consumers (as indicated by the partial representation (for update) associated with the Resource's Type), MUST appear in the body of the PUT request. This restriction is required to maintain the idempotency of PUT operations.

In certain situations, it is not reasonable or convenient for the consumer to specify ALL the properties in a partial representation (for update) of a resource. In these situations PUT SHOULD NOT be used. Rather, the PATCH operation MUST be used.

In an EDAA, if PATCH is supported (look for PATCH as one of the items listed in the response to an OPTIONS request to the resource.), then the consumer MUST include the ETag of the version of the resource's representation that is being patched in an HTTP If-Match: header. An EDAA implementation MUST reject a PATCH request that omits an If-Match header, and it MUST reject the request if the ETag in the If-Match header of the request does not match the EDAA's current version of the resource's representation.

Note, many tools in common use (eg service implementation frameworks, client coding tools, etc.) do not support PATCH. Use of PATCH should be considered only when PUT is not appropriate to solve a particular use case.

EDAA Primer

This topic is a useful starting point for developers trying to understand how to use EDAA.

This topic can be thought of as a "primer" for product architects, designers, product management staff and others to understand how EDAA can help the product team address certain kinds of use cases in a consistent style.

Unlike the normative portions of the EDAA specification, which is aimed at middleware developers, this topic looks at the same material, but from the point of view of someone trying to understand how to adopt the specifications.

Chapters

1. Introduction gives an overview of the EDAA specification, motivation, etc.
2. Example gives an example of how to design a REST API using EDAA
3. Detailed Primer on EDAA details of the various EDAA features

EDAA Primer - Introduction to EDAA

This topic is an introduction to the Data Access API (EDAA). This topic is referenced as a non-normative section of the Base EDAA spec and it is part of the EDAA Primer.

Contents

1 Chapters
2 Introduction
  2.1 Who should read this Primer?
  2.2 What problem is EDAA Addressing?
  2.3 Why should a product want an API to their product?
    2.3.1 APIs based on Broadly Adopted Approaches are better
    2.3.2 It is Tricky to make Good APIs
  2.4 Why use REST for the API Architecture Style?
    2.4.1 Learning about REST
  2.5 Why adopt the EDAA style of REST API
Design?3 EDAA key Tenets
4 How does EDAA fit into Products
  4.1 EDAA + Domain Model = REST API for Product
5 Next Chapter

Chapters

1. **Introduction** gives an overview of the EDAA specification, motivation, etc. **<--- you are here**
2. Example gives an example of how to design a REST API using EDAA
3. Detailed Primer on EDAA details of the various EDAA features

Introduction

This topic can be thought of as a "primer" for product architects, designers, product management staff and others to understand how EDAA can help the product teams build APIs that address certain kinds of use cases in a consistent style.

Unlike the normative portions of the EDAA specification, which is aimed at middleware developers, this topic looks at the same material, but from the point of view of someone trying to understand how to adopt the specifications.

Who should read this Primer?

- Product architects who want to understand good API design for their products
- Product developers who want to understand more context around the EDAA specification
- Product managers attempting to plan how to sequence delivery of functionality over a series of product releases

What problem is EDAA Addressing?

- Improve access of information contained in OpenText products by providing consistent, API styles using widely-adopted web-based distributed computing protocols, standards, techniques and best practices.
- Improve interoperability and consumability of products by defining a standard API style. Products that adopt the conventions defined by EDAA share a common and predictable approach to exposing data and functionality through the API. Consumers that are used to one compliant product's API will be very familiar with the API style exposed by any compliant product.

Why should a product want an API to their product?

APIs based on Broadly Adopted Approaches are better

- In areas, where performance is not fundamentally critical, unlike for example, the IO path to storage, ubiquity of the standards trump other consideration when implementing a function.
  - The more broadly adopted the standard, the easier it is for clients to consume APIs based on those standards
- Broadly adopted approaches, that have broad industry adoption (like http, ATOM, etc) may not be perfect, but they are widely available and are supported in lots of client tools, middleware components, server side libraries, etc.
- Some popular standards/techniques/best practices
  - REST/ROA ATEOAS
  - ATOM
  - JSON
  - CAS
  - AMQP
  - See also RabbitMQ Spring
  - URI Patterns
  - Patch
  - Etag
  -

It is Tricky to make Good APIs

Simple APIs are the best to deliver, but it is pretty tricky to make something simple. Certainly basing a product's API on broadly adopted approaches is a good step towards simplicity, as it minimizes the amount of invention the product team needs to make and is easier to explain to consumers. But there are other considerations important to think about in order to get a GOOD and SIMPLE API:

- How to use REST “predictably” for my product?
- What is the right granularity of API?
- How do I make this simple for developers?
- How do I secure this API?
- How do I handle transactions?
- How do I advertise and document this API?

Why use REST for the API Architecture Style?

We chose REST as the basis for the API style in our work for several reasons:

1. REST is SIMPLE
2. REST is broadly adopted
3. REST is currently the most popular form of distributed computing architectural style
   - Other styles such as CORBA or SOAP/Web Services were considered, but they lack some or all of the characteristics listed above.

Learning about REST

A tutorial on REST is beyond the scope of this primer, however we recommend that the reader be familiar with REST.

Several possibilities for introductory information include:

- Joe Gregorio's (Google) Introduction to REST.
- Wikipedia intro to REST.
  - Google's intro to Atom Syndication Format is also very good first part of this video.
-

Optionally, Richardson/Ruby book on Resource Oriented Design is a good book about REST API design, it is an advanced topic, not necessary but if you want to get to the point of being a REST expert, you have to be familiar with the concepts in this book.

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 OpenText products, allowing developers to learn EDAA once and then be very familiar with the REST API produced by many OpenText products. Google's gdata achieves a similar benefit across the various Google web properties.

EDAA key Tenets

EDAA is a "style" of REST architecture. REST itself is very simple, there are myriad 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 OpenText 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. The notion of defining a consistent pattern of REST API design has been demonstrated by Google's gdata and Microsoft's odata and others.

> ... We hold these truths to be self evident ...

1. There is **no single canonical data model**, but rather, clients need to be provided with enough metadata to be able to determine what kind of information is made available from any particular data source and how to reconcile/correlate information items between data feeds.
2. There is **no single technology base** upon which all products will be built, there is simply too much diversity of requirements, legacy, skills etc. Integration between products must be based upon common interfaces, not common technology choices.
3. We seek to **minimize shared understanding** between any component of the system. This has two sub tenets. First, we want to use as much common, off the shelf design patterns as possible, to minimize surprise and maximize the use of common tools, software components and skill sets. Second, we want to be parsimonious in invention, using as much as possible, commonly accepted best/leading practices from the broader software community.
4. A principle mechanism to share data between components in any software portfolio is by implementing **RESTful interfaces** providing metadata and data access following a predictable, uniform interface.
5. In order to maximize the ease of building interoperable software components, software development should emphasize the use of **Service-Oriented Architecture (SOA) principles**.
6. Interfaces change, we must build software that **accommodates change**. Specifically, we must not build software and interfaces that require complicated, synchronized component release schedules that unnecessarily increase time to market to provide customer value.

With respect to the EDAA design approach:

1. The REST API will be centered around a "resource model". Wherever possible we will adopt a Resource-Oriented Architecture, or ROA approach
2. The principle of "Hypermedia As The Engine of Application State or HATEOAS provides a means of isolating the dependency of consumers on particular URI choices made by the service.
   - We will encourage the resource representations returned by REST API invocations to use "link" elements to indicate possible operations that can be performed on the resource and related resources that may be of interest to the consumer. This "navigability" or "interconnectedness" of a resource model is important property of a REST API.
3. We encourage the use of Atom Syndication Format and related specifications for consistent collections of resources and standardized pagination
   - For consumers that prefer JSON, we have defined an equivalent JSON representations for all aspects of EDAA.

How does EDAA fit into Products

The idea with EDAA is to establish a consistent style of REST API design across OpenText products. 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 OpenText products, we simplify the task of any consumer using two or more OpenText software products.

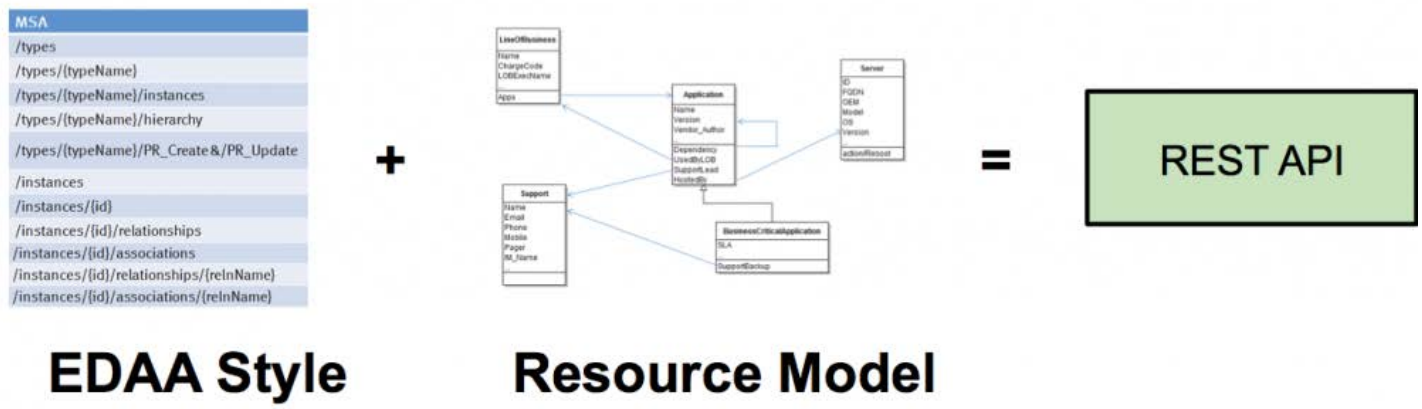
Consider the following analogy:

REST then, is like painting in the impressionist style; this style has lots of different approaches and yields a surprising variety of different painting styles all under the impressionist label. EDAA, then, tries to standardize the approach to REST API design. So the analogy continues that if REST API design is like impressionist painting, EDAA imposes boundaries on the style of REST API design such that all product REST API designs following the EDAA approach appear like they are designed by the same creative force -- as if we all painted like Vincent Van Gogh.

EDAA + Domain Model = REST API for Product

EDAA itself is not a REST API, it is a "style" of REST API design. The idea with EDAA is that a product team takes the EDAA approach, applies it to the data and functionality (resource model) of their product and the result is the REST API to their product that has the consistent characteristics embodied in EDAA.

The following figure describes this equation:



Note also that there could be multiple resource models per product. For example, a product might choose to design a highly abstracted and simplified representation of the product's data and functionality to present to a set of consumers that value simplicity of operation (for example in a cloud service provider market). The product team would design a resource model and use EDAA to present a REST API targeted towards that kind of consumer. The same product could expose another REST API focusing on a sophisticated administrator who needs the complexity and functionality available in the product. This richer functionality would be another resource model exposed by the product and would be the basis of another REST API against the product, tailored to the more sophisticated admin consumer. One product, two resource models, two EDAAs.

Next Chapter

Example is the next chapter.

Public EDAA Primer Example

This topic is an example of using the EDAA to design an API to an application. This topic is part of the EDAA Primer.

Contents

1 Chapters
2 Applying EDAA to Design a REST API to a Product
  2.1 ODAAs and Products
3 Approach
  3.1 Understand the use cases
  3.2 Determine which ODAAs will be built
  3.3 Design the resource model
  3.4 Map the resource model to the URI patterns and resource representations
  3.5 Implement using shared components
  3.6 Securing the interfaces
4 Example: Applying EDAA to "Thin Ice Enterprises"
  4.1 Understand the use cases
  4.2 Determine which ODAAs will be built
  4.3 Design the resource model
    4.3.1 Type Declarations
    4.3.2 Notes on the resource model
  4.4 Map the resource model to the URI patterns and resource representations
  4.5 Implement using shared components
  4.6 Securing the interfaces
5 Next Chapter

Chapters

1. Introduction gives an overview of the EDAA specification, motivation, etc.
2. **Example** gives an example of how to design a REST API using EDAA **<--- you are here**
3. Detailed Primer on EDAA details of the various EDAA features

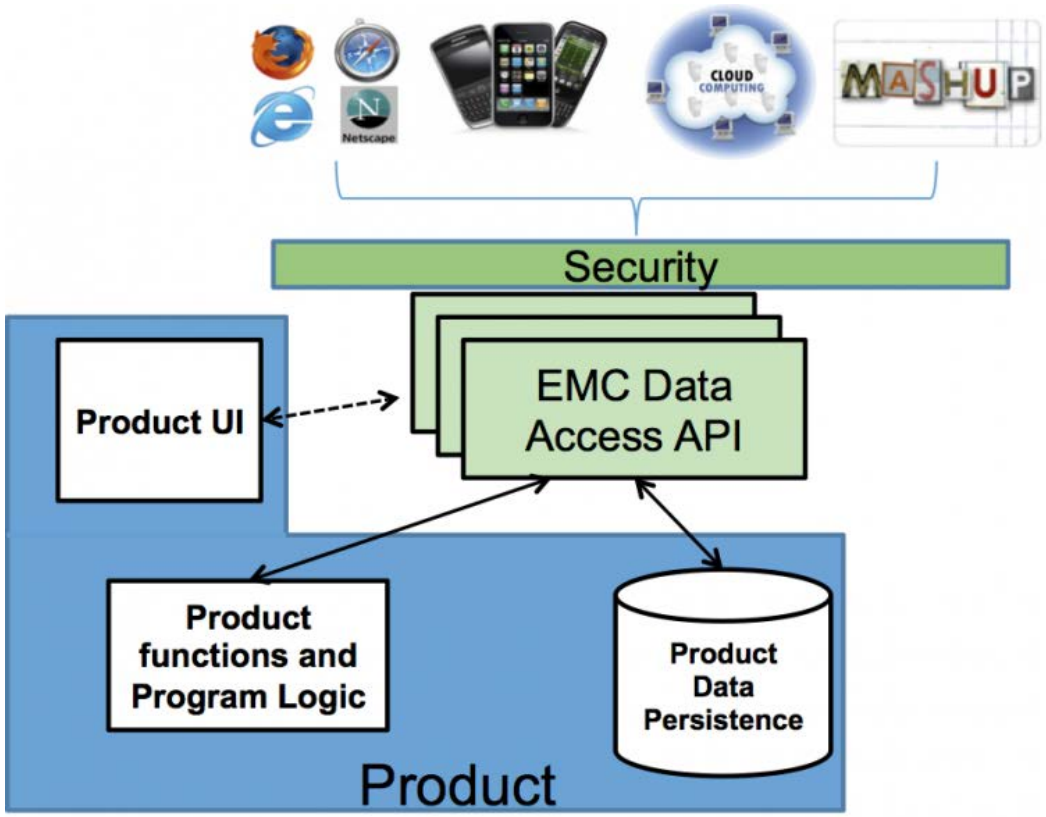
Applying EDAA to Design a REST API to a Product

Before we get into details about what are the features, best practices, patterns, etc. called out by the EDAA spec, lets discuss how a product group could think about EDAA APIs for their product, outline an approach to designing the APIs for the product. Following that, lets walk through an example of applying EDAA to a problem domain.

ODAAs and Products

The idea behind EDAA is to use the "Adapter Pattern" to layer a REST API on top of the product, imposing little or no change to the underlying product code. Basically, the approach taken by many products is to have 1 or more ODAAs for their product, each adapting a particular model of underlying resources manipulated by the product. The product may have 2 or 3 different abstractions it wishes to present to different consumers. For example, consider an IT management product that would present a fine grained, lots of knobs and dials abstraction suitable to traditional IT infrastructure management, and a highly abstracted, coarse grained, simplified interface for consumers looking for operational efficiency, like a Service Provider customer.

This notion of using one or more ODAAs layered on top of existing product code is shown in the following figure:



The idea is that existing product functionality, eg the product's UI, the product's core logic and behavior, and the product's core data source do not need to be changed when an EDAA is added. The source of the data is important. Does the product use a data persistence mechanism like an RDF tuple store or a traditional RDBMS? Great, that will support an EDAA just fine. Does the product get its core data by making CLI calls out to some "element manager"? That will work for building an EDAA too, although it might be a challenge to build some of the advanced features in EDAA.

One or more EDAA interfaces can be layered on top of the product's core logic and data source. For any EDAA the product team chooses to design, the bulk of the work is to agree the resource model (the set of entities, attributes, relationships and actions) that will be exposed by the EDAA. Once this design is done, it is a matter of taking some REST middleware (eg RESTLet or Spring MVC), and filling in the REST API logic to dispatch the URI patterns to controller logic that knows how to implement the semantics of the URI pattern in terms of the product's data source.

The end result is that new APIs to the product are now available to a large range of new modalities of client access. Browser based Flex/Flash or Javascript clients can take advantage of the EDAA to the products. Apps for Mobile devices, through a browser or otherwise, are easier to build when the UIs they present are loosely coupled from the product via EDAA. PaaS environments, like force.com or Google App Engine are easier to build applications on when they can use EDAA interfaces to access product data and functionality. Mashups of course are tailor made to consume data and functionality surfaced by a product's EDAAs.

Of course, all the access from the consumers goes through some authentication mechanism protecting the product's data and functionality from unauthorized use.

So, hopefully you see the notion of EDAA as an "adapter pattern" in the above discussion. The various modalities of client use one of the EDAAs to the product to access product data and functionality. The EDAA brokers the interaction between the client and the product. Northbound, the interface to the client presented by the EDAA is a standardized, consistent REST API. The client needs to understand the resource model presented by the product, but it does not need to understand a product specific API protocol. Southbound from the EDAA, the interface is built to speak to the product using whatever product-specific API available. Overtime (and we are seeing this with several products right now) the product's UI evolves to use the EDAA to access product data and functionality, loosening the coupling between the product UI and the underlying product logic and data access mechanisms.

Approach

What follows is not really a recipe, but is more like a guideline for a set of steps on approaching the problem of applying IMG Interop congress specs to achieve EDAA compatible APIs to a product.

Note also that we are not recommending this approach as a waterfall model. Rather, we emphasize an iterative approach, cycling through some of these steps repeatedly until the EDAA API(s) to the product are designed and implemented.

Understand the use cases

- No magic here, use cases should drive the decisions on EDAA use and API design
- What data do consumers want to see? How can that data be made available to them?
- Can consumers create new resources? Can existing resources be changed or deleted by consumers?
- Are there additional actions, like active management that need to be supported?
- Care needs to be taken that the use cases guide the kinds of resource model and functionality surfaced through the EDAAs of the product and do not inadvertently constrain the APIs.
  - When designing for mashups, it is impossible to predict all the different detailed ways that consumers will want to access and manipulate the data
  - The use cases suggest categories of access, but implement in as general a way as possible, that satisfy the known use cases and do not overly constrain what consumers can do through the EDAA APIs

Determine which EDAAs will be built

- Are there different levels of abstraction of the resource model that need to be presented to different kinds of consumer?
- Are different subsets of data to be presented to different classes of consumer?
- Does the consumer only read data, or is it possible to also allow consumers to create/update/delete resources?

Design the resource model

- Second only to the use case effort, this is where care should be taken to come up with the set of entities (resources) and for each entity, the set of attributes, relationships and actions.
- This is fundamentally an information modeling exercise, balancing the data access needs of consumers, as illuminated by use cases, against the kind of data and data access mechanisms available in the existing product.
- For each entity, it is important that the designer clarify what the identifier scheme will be.
  - Often, an attribute like "name" is a unique identifier (for any value of the attribute, there is at most one resource instance containing that value) Sometimes, an artificial
  - attribute like "id" needs to be added specifically to implement an identifier scheme.
  - We will see why an identifier scheme is important when we consider how the resource model dovetails with the EDAA URI patterns to form the API to the product.
- Ideally, the resource model is iteratively defined by a team combining product experts, domain experts (representing "voice of the customer"), and resource modeling experts.
- Usually, the deliverable from this step is a set of UML diagrams (class diagrams) or other representations of the resource types in the model
- It is a good idea to exit this step with a strategy on how the resource model relates to the underlying data source within the product

Map the resource model to the URI patterns and resource representations

- The way EDAA is designed, to use URI patterns, allows the URIs in the API to be driven by the resource model.
- This mapping is pretty straight forward
- It is often worthwhile to "try out" the EDAA API derived by combining the EDAA URI patterns and the product's resource model, to make sure that key use cases are nicely and conveniently supported by invocations of one or more EDAA API operations.

Implement using shared components

- Start building the EDAA API as soon as possible.
- Use a REST framework (there are many to choose from, including RESTLet or Spring MVC).

Securing the interfaces

- Make sure your team has access to 1 or more people with good security background and understand how to use the chosen security mechanisms.

Example: Applying EDAA to "Thin Ice Enterprises"

Note, this example is based on a fictitious company. Furthermore, the domain chosen is purposefully NOT in the IT Infrastructure Management domain. Although EDAA was designed with OpenText products in mind, most of the principles in EDAA are not specific to IT Infrastructure management. We chose something in a completely different domain largely to avoid readers being distracted by thoughts like "Storage management doesn't work like that", or "no way that is a pragmatic way to model Storage Arrays". In order to provide a simple model for pedagogical purposes, and to avoid domain specific arguments, we chose a domain completely outside IT.

We chose as our example domain, information about Hockey. (note of disclosure, the author is a hockey nut).

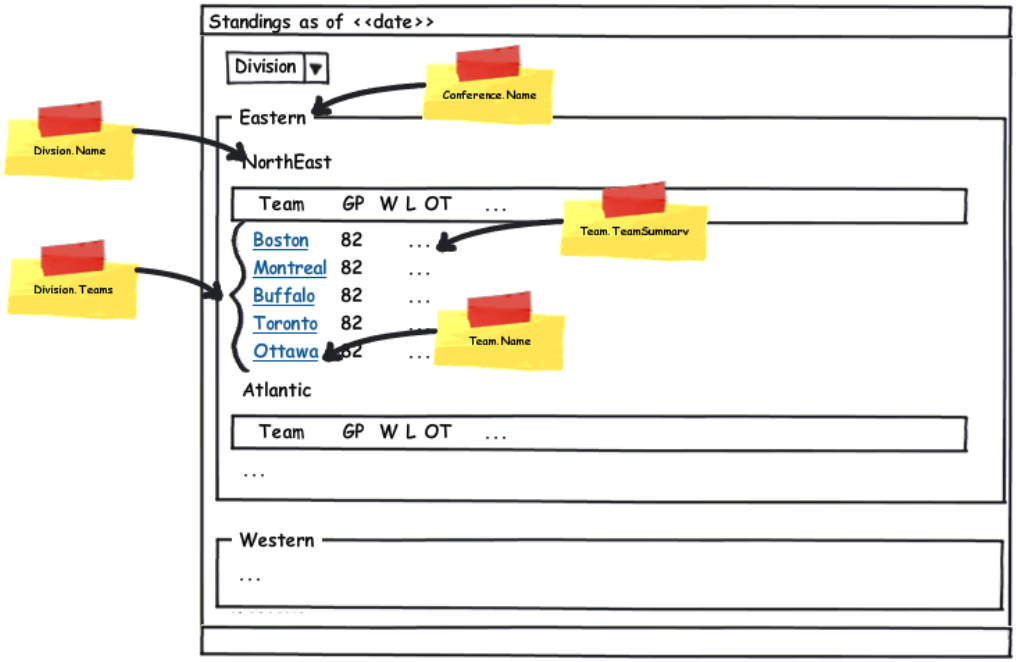
"Thin Ice Enterprises" (TIE) is a fictitious company that wishes to build a data source about hockey teams, hockey players and statistics associated with the game. The service "TIE" wishes to provide is a basic hockey information source, that can be consumed by its customers (eg sports websites, sports gambling services, sports news syndication, player scouting and ranking agencies, etc.). Basically, TIE wants to be a source of basic hockey facts that can be used in mashups. TIE will also build a simple browser based UI, while boring (if it is intellectually coherent to combine the concepts "hockey" and "boring"), it provides nice visuals of the data that is available for TIE's customers to mashup. For the sake of simplicity for this example, we will focus on use cases derived from the "simple" UI TIE wants to build.

Understand the use cases

For the sake of simplicity, we will describe the use cases in terms of UI snippets. Note, this is typically just one of many sources for use cases, but for the sake of this example, these should be enough to get a "feel" for how the use cases drive other steps.

1. Show a list of conferences and the teams within each conference sorted by points, ie show the "standings" in each conference

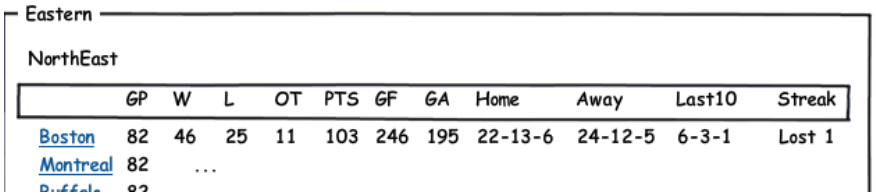
Mockup of UI is given here



2. Show details of a team's record

Basically the team's record, including games played, wins, losses, overtime losses, total points, goals for, goals against, home record (W-L-OT format), away record (W-L-OT), record over the last 10 games (W-L-OT) and current streak (ie Lost n games or Won n games)

Mockup of UI is given here

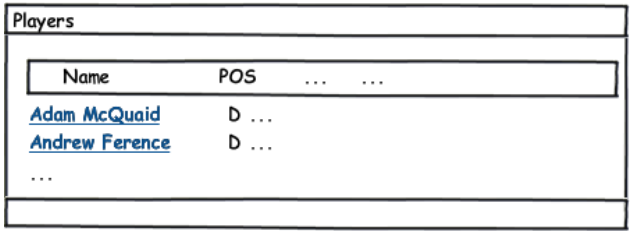


3. Update a team's record

It must be possible for authorized consumers, such as league officials, to update a team's record based on official results of games.

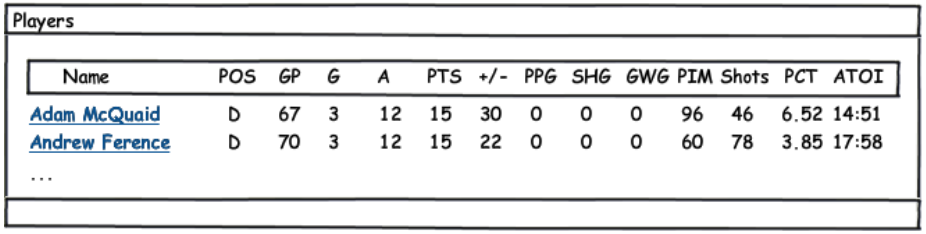
4. List the players currently on a team's roster. Show only active players, but allow for injured players or minor-league players to be listed if requested by the consumer

Mockup of the UI is given here



5. Report the current statistics for any given player, including games played, goals, assists, total points, plus/minus rating, powerplay goals, short handed goals, game winning goals, penalty minutes, shots and average time spent on ice during games. Note, for goaltenders, the list of stats is slightly different, games played, goals against, goals against average, saves, save percentage, shut outs, empty net goals given up, wins, losses, overtime losses, minutes on ice, average time on ice, penalty minutes, goals and assists.

Mockup of the UI is given here



6. Update a player's record

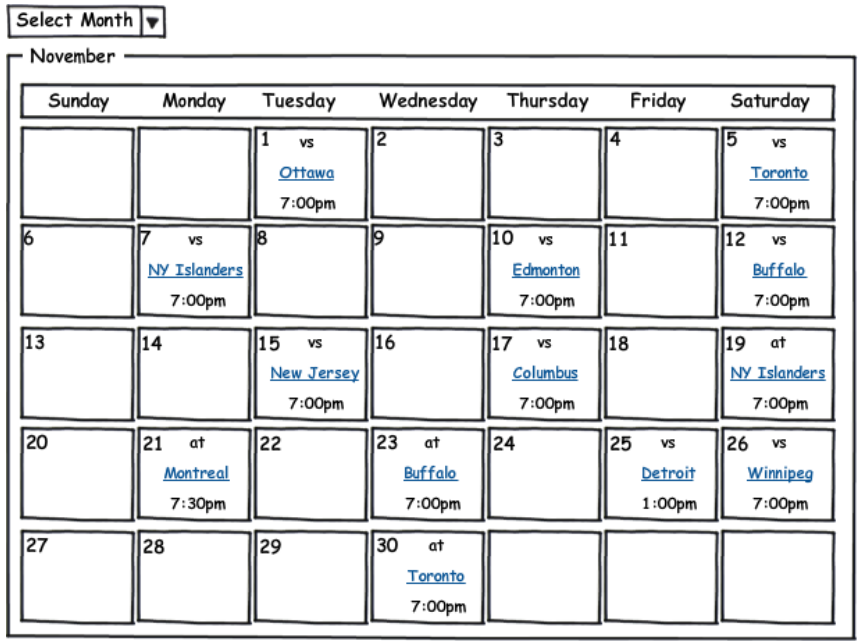
It must be possible for authorized consumers, such as league officials to update a player's record based on official results of games.

7. A consumer can retrieve a list of players for a team containing only those players that play a particular position

This is a filtering request, like give me a list of all the defencemen on a given team.

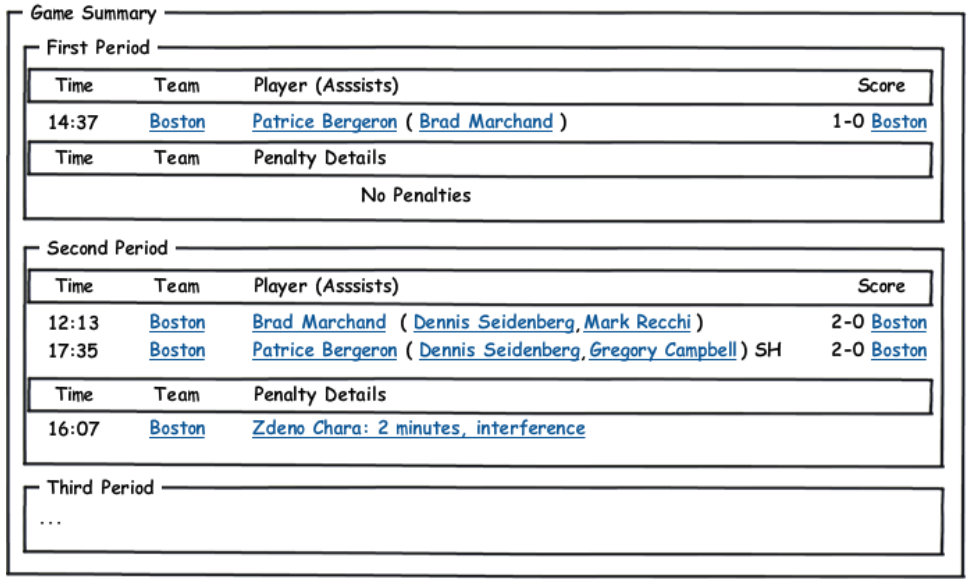
8. Request a team's schedule of games, by month

Mockup of the UI is given here



9. Request a summary of a given game

Mockup of the UI is given here



10. Update a game summary by adding a new GameSummary resource to a given Game resource's summary

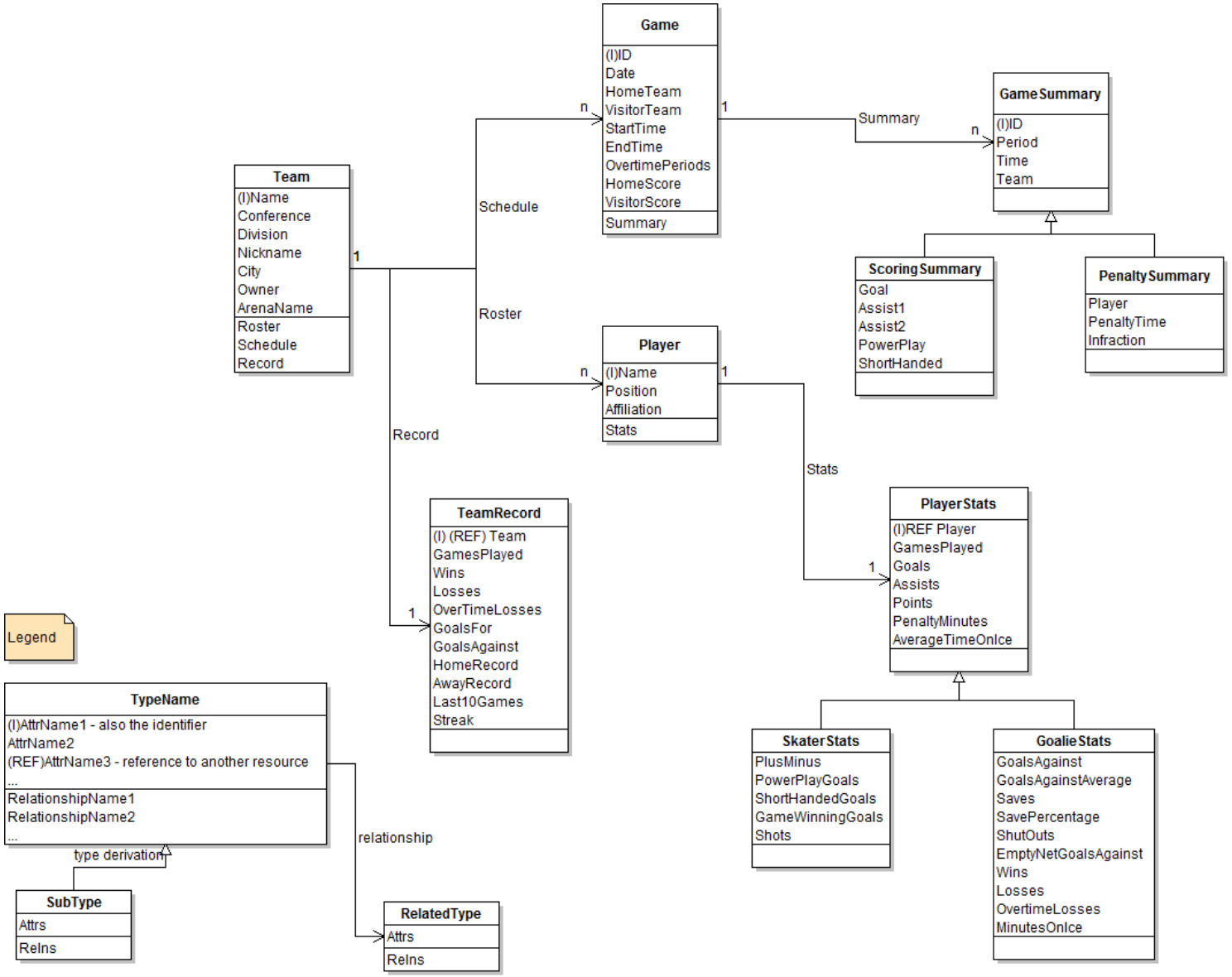
It must be possible for authorized consumers, such as league officials to update a game's summary based on official results of the game.

Determine which EDAAs will be built

For the sake of simplicity, TIE will provide one EDAA, exposing the resource model defined below.

Design the resource model

Based on an analysis of the use cases, TIE has settled on the following resource model:



Type Declarations

See here for the type declarations corresponding to the entities in the figure.

Notes on the resource model

This resource model is a pretty straight forward one. It exhibits a nice number of entities (6) two of which have some specialization. Some products have resource models that number in the hundreds. We kept this example small so that it would be reasonably digestible.

Usually, the resource model directly translates into VS-XML representations. The translation of this one into VS-XML is straight forward.

Note the appearance of type hierarchies, GameSummary, for example, is specialized by ScoringSummary and PenaltySummary. We wanted to include at least one type hierarchy in the example. This means, that the properties defined in the parent type (the attributes named Period, Time and Team) are "inherited" in the specializations, ie ScoringSummary instances also contain Period, Time and Team attributes in addition to the properties defined in the ScoringSummary type. A similar situation holds for the type hierarchy rooted by the PlayerStats type.

Note also the relationship between Team and TeamRecord. These kinds of 1-1 relationships can be modeled as a relationship, as shown in this resource model. We could also have modeled this with an attribute, ie define an attribute property of Team whose type is a complex type defined by TeamRecord. This mechanism of collapsing a 1-1 relationship "in line" into an attribute is convenient for read operations, and can be done if there is no reason to have the related resource (eg TeamRecord) be modeled as a stand alone resource. However, in our case, to support the use case where a league official can update a Team's TeamRecord, we chose to model TeamRecord as a separate entity. This is an example of doing a tradeoff for simplicity of the update operation at the cost of read operations needing to traverse the relationship. Note also that by using the expand relationships query parameter, the consumer can request this information be automatically "in lined", and therefore we can get the "best of both worlds".

The identifier schemes chosen for the entities are pretty straight forward. Conference, Division and Team are uniquely identified by a Name attribute, mind you, some care must be taken to make sure the team names are unique. Happily, there are very few instances of each of these types and they very rarely change (eg one change since mid-2011, as the Atlanta team moved to Winnipeg).

The identifier for TeamRecord is the same as Team, as there is a 1:1 relationship. The actual attribute property of TeamRecord, named Team, can be modeled as a URI reference to the corresponding team (link style) or contain just the contents of the Team's identifier, ie a Name value (foreign key style). Of course there is an approach that can be considered that the entity itself doesn't have a property that corresponds to the identity scheme, but rather, relies on business logic to support an {id} style like "TeamRecord::Boston" to identify the TeamRecord instance that corresponds to the Team identified by the Name "Boston". This is perfectly legal and does not require the property to be declared on the TeamRecord type.

The Game type has an artificial identifier associated with it, named ID. In this case, the ID property is composed by concatenating the date (in YYYYMMDD format) with the name of the home team and the name of the visiting team. For example, for the game scheduled on November 1, 2011 Ottawa visiting Boston, the value of ID would be 20111101OttawaBoston. An representation of this Game could be retrieved by doing a GET /instances/Game::20111101OttawaBoston.

Instances of GameSummary are identified another artificial ID. Because a given Game is associated with zero or more GameSummary instances, the ID of a GameSummary is the ID of the Game concatenated with an integer counter starting at zero. So the GameSummary resource that occurred earliest in the game between Ottawa and Boston on November 1, 2011 would have an identifier 20111101OttawaBoston0.

Because a PlayerStats type is in a 1-1 relationship with Player, we could use the link style (ie a URI to the player) or foreign key style (eg just the Name of the player) to associate the PlayerStats resource instance with the Player instance it corresponds to.

Map the resource model to the URI patterns and resource representations

Lets walk through how the resource model combine with the EDAA patterns combine to address the identified use cases.

1. Show a list of conferences and the teams within each conference sorted by points, ie show the "standings" in each conference

A list of Teams can be retrieved by GET /types/Teams/instances. This will return a feed of the Team instances

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE /msa/types/Team/instances</title>
  <updated>2011-07-03T11:51:55-05:00</updated>
  <author><name>msa framework</name></author>
  <id>aaabb570-340e-4e6d-adaf-91c38c5e7aaa</id>
  <link rel="self" href="https://tie.com/msa/types/Team/instances"/>
  <entry>
    <title type='text'>Team - Anaheim</title>
    <id>https://tie.com/msa/instances/Team::Anaheim</id>
    <updated>2011-07-03T11:51:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Team::Anaheim' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Team' />
    <content type='application/xml'>
      <inst:Team xmlns:atom='http://www.w3.org/2005/Atom'
```

```
                xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>Anaheim</inst:Name>
        <inst:Division>Pacific</inst:Division>
        <inst:Conference>Western</inst:Conference>
        <inst:City>Anaheim</inst:City>
        <inst:Owner>Harry Samueli</inst:Owner>
        <inst:ArenaName>Honda Center</inst:ArenaName>

        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Roster'
                   href='https://tie.com/msa/instances/Team::Anaheim/relationships/Roster' />
        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Schedule'
                   href='https://tie.com/msa/instances/Team::Anaheim/relationships/Schedule' />
        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Record'
                   href='https://tie.com/msa/instances/Team::Anaheim/relationships/Record' />
      </inst:Team>
    </content>
  </entry>
  <entry>
    <title type='text'>Team - Boston</title>
    <id>https://tie.com/msa/instances/Team::Boston</id>
    <updated>2011-07-03T11:51:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Team::Boston' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Team' />
    <content type='application/xml'>
      <inst:Team xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>Boston</inst:Name>
        <inst:Division>NorthEast</inst:Division>
        <inst:Conference>Eastern</inst:Conference>
        <inst:City>Boston</inst:City>
        <inst:Owner>Jeremy Jacobs</inst:Owner>
        <inst:ArenaName>TD Garden</inst:ArenaName>

        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Roster'
                   href='https://tie.com/msa/instances/Team::Boston/relationships/Roster' />
        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Schedule'
                   href='https://tie.com/msa/instances/Team::Boston/relationships/Schedule' />
        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Record'
                   href='https://tie.com/msa/instances/Team::Boston/relationships/Record' />
      </inst:Team>
    </content>
  </entry>
  <entry>
    <title type='text'>Team - Calgary</title>
    <id>https://tie.com/msa/instances/Team::Calgary</id>
    <updated>2011-07-03T11:51:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Team::Calgary' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Team' />
    <content type='application/xml'>
      <inst:Team xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>Calgary</inst:Name>
        <inst:Division>NorthWest</inst:Division>
        <inst:Conference>Western</inst:Conference>
        <inst:City>Calgary</inst:City>
        <inst:Owner>N Murray Edwards</inst:Owner>
        <inst:ArenaName>Pengrowth Saddledome</inst:ArenaName>

        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Roster'
                   href='https://tie.com/msa/instances/Team::Calgary/relationships/Roster' />
        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Schedule'
                   href='https://tie.com/msa/instances/Team::Calgary/relationships/Schedule' />
        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Team/relationship/Record'
                   href='https://tie.com/msa/instances/Team::Calgary/relationships/Record' />
      </inst:Team>
    </content>
  </entry>

  ... etc. for the other 27 teams in the National Hockey League

</feed>
```

This feed will not be sorted at all. The consumer would need to do client-side sorting of this list, including client side sorting of teams into divisions and divisions into conferences. There would also need to be client-side sorting of teams within division based on the TeamRecord of each Team.

If the TIE EDAA supported the ?orderby optional query parameter, the sorting of teams into divisions and divisions into conferences could be done by: GET /types/Team/instances?orderby=Conference,Division, which would format the feed of Team instances so that the entry elements within the feed were sorted by the value of Team.Conference, and within that sorting, further sorted by Team.Division.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
```

```
  <title>TIE /msa/types/Team/instances?orderby=Conference,Division</title>
  <updated>2011-07-03T11:51:55-05:00</updated>
  <author><name>msa framework</name></author>
  <id>bbbbb570-340e-4e6d-adaf-91c38c5e7bbb</id>
  <link rel="self" href="https://tie.com/msa/types/Team/instances?orderby=Conference,Division"/>

  ... the first set of teams are in the Eastern Conference

  ... the Atlantic Division comes first

  <entry>
    <title type='text'>Team - New Jersey</title>
    <id>https://tie.com/msa/instances/Team::New%20Jersey</id>
    <updated>2011-07-03T11:51:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Team::New%20Jersey' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Team' />
    <content type='application/xml'>
      <inst:Team xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>New%20Jersey</inst:Name>
        <inst:Division>Atlantic</inst:Division>
        <inst:Conference>Eastern</inst:Conference>
...
      </inst:Team>
    </content>
  </entry>
  <entry>
    <title type='text'>Team - New Jersey</title>
    <id>https://tie.com/msa/instances/Team::New%20Jersey</id>
    <updated>2011-07-03T11:51:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Team::New%20Jersey' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Team' />
    <content type='application/xml'>
      <inst:Team xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>New%20Jersey</inst:Name>
        <inst:Division>Atlantic</inst:Division>
        <inst:Conference>Eastern</inst:Conference>
...
      </inst:Team>
    </content>
  </entry>
  <entry>
    <title type='text'>Team - NY Islanders</title>
    <id>https://tie.com/msa/instances/Team::NY%20Islanders</id>
    ...
  </entry>

  ... etc. for the other 3 teams in the Atlantic Division of the Eastern Conference

  ... 5 entries, one for each team in the NorthEast Division of the Eastern Conference

  ... 5 entries, one for each team in the SouthEast Division of the Eastern Conference

  ... 5 entries, one for each team in the Central Division of the Western Conference

  ... 5 entries, one for each team in the NorthWest Division of the Western Conference

  ... 5 entries, one for each team in the Pacific Division of the Western Conference

</feed>
```

The final list of team instances would still need to be sorted by Points (logic to calculate Points = 2*TeamRecord.Wins + 1*TeamRecord.OverTimeLosses). We could have made this easy and changed the data model to cache the Points property as an attribute of Team. Had we done this, then it would have been very simple for the client to get a list of teams in a division sorted in order by Points: GET /types/Team/instances?orderby=Conference,Division,Points

2. Show details of a team's record

Most of the necessary information is stored in TeamRecord instances. To get the record for a given team, first retrieve the representation of the Team, eg GET /instances/Team::Boston and then traverse the atom:link to the TeamRecord relationship (GET /instances/Team::Boston/relationships/Record).

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE /msa/instances/Team::Boston/relationships/Record</title>
  <updated>2011-07-03T12:16:20-05:00</updated>
  <author><name>msa framework</name></author>
  <id>cccbb570-340e-4e6d-adaf-91c38c5e7ccc</id>
  <link rel="self" href="https://tie.com/msa/instances/Team::Boston/relationships/Record"/>
  <entry>
    <title type='text'>TeamRecord - Boston</title>
    <id>https://tie.com/msa/instances/TeamRecord::Boston</id>
    <updated>2011-07-03T12:16:20-05:00</updated>
```

```
    <link rel='self' href='https://tie.com/msa/instances/Team::Boston/relationships/Record' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/TeamRecord' />
    <link rel='edit' href='https://tie.com/msa/instances/TeamRecord::Boston' />
    <atom:content type='application/xml'>
      <inst:TeamRecord xmlns:atom='http://www.w3.org/2005/Atom'
                       xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Team>https://tie.com/msa/instances/Team::Boston</inst:Team>
        <inst:GamesPlayed>81</inst:GamesPlayed>
        <inst:Wins>46</inst:Wins>
        <inst:Losses>24</inst:Losses>
        <inst:OverTimeLosses>11</inst:OverTimeLosses>
        <inst:GoalsFor>245</inst:GoalsFor>
        <inst:GoalsAgainst>193</inst:GoalsAgainst>
        <inst:HomeRecord>22-12-6</inst:HomeRecord>
        <inst:AwayRecord>24-12-5</inst:AwayRecord>
        <inst:Last10Games>6-3-1</inst:Last10Games>
        <inst:Streak>Won 1</inst:Streak>
      </inst:TeamRecord>
    </atom:content>
  </entry>
</feed>
```

Of course, if the TIE EDAA supported expanding relationships (the ?expand= query parameter), then GET /instances/Team::Boston?expand=* would inline the Record relationship (and the other relationships) and relieve the client from doing additional GET operations. Note in the following representation of Team::Boston the single atom:link elements for each relationship (Roster, Schedule and Record) is replaced with an atom:link element containing child content. The child content of each atom:link is essentially the feed that would be retrieved if the value of @href was resolved.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE /msa/instances/Team::Boston?expand=*</title>
  <updated>2011-07-03T12:31:47-05:00</updated>
  <author><name>msa framework</name></author>
  <id>dddbb570-340e-4e6d-adaf-91c38c5e7ddd</id>
  <link rel="self" href="https://tie.com/msa/instances/Team::Boston/expand=*"/>
  <entry>
    <title type='text'>Team - Boston</title>
    <id>https://tie.com/msa/instances/Team::Boston</id>
    <updated>2011-07-03T11:51:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Team::Boston' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Team' />
    <content type='application/xml'>
      <inst:Team xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>Boston</inst:Name>
        <inst:Division>NorthEast</inst:Division>
        <inst:Conference>Eastern</inst:Conference>
        <inst:City>Boston</inst:City>
        <inst:Owner>Jeremy Jacobs</inst:Owner>
        <inst:ArenaName>TD Garden</inst:ArenaName>

        <atom:link href="https:tie.com/instances/Team::Boston/relationships/Roster"
              rel="http://schemas.tie.com/msa/Team/relationships/Roster">
          <ae:inline xmlns:ae="http://schemas.emc.com/atom/ext/">
            <atom:feed ...
             ... a paginated feed of all the players on Boston's roster
            </atom:feed>
          </ae:inline>
        </atom:link>

        <atom:link href="https:tie.com/instances/Team::Boston/relationships/Schedule"
              rel="http://schemas.tie.com/msa/Team/relationships/Schedule">
          <ae:inline xmlns:ae="http://schemas.emc.com/atom/ext/">
            <atom:feed ...
             ... a paginated feed of all the Games on Boston's schedule
            </atom:feed>
          </ae:inline>
        </atom:link>

        <atom:link href="https:tie.com/instances/Team::Boston/relationships/Record"
              rel="http://schemas.tie.com/msa/Team/relationships/Record">
          <ae:inline xmlns:ae="http://schemas.emc.com/atom/ext/">
            <atom:feed ...
              <atom:title>TIE /msa/instances/Team::Boston/relationships/Record</atom:title>
              <atom:updated>2011-07-03T12:16:20-05:00</atom:updated>
              <atom:author><name>msa framework</atom:name></atom:author>
              <atom:id>cccbb570-340e-4e6d-adaf-91c38c5e7ccc</atom:id>
              <atom:link rel="self" href="https://tie.com/msa/instances/Team::Boston/relationships/Record"/>
              <atom:entry>
                <atom:title type='text'>TeamRecord - Boston</atom:title>
                <atom:id>https://tie.com/msa/instances/Team::Boston/relationships/Record</atom:id>
                <atom:updated>2011-07-03T12:16:20-05:00</atom:updated>
                <atom:link rel='self' href='https://tie.com/msa/instances/TeamRecord::Boston' />
                <atom:link rel='edit' href='https://tie.com/msa/instances/TeamRecord::Boston' />
```

```
              <atom:link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/TeamRecord' />
                <atom:content type='application/xml'>
                  <inst:TeamRecord xmlns:atom='http://www.w3.org/2005/Atom'
                                   xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

                    <inst:Team>https://tie.com/msa/instances/Team::Boston</inst:Team>
                    <inst:GamesPlayed>81</inst:GamesPlayed>
                    <inst:Wins>46</inst:Wins>
                    <inst:Losses>24</inst:Losses>
                    <inst:OverTimeLosses>11</inst:OverTimeLosses>
                    <inst:GoalsFor>245</inst:GoalsFor>
                    <inst:GoalsAgainst>193</inst:GoalsAgainst>
                    <inst:HomeRecord>22-12-6</inst:HomeRecord>
                    <inst:AwayRecord>24-12-5</inst:AwayRecord>
                    <inst:Last10Games>6-3-1</inst:Last10Games>
                    <inst:Streak>Won 1</inst:Streak>
                  </inst:TeamRecord>
                </atom:content>
              </atom:entry>
            </atom:feed>
          </ae:inline>
        </atom:link>

      </inst:Team>
    </content>
  </entry>
</feed>
```

3. Update a team's record

The consumer (in this case a league official) should retrieve the TeamRecord for the team (eg do a GET on /instances/Team::{Team.Name}) and follow the link to the TeamRecord relationship. The TeamRecord for Boston is shown in the various example results shown in the solution to the previous use case.

The consumer would then make whatever updates to the properties are necessary, eg increment the TeamRecord.Wins attribute, update the TeamRecord.GoalsFor, etc.

The consumer would then PUT /instances/TeamRecord::Boston (recall that the identifier scheme for TeamRecord is the Team's name) with the body of the message containing the updated attributes (all of them). Here is the contents of the body of the POST. Note the atom:link in the TeamRecord entry that has @rel="edit", this entry indicates the URL to use to make changes to the TeamRecord. The body of the PUT message contains:

```
<inst:TeamRecord xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>
  <inst:GamesPlayed>82</inst:GamesPlayed>
  <inst:Wins>46</inst:Wins>
  <inst:Losses>25</inst:Losses>
  <inst:OverTimeLosses>11</inst:OverTimeLosses>
  <inst:GoalsFor>246</inst:GoalsFor>
  <inst:GoalsAgainst>195</inst:GoalsAgainst>
  <inst:HomeRecord>22-13-6</inst:HomeRecord>
  <inst:AwayRecord>24-12-5</inst:AwayRecord>
  <inst:Last10Games>6-3-1</inst:Last10Games>
  <inst:Streak>Lost 1</inst:Streak>
</inst:TeamRecord>
```

Note that even those properties that have not changed are included in the PUT message. This is because in REST, the PUT operation is idempotent.

If TIE supported the PATCH operation, then a subset of the properties of TeamRecord could be updated. A PATCH /instances/TeamRecord::Boston could contain a couple of fewer elements:

```
<inst:TeamRecord xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>
  <inst:GamesPlayed>82</inst:GamesPlayed>
  <inst:Losses>25</inst:Losses>
  <inst:GoalsFor>246</inst:GoalsFor>
  <inst:GoalsAgainst>195</inst:GoalsAgainst>
  <inst:HomeRecord>22-13-6</inst:HomeRecord>
  <inst:Streak>Lost 1</inst:Streak>
</inst:TeamRecord>
```

Note, eTags are very useful in update situations. Had TIE chosen to support eTags, it would have a very nice way to ensure that multiple writers didn't overwrite each other's updates. Assume for a moment that it is possible for multiple consumers to update a TeamRecord instance. In this case, TIE would need to support eTag. With eTag support, the consumer still needs to do a GET operation on the TeamRecord:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom"
```

```
      xmlns:gd='http://schemas.google.com/g/2005'
      gd:etag='W/"AAroqXcycSp7ImA9WxRVDdk."'>
  <title>TIE /msa/instances/Team::Boston/relationships/Record</title>
  ...
  <entry gd:etag='"BBB2F2drp7tKA7QxRD2Ko."'>
    <title type='text'>TeamRecord - Boston</title>
    ...
  </entry>
</feed>
```

The response looks almost exactly the same as we showed previously, but this time, there is an eTag associated with the TeamRecord in the atom:feed element and the atom:entry element. This eTag would be used on the subsequent PUT operation:

```
PUT /instances/TeamRecord::Boston
If-Match: "BBB2F2drp7tKA7QxRD2Ko."

<inst:TeamRecord xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>
  <inst:GamesPlayed>82</inst:GamesPlayed>
  <inst:Losses>25</inst:Losses>
  <inst:GoalsFor>246</inst:GoalsFor>
  <inst:GoalsAgainst>195</inst:GoalsAgainst>
  <inst:HomeRecord>22-13-6</inst:HomeRecord>
  <inst:Streak>Lost 1</inst:Streak>
</inst:TeamRecord>
```

The EDAA implementation is obliged to compare the eTag given in the PUT with the eTag computed from the current state of the resource. If no writer has made any updates in the time between when our consumer did the GET operation and the PUT operation, the eTag given in the PUT operation will match the eTag generated from the current state of the resource, and the resource update can proceed. However, in the case where some other consumer snuck in and made a modification between the time our consumer did the GET and the PUT, then the eTags will NOT match and our consumer's update will fail. A bit of a bummer for our consumer, but at least he/she is assured that any modifications he/she makes will not be accidentally lost.

4. List the players currently on a team's roster. Show only active players, but allow for injured players or minor-league players to be listed if requested by the consumer

This use case shows the use of the ?filter query parameter.

The list of players on a Team can be found by traversing the Roster relationship between Team and Player. So, to find all the players on Boston's team, do GET /instances/Team::Boston/relationships/Roster. This returns a feed of all players "owned" by Boston, those on the active roster, injured players, minor-league players.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE /msa/instances/Team::Boston/relationships/Roster</title>
  <updated>2011-07-03T16:43:12-05:00</updated>
  <author><name>msa framework</name></author>
  <id>fffbb570-340e-4e6d-adaf-91c38c5e7fff</id>
  <link rel="self" href="https://tie.com/msa/instances/Team::Boston/relationships/Roster"/>
  <entry>
    <title type='text'>Player - David Krejci</title>
    <id>https://tie.com/msa/instances/Player::David%20Krejci</id>
    <updated>2011-07-03T16:45:53-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Player::David%20Krejci' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Player' />
    <content type='application/xml'>
      <inst:Player xmlns:atom='http://www.w3.org/2005/Atom'
                   xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>David Krejci</inst:Name>
        <inst:Position>C</inst:Position>
        <inst:Affiliation>Roster</inst:Affiliation>

        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Player/relationship/Stats'
                   href='https://tie.com/msa/instances/Player::David%20Krejci/relationships/Stats' />
      </inst:Player>
    </content>
  </entry>
   <entry>
    <title type='text'>Player - Nathan Horton</title>
    <id>https://tie.com/msa/instances/Player::Nathan%20Horton</id>
    <updated>2011-07-03T16:48:43-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Player::Nathan%20Horton' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Player' />
    <content type='application/xml'>
      <inst:Player xmlns:atom='http://www.w3.org/2005/Atom'
                   xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Name>Nathan Horton</inst:Name>
        <inst:Position>RW</inst:Position>
```

```
        <inst:Affiliation>Injured</inst:Affiliation>

        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Player/relationship/Stats'
                   href='https://tie.com/msa/instances/Player::David%20Krejci/relationships/Stats' />
      </inst:Player>
    </content>
  </entry>

  ... etc. for all the players affiliated with the Team "Boston"

</feed>
```

But what if I want just the active roster? The response snippet shown above shows Injured players and Prospects as well. Do GET /instances/Team::Boston/relationship/Roster?filter="Affiliation.eq.Roster", and given the consumer's understanding of the value set of the Player.Affiliation attribute (in this case it is an enum of values "Roster", "Injured", "Prospect"), then the feed contains only the active roster. If the consumer wanted just the injured players, it would do GET /instances/Team::Boston/relationship/Roster?filter="Affiliation.eq.Injured".

5. Report the current statistics for any given player

The statistics for a player is found by traversing the Stats relationship between Player and PlayerStats. Note, PlayerStats has two specializations, SkaterStats and GoalieStats. For any given Player, the type of PlayerStat returned is based on the value of Player.Position attribute. If the player is a Goalie, GoalieStats is related to the player, otherwise, the player has SkaterStats.

The basic way to do the traversal is to go from the Player representation, via the Stats relationship: GET /instances/Player::{Player.Name}}/relationships/Stats. To get David Krecji's stats, one would retrieve the Player record for David Krejci (for example off the team's roster) and then resolve the atom:link for the "Stats" relationship, ie GET /instances/Player::David%20Krejci/relationships/Stats

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE /msa/instances/Player::David%20Krejci/relationships/Stats</title>
  <updated>2011-07-03T16:52:13-05:00</updated>
  <author><name>msa framework</name></author>
  <id>000bb570-340e-4e6d-adaf-91c38c5e7000</id>
  <link rel="self" href="https://tie.com/msa/instances/Player::David%20Krejci/relationships/Stats"/>
  <entry>
    <title type='text'>SkaterStats for Player - David Krejci</title>
    <id>https://tie.com/msa/instances/SkaterStats::David%20Krejci</id>
    <updated>2011-07-03T16:45:53-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/SkaterStats::David%20Krejci' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/SkaterStats' />
    <content type='application/xml'>
      <inst:SkaterStats xmlns:atom='http://www.w3.org/2005/Atom'
                        xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Player>https://tie.com/msa/instances/Player::David%20Krejci</inst:Player>
        <inst:GamesPlayed>75</inst:GamesPlayed>
        <inst:Goals>13</inst:Goals>
        <inst:Assists>49</inst:Assists>
        <inst:Points>62</inst:Points>
        <inst:PenaltyMinutes>28</inst:PenaltyMinutes>
        <inst:AverageTimeOnIce>18:51</inst:AverageTimeOnIce>
        <inst:PlusMinus>23</inst:PlusMinus>
        <inst:PowerPlayGoals>1</inst:PowerPlayGoals>
        <inst:ShortHandedGoals>0</inst:ShortHandedGoals>
        <inst:GameWinningGoals>2</inst:GameWinningGoals>
        <inst:Shots>157</inst:Shots>
      </inst:SkaterStats>
    </content>
  </entry>
</feed>
```

Of course, the stats from ALL players could be retrieved by /types/PlayerStats/instances, which would retrieve all the GoalieStats and SkaterStats resources known to TIE. Perhaps a slightly more interesting approach would be to retrieve all the GoalieStats sorted by, say, GoalieStats.GoalsAgainstAverage, with GET /types/GoalieStats/instances?orderby=GoalsAgainstAverage. Another interesting approach would be to get a handle on the current "plus-minus" race amongst skaters by doing GET /types/SkaterStats/instances?orderby=PlusMinus. Who is in the race for the Art Ross trophy (scoring leader)? Do GET /types/PlayerStats/instances?orderby=Points. Note that this would retrieve both skaters and goaltenders (Goalies can score the odd assist on occasion and once in a decade or so even score goals!). So maybe a more fine tuned query to get the scoring race would be GET /types/SkaterStats/instances?orderby=Points.

Note, the above lists will be fairly long (there are perhaps 22 or so active Players on a roster times 30 teams, so that is over 600 entries), so the consumer should expect these responses to be paginated.

6. Update a player's record

A league official can update a player's record using PUT on /instances/Player::{Player.Name}/relationships/Stats. Note, the discussion above (use case 3. Update a team's record)

applies to updating a Player's statistics as well.

Sometimes, especially if it is early in the season, or perhaps if a Player has just be activated, there will not be a PlayerStats instance associated with the Player. In that case, the consumer would need to create one, using POST /instances/Player::{Player.Name}/relationships/Stats, passing in a partial representation (for Create) of the right PlayerStats sub type (eg if the Player is a Skater, SkaterStats, if a Goalie, GoalieStats). Here is a "first of the season" creation of stats for a Goalie (Cam Ward of the Carolina Hurricanes):

```
POST /instances/Player::Cam%20Ward/relationships/stats

<inst:GoalieStats xmlns:atom='http://www.w3.org/2005/Atom'
                  xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

  <inst:Player>https://tie.com/msa/instances/Player::Cam%20Ward</inst:Player>
  <inst:GamesPlayed>1</inst:GamesPlayed>
  <inst:Goals>0</inst:Goals>
  <inst:Assists>0</inst:Assists>
  <inst:Points>0</inst:Points>
  <inst:PenaltyMinutes>0</inst:PenaltyMinutes>
  <inst:AverageTimeOnIce>60:00</inst:AverageTimeOnIce>
  <inst:GoalsAgainst>2</inst:GoalsAgainst>
  <inst:GoalsAgainstAverage>2:00</inst:GoalsAgainstAverage>
  <inst:Shutouts>0</inst:Shutouts>
  <inst:EmptyNetGoalsAgainst>0</inst:EmptyNetGoalsAgainst>
  <inst:Wins>1</inst:Wins>
  <inst:Losses>0</inst:Losses>
  <inst:OvertimeLosses>0</inst:OvertimeLosses>
  <inst:MinutesOnIce>60</inst:MinutesOnIce>
</inst:GoalieStats>
```

What needs to go into a body of a request to create a new GoalieStats is shown by the partial representation for create of GoaliStats (eg GET /types/GoalieStats/PR_Create).

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
      xmlns:vsc="http://schemas.emc.com/vs-xml/namespace/Common/1.0">
  <updated>2011-07-03T17:10:33-5:00</updated>
  <title type="text">Partial Representation (Create) for to https://tie.com/msa/types/GoalieStats</title>
  <link href="https://tie.com/msa/types/TeamRecord/PR_Create" rel="self" />
  <author><name>msa framework</name></author>
  <id>111bb570-340e-4e6d-adaf-91c38c5e7111</id>
  <entry>
    <title type="text">GoalieStats Create Partial Representation</title>
    <id>https://tie.com/msa/types/GoalieStats/PR_Create</id>
    <updated>2011-07-03T17:10:33-5:00</updated>
    <link href="https://tie.com/msa/types/GoalieStats/PR_Create" rel="self" />
    <link href="https://tie.com/msa/types/GoalieStats" rel="related" />
    <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.tie.com/msa/tie/1.0">GoalieStats_PR_Create</type:typeName>
        <atom:link rel="self" href="https://tie.com/msa/types/GoalieStats/PR_Create"/>

        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:anyURI" minOccurs="1" maxOccurs="1"
                       description="The reference to the Player associated with these stats.  The identifier of instances of this type is the same identifier used for
the Player type">Player</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of games played by the player to date">GamesPlayed</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of goals scored by the Player">Goals</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of assists rewarded to the Player">Assists</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The total points earned by the Player">Points</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of penalty minutes assigned to the Player">PenaltyMinutes</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:float" minOccurs="1" maxOccurs="1"
                       description="The average amount of time in the game the player spent on the ice">AverageTimeOnIce</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of goals allowed">GoalsAgainst</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:float" minOccurs="1" maxOccurs="1"
                       description="The average number of goals allowed per game">GoalsAgainstAverage</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of saves made.">Saves</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:float" minOccurs="1" maxOccurs="1"
                       description="The number of saves made as a ratio of shots faced.">SavePercentage</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of games in which no goal was scored against the player.">Shutouts</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of goals scored against the player when he was NOT on the ice">EmptyNetGoalsAgainst</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of games in which the player was in the net when his team scored the game winning goal">Wins</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of games in which the player allowed the game winning goal to be scored during regulation time">Losses</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of games in which the player allowed the game winning goal to be scored during overtime">OvertimeLosses</vsc:attribute>
        <vsc:attribute type="http://www.w3.org/2001/XMLSchema-datatypes:integer" minOccurs="1" maxOccurs="1"
                       description="The number of minutes the player was on the ice">MinutesOnIce</vsc:attribute>
```

```
      </type:Type>
    </content>
  </entry>
</feed>
```

7. A consumer can retrieve a list of players for a team containing only those players that play a particular position

This request is another application of filtering applied to a GET request over a relationship. This request focuses on the relationship between a given Team and its Roster relationship to Player resources. Given the consumer has a URL to a Team instance (for example in a GET /types/Team/instances, retrieving all Team instances), the consumer can choose a team, for example Team::Boston, and traverse the Roster relationship to determine, for example, the list of Player instances for the position "Defense". So, to retrieve a list of defensemen on Boston's roster, the following request would be made: GET /instances/Team::Boston/relationships/Roster?filter="Position.eq.Defense".

The response is a feed of players similar to that shown for the solution to use case 4, above. Of course the feed for this particular request would contain only players on Boston that play the defense (have the value "D" in the Player.Position property).

8. Request a team's schedule of games, by month

To satisfy this case, we note the Schedule relationship between Team and Game. Team.Schedule relationship refers to a collection of Game instances, one for each Game the Team instance is scheduled to play in. A simple traversal of the relationship, GET /instances/Team::{Team.name}}/relationship/Schedule would retrieve a feed of Game instances that comprise that Team's schedule. Because this list is fairly large, eg 82 games per team in the regular season, the list is likely paginated. Here is the response to GET /instances/Team::Boston/relationship/Schedule

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE /msa/instances/Team::Boston/relationship/Schedule</title>
  <updated>2011-07-03T21:39:55-05:00</updated>
  <author><name>msa framework</name></author>
  <id>444bb570-340e-4e6d-adaf-91c38c5e7444</id>
  <link rel="self" href="https://tie.com/msa/instances/Team::Boston/relationships/Schedule"/>
  <link rel="next" href="https://tie.com/msa/instances/Team::Boston/relationships/Schedule?page=2" />
  <link rel="last" href="https://tie.com/msa/instances/Team::Boston/relationships/Schedule?page=5" />
  <entry>
    <title type='text'>Game - October 6, 2011 Philadelphia at Boston</title>
    <id>https://tie.com/msa/instances/Game::20111006BostonPhiladelphia</id>
    <updated>2011-07-03T21:39:55-05:00</updated>
    <link rel='self' href='https://tie.com/msa/instances/Game::20111006BostonPhiladelphia' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/Game' />
    <content type='application/xml'>
      <inst:Game xmlns:atom='http://www.w3.org/2005/Atom'
                 xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:ID>20111006BostonPhiladelphia</inst:ID>
        <inst:Date>2011-10-06</inst:Date>
        <inst:HomeTeam>https://tie.com/msa/instances/Team::Boston</inst:HomeTeam>
        <inst:VisitorTeam>https://tie.com/msa/instances/Team::Philadelphia</inst:VisitorTeam>
        <inst:StartTime>19:00:00-05:00</inst:StartTime>
        <inst:EndTime></inst:EndTime>
        <inst:HomeScore></inst:HomeScore>
        <inst:VisitorScore></inst:VisitorScore>

        <atom:link rel='http://schemas.tie.com/msa/tie/1.0/Player/relationship/Summary'
                   href='https://tie.com/msa/instances/Game::20111006BostonPhiladelphia/relationships/Summary' />
      </inst:Game>
    </content>
  </entry>
  <entry>
    <title type='text'>Game - October 8, 2011 Tampa Bay at Boston</title>
    <id>https://tie.com/msa/instances/Game::20111008BostonTampa%20Bay</id>
...
  </entry>

  ... etc. for the first 20 Games the Team "Boston" is involved with.  The next "page" (see atom:link rel="next") contains games 21 thru 40)

</feed>
```

We would use filtering to pare the list down to Game instances scheduled for a given month. The filtered request, GET /instances/Team::{Team.name}}/relationship/Schedule?filter="Date.ge.20111101 AND Date.le.20111130", will return only those Games scheduled between Nov 1, 2011 and Nov 30, 2011.

9. Request a summary of a given game

The summary information for a Game is contained in the relationship named Summary between Game and GameSummary. The Game.Summary relationship refers to a collection of zero or more GameSummary instances (specifically instances of any of GameSummary's subtypes). A request such as GET /instances/Game::{Game.ID}/relationships/Summary would retrieve this collection. For example, a Summary of the Game between Ottawa and Boston on November 1, 2011, would be retrieved by GET /instances/Game::20111101OttawaBoston/relationships/Summary.

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns="http://www.w3.org/2005/Atom" >
  <title>TIE Game Summary for Game on November 1, 2011 Boston at Ottawa</title>
  <updated>2011-07-03T21:52:13-05:00</updated>
  <author><name>msa framework</name></author>
  <id>555bb570-340e-4e6d-adaf-91c38c5e7555</id>
  <link rel="self" href="https://tie.com/msa/instances/Game::20111101OttawaBoston/relationships/Summary"/>
  <entry>
    <title type='text'>Bergeron from Marchand and Seidenberg 14:37 of Period 1</title>
    <id>20111006OttawaBoston::1</id>
    <updated>2011-07-03T21:52:13-05:00</updated>
    <link rel='self' href='https://tie.com/msahttps://tie.com/msa/instances/GameSummary::20111101OttawaBoston::1' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/ScoringSummary' />
    <content type='application/xml'>
      <inst:ScoringSummary xmlns:atom='http://www.w3.org/2005/Atom'
                           xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Game>https://tie.com/msahttps://tie.com/msa/instances/Game::20111101OttawaBoston</inst:Game>
        <inst:Period>1</inst:Period>
        <inst:Time>14:37</inst:Time>
        <inst:Team>https://tie.com/msa/instances/Team::Boston</inst:Team>
        <inst:Goal>https://tie.com/msa/instances/Player::Patrice%20Bergeron</inst:Goal>
        <inst:Assist>https://tie.com/msa/instances/Player::Brad%20Marchand</inst:Assist>
        <inst:Assist>https://tie.com/msa/instances/Player::Dennis%20Seidenberg</inst:Assist>

      </inst:ScoringSummary>
    </content>
  </entry>
  <entry>
    <title type='text'>Chris Neil 2 minutes for Tripping 18:21 of Period 1</title>
    <id>20111006OttawaBoston::2</id>
    <updated>2011-07-03T22:00:02-05:00</updated>
    <link rel='self' href='https://tie.com/msahttps://tie.com/msa/instances/GameSummary::20111101OttawaBoston::2' />
    <link rel='http://schemas.emc.com/msa/reln/type' href='https://tie.com/msa/types/PenaltySummary' />
    <content type='application/xml'>
      <inst:PenaltySummary xmlns:atom='http://www.w3.org/2005/Atom'
                           xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>

        <inst:Game>https://tie.com/msahttps://tie.com/msa/instances/Game::20111101OttawaBoston</inst:Game>
        <inst:Period>1</inst:Period>
        <inst:Time>18:21</inst:Time>
        <inst:Team>https://tie.com/msa/instances/Team::Ottawa</inst:Team>
        <inst:Player>https://tie.com/msa/instances/Player::Chris%20Neil</inst:Player>
        <inst:PenaltyTime>2</inst:PenaltyTime>
        <inst:Infraction>Tripping</inst:Infraction>

      </inst:PenaltySummary >
    </content>
  </entry>

  ... etc. for the other ScoringSummary items and PenaltySummary items in the game.

</feed>
```

10. Update a game summary by adding a new GameSummary resource to a given Game resource's summary

League officials can add a new item to a game's summary by doing a POST operation on the Game's Summary relationship. For example, to add a ScoringSummary, recording the fact that Boston Player Milan Lucic scored a power play goal at 3:57 in the second period of the November 1, 2011 game between Ottawa and Boston, assisted by David Krecji, the league official would do a POST Game::20111101OttawaBoston/relationships/Summary, passing in a partial representation (for create) of a ScoringSummary instance:

```
POST /instances/Game::20111101OttawaBoston/relationships/Summary

<inst:ScoringSummary xmlns:atom='http://www.w3.org/2005/Atom'
                     xmlns:inst='http://schemas.tie.com/msa/tie/1.0'>
  <inst:Game>https://tie.com/msahttps://tie.com/msa/instances/Game::20111101OttawaBoston</inst:Game>
  <inst:Period>2</inst:Period>
  <inst:Time>3:57</inst:Time>
  <inst:Team>https://tie.com/msa/instances/Team::Boston</inst:Team>
  <inst:Goal>https://tie.com/msa/instances/Player::Milan%20Lucic</inst:Goal>
  <inst:Assist>https://tie.com/msa/instances/Player::David%20Krecji</inst:Assist>
  <inst:PowerPlay>true</inst:PowerPlay>
</inst:ScoringSummary>
```

Note the ID property of the ScoringSummary is not included, it is computed by the EDAA. The partial representation (for create) of a ScoringSummary instance can be examined by doing GET /types/ScoringSummary/PR_Create.

Implement using shared components

TIE would either need to acquire an EDAA compatible framework, or build one from scratch, using for example Spring MVC.

TIE will use the Central Authentication Service from jsig.org as the basis for authenticating consumers to its EDAA.

Next Chapter

Detailed Primer on EDAA is the next chapter.

EDAA Primer - Details

This topic is an overview/explanation of some of the features of EDAA. It is part of the the EDAA Primer.

Contents

1 Chapters
2 Quick tour of EDAA
2.1 On Resource types and VS-XML
2.2 On Resource Representations
2.3 Query parms and why they are useful 2.4 On relationships and links
2.5 Using JSON as an alternative to atom/xml 2.6 Making changes to resources

Chapters

1. Introduction gives an overview of the EDAA specification, motivation, etc
2. Example gives an example of how to design a REST API using EDAA
3. **Detailed Primer on EDAA** details of the various EDAA features **<--- you are here**

Quick tour of EDAA

The following is a list of the various features of EDAA from the point of view of examples (as opposed to austere, normative behavior description that is found in the specs). Each item is a topic on its own.

It is important that EDAA developers and consumers understand the notion of how IT Infrastructure resource information is presented in the EDAA. That material together with an understanding of how resource type information is available through the API is a good grounding to understand what an EDAA is all about. These happen to be the first two topics in this section of the primer. The other topics listed below provide insight into various details and features that make the EDAA more consumable.

On Resource types and VS-XML

This topic is a brief discussion of how resource "types" (or classes) are represented in EDAA using a notation called "VS-XML". URI patterns that begin with /types/... typically contain responses comprised of VS-XML elements.

On Resource Representations

This topic reviews how resources are represented in EDAA, discussing concepts associated with URI patterns beginning with /instances/...

This topic is a discussion of how EDAA uses query parameters to allow consumer specified pagination, response formatting, resource partial representations, sorting and filtering.

On relationships and links

This topic describes the two styles of representing relationships and discusses the tradeoffs.

Using JSON as an alternative to atom/xml

This topic describes the two serialization formats described in EDAA: atom/xml and JSON. EDAA implementations can support either or both formats and EDAA consumers can choose which serialization mechanism they want.

Making changes to resources

This topic reviews a set of "patterns" that developers should consider when they attempt to enhance the EDAA capabilities of their product to support operations that create/update and delete resources or otherwise allow consumers to directly modify data within their product through an EDAA interface.

EDAA Primer - Details - Resource Types

This topic is an overview of how information about resource types is handled in EDAA.

Contents

1 Resource Types
2 Why is Resource Type information useful?
3 Using REST to access Type information
  3.1 /types
  3.2 /types/{typeName}
  3.3 /types/{typeName}/hierarchy
  3.4 /types/{typeName}/PR_Create

Resource Types

In applications as diverse as IT Infrastructure Management, there are often many different "kinds" or "types" of resource. For example in DELL EMC Smarts Service Assurance Management (SAM), there are 702 different types of resource exposed through its REST interface.

For any product, whether there are dozens or hundreds, having a consistently represented resource model is a benefit to developers trying to interpret and use the EDAA to that product.

In the REST community, there is no well agreed standard for representing resource type information. This is where the types information defined by EDAA comes in. In EDAA, we define a standard resource model to represent resource types, called VS-XML, and provide a small number of URI patterns to provide read-only (GET) access to information about which types are supported by a given EDAA implementation.

VS-XML is described in detail as part of the base EDAA specification.

Why is Resource Type information useful?

- consistent documentation of the resource model, available for developers to view, augmenting the basic human readable developer documentation
- can contain hints to UI developers to choose UI widgets/forms to display/edit instances of the type
- code generation, for example, client library generation
- Format transformation
  - A project is being prototyped that takes in a /types feed, generates SpringDataGraph annotated POJOs and then consumes /instances feed to "dump" data from an EDAA into a neo4j graph-database to support more efficient topology (graph) style queries.

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 "VS-XML", 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/vs-
xml/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>
</feed>
```

EDAA Primer - Details - Resource Representations

The major task of an EDAA is to provide read access to resource information, including collections or topologies of related resources.

This topic is an overview of how resources (often called instances) are represented in EDAA.

Contents

1 Representing Resource Instances
- 1.1 atom:link use
- 1.2 eTag
- 1.3 Atom:entry metadata
- 1.4 Resource Identifier
- 1.5 Resource type information
- 1.6 Actions
- 1.7 Attributes
- 1.8 Relationships

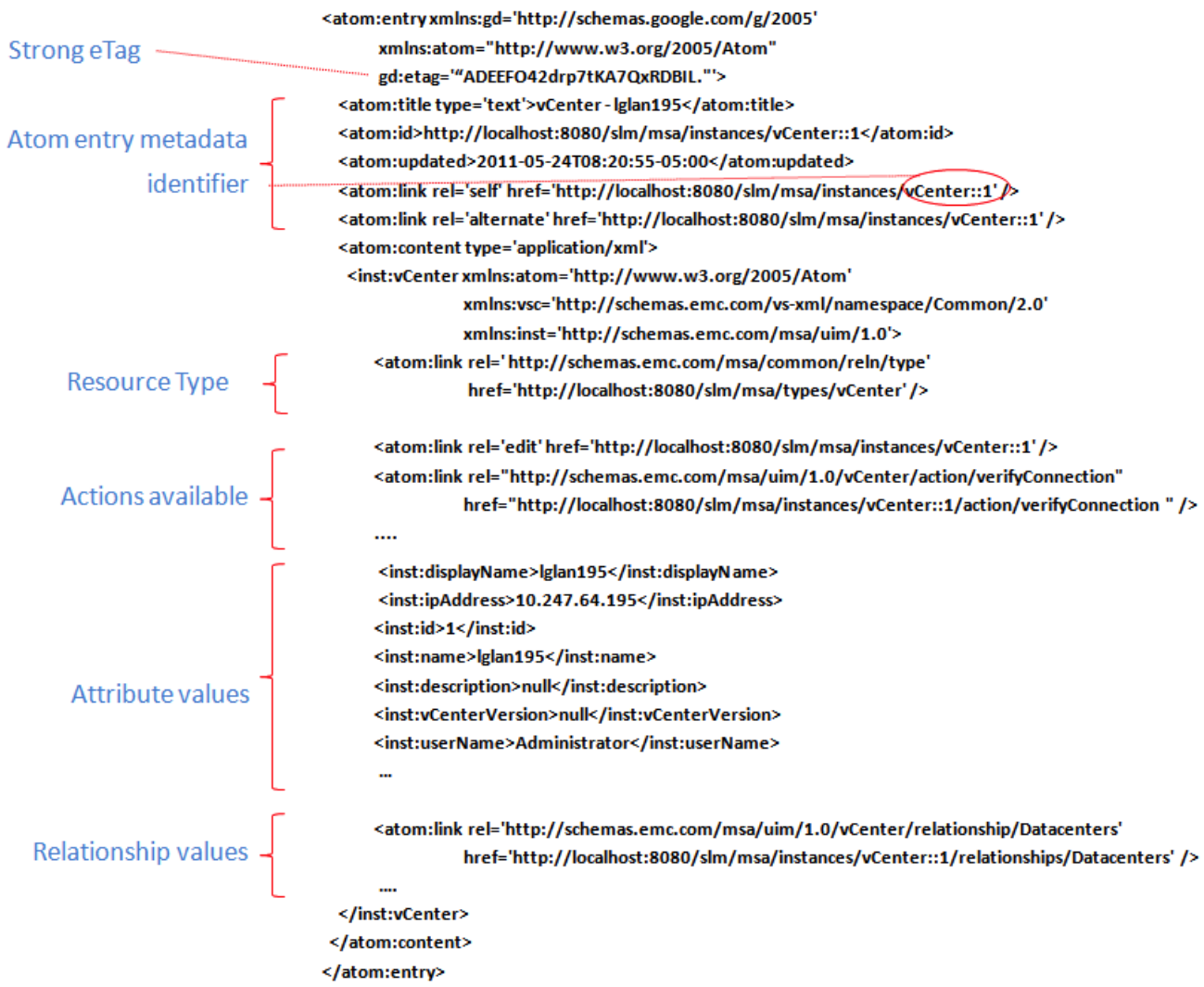
2 GET Operations Retrieving Individual Resources
- 2.1 /instances/{id}
- 2.2 /instances/{id}/relationships

3 Feeds of Resources
- 3.1 Feed level eTag
- 3.2 Atom Feed Required Metadata
- 3.3 Pagination Links

4 GET Operations Retrieving Feeds of Resources
- 4.1 /types/{typeName}/instances
- 4.2 /instances
- 4.3 /instances/{id}/relationships/{relName}

Representing Resource Instances

The following figure shows an example "instance" representation contained within an atom:entry.



The resource is represented within an atom:entry through a combination of entry-level metadata and specific markup appearing under the /entry/content element. The bulk of the "interesting" markup is within the content element.

Note also that the representation shown above is atom/XML. The actual serialization of the representation can be of different media types. EDAA also defines a standard approach to serializing resources using JSON.

atom:link use

It should be noted that EDAA uses the atom:link element for many different approaches. The link construct itself is a very generic construct for describing relationships between resources in resource representations.

Atom:link elements that appear as child elements of the atom:entry describe relationships between EDAA resources that are related to constructing

legal atom:feed syntax. Links such as @rel="self", @rel="alternate" typically appear at this level.
Atom:links that are more closely associated with the EDAA representation of resources, things like the relationship to the resource's type, indication of eventing possibilities or action possibilities, and of course relationships to other EDAA resources themselves, are contained under the content element.

eTag

Following the recommendations in the EDAA spec, the atom:feed elements and atom:entry elements should be associated with an eTag. The ETag approach is a feature of http that provides a convenient mechanism of determining versions of web resources, and a convention in http to avoid retrieving copies of a resource's representation if there has been no change since the last retrieval of that resource.

Although eTag data usually appears within http headers, Google's gData pioneered an approach to decorate atom:feed and atom:entry elements with eTag information.

An eTag appearing in an atom:entry is useful for forming cache-friendly retrievals on the resource contained within the atom:entry and, in the case of modification operations on the resource, determine that the version of the resource being operated on is the most current.

Atom:entry metadata

Per Atom Syndication Format the atom:entry must contain an id tag, a title tag and an updated tag. We note that the atom:feed in EDAA contains the author tag.

Resource Identifier

Arguably the most important property of a resource is its identifier. An identifier is a property that distinguishes one resource from all the other resources in any EDAA implementation. It is critical in the design of a resource model that the designer articulate the identifier scheme used for each resource in the model. Most resource models have a single identifier scheme and syntax. Some models use a hybrid approach, using multiple different schemes for the resources in the model.

Identifiers are usually properties associated with the resource. Sometimes a single property can be used to distinguish an individual resource instances from all other instances. Sometimes it takes a combination of multiple properties to uniquely identify an instance. Frequently, a naturally occurring property, like "ip address" or "fully qualified name", can be used as an identifier. Sometimes, an artificial "id" property is created, containing a string, or integer or even UUID serves as the identifier. The examples in the IMG Interop Congress wiki demonstrate each of these possible approaches to identifier scheme. EDAA does NOT dictate or limit the approach a designer takes to choose the identifier scheme(s) used in a resource model.

Syntactically, identifier schemes play an important role in the URI patterns described by EDAA. URI patterns such as /instances/{id}... define parameters where the identifier should appear in the URI to refine which resource is being targeted by an operation initiated by a consumer.

There are several different identifier syntax approaches seen in practice:

type encoded with identifiers

- commonly found in many EDAA (especially older) implementations, identifiers like VLAN::VLAN-106, where the identifier is formed by encoding the name of the resource type, followed by two ':' characters, followed by a ':' separated list of property values that, in aggregate uniquely identify the instance within the scope of the resource type, and by including the resource type therefore uniquely identify the resource amongst all resources in the EDAA.
- to retrieve a representation of the resource identified by "VLAN::VLAN-106" a GET operation on https://.../instances/VLAN::VLAN-106 can be sent by the consumer. The "VLAN::VLAN-106" is the substitution for {id} within the /instances/{id} URI pattern.

type-based node sets

- another approach, that matches the URI style used within the broader web community, uses a '/' separated approach to forming the {id} within the URI. For example, the identifier could be "VLAN/VLAN-106" to act as the identifier for a specific resource of type "VLAN".
-

to retrieve a representation of the resource identified by "VLAN::VLAN-106" a GET operation on https://.../instances/VLAN/VLAN-106 can be sent by the consumer. The "VLAN/VLAN-106" is the substitution for {id} within the /instances/{id} URI pattern.

- a concern with this approach is that it may make resolving URI patterns a bit trickier on the server side. In particular to resolve a pattern like /instances/VLAN/VLAN-106/relationships/Aggregates can be tricky if the number of nodes in the {id} portion can vary. Furthermore, it restricts somewhat, the identifier scheme, as the identifier scheme cannot include certain literals like "relationships" that conflict with fixed literals in the EDAA URI patterns.

odata approach

- Microsoft's odata uses '(' and ')' to syntactically isolate the identifier within a URI. Following the example of the previous two URIs, an identifier like "(VLAN-106)" could appear, and if that string was unique amongst all identifiers within the EDAA, could server as an {id} within a
- URI.
  to retrieve a representation of the resource identified by "VLAN-106" a GET operation on https://.../instances/(VLAN-106) can be sent by the
- consumer. The "(VLAN-106)" is the substitution for {id} within the /instances/{id} URI pattern.

although syntactically legal URIs, no other web property has popularized this approach to identifiers other than Microsoft's odata.

It should be noted that for the most part, the URI syntax is not terribly important for the consumer. For the most part, following principles of HATEOS , the consumer traverses the interconnected set of resources by following atom:link elements, and not by manufacturing URIs.

Resource type information

It is always clear what is the type of a given resource by examining the atom:entry level metadata for an atom link with value of @rel = "{common}/reln/type". In the example given above, there is just such an atom:link:

```
    <atom:link rel='http://schemas.emc.com/msa/common/reln/type'
               href='http://localhost:8080/slm/msa/types/vCenter' />
```

That indicates the resource is of the type "/types/vCenter" and the value of @href indicates an address at which a representation of that type can be retrieved.

Any resource representation in EDAA MUST include an atom:link to the resource's type.

Actions

Per EDAA spec and EDAA Read/Write a resource representation may include a collection of atom:links that suggest additional operations that may be performed on the resource.

These atom:links follow one of two patterns:

1. an Atom publication protocol "edit" link
   - eg. <atom:link rel="edit" href="https:// ..." />
   - Per the atom publication protocol, the "edit" link indicates that the consumer may use PUT/POST/PATCH/DELETE against the URL contained in @href to perform simple REST style manipulations of the resource.
2. an "action" style href
   - e.g. <atom:link rel="http://schemas.emc.com/msa/uim/1.0/vCenter/action/verifyConnection" href="http://localhost:8080/slm/msa/instances/vCenter::1/action/verifyConnection" />
   - This style of action link includes a value in @rel that follows the pattern .../{typeName}/action/{actionName}. An operation (usually POST) can be performed on the URL contained in @href to achieve some specific action on the resource. Often, the URI in @rel itself can be resolved to retrieve further semantics on the operation itself to aid developers.

Although the representation of a resource's type (see type-level action declaration) may suggest an action is possible on instances of a type, the atom:links corresponding to that action may not appear in any given instance's representation. An action-related atom:link may not appear in a

resource representation if the current state of the resource does not permit the operation to be performed. The link may not appear if the authenticated user doesn't have permissions to invoke the operation associated with the link. The server ultimately determines if it is appropriate or not to include an action-related atom:link in the resource's representation.

Attributes

A resource's attributes are represented in a very straight forward fashion, in a child element of the content element. The set of attributes that may appear in a resource representation follows the pattern defined by that resource's type .

Relationships

In most EDAA situations, resources are related to other resources in some complex topology graph. Related resources are represented as "relationships" in EDAA. At the type level, the resource's type defines a set of relationship declarations describing the kinds of relationships instances of the type are typically involved in.

Within the representation of any resource instance of that type, an atom:link will appear, one for each relationship defined in the type. Each atom:link will contain a value of @rel that corresponds to an identifier of a relationship and the value of the @href will contain a URL to a resource that represents the collection of resource instances related via that relationship.

An example relationship atom:link appears in the example above:

```
            <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' />
```

As defined in EDAA expand query parameter, EDAA introduced an additional, alternative form of representing relationships that may be included in some resource representations, that augments the atom:link of the relationship with some cached information about the related resources that are contained in the relationship and may even contain partial representations of each of those resources.

GET Operations Retrieving Individual Resources

The following URI patterns can be used to retrieve representations of individual instances.

In the case where the resource is not found (eg the value of {id} does not correspond to any resource known to the EDAA and visible to the user), the response to a GET operation is an http response code 404 (not found).

In the case where the resource is found, the response to a GET operation is an atom:feed containing exactly one child atom:entry. The atom:entry contains a representation of the identified resource in the style described above.

The atom:feed also contains feed-level metadata.

/instances/{id}

This URI pattern is used to uniquely identify an EDAA resource instance. A GET operation on this URI will respond with an http 404 (not found) if the resource is not available to the user or it will respond with an http 200 (ok) and an atom:feed containing exactly one child atom:entry, which itself contains a representation of the identified resource.

An example of a response to GET /instances/

```
<?xml version="1.0" encoding="UTF-8"?><atom:feed xmlns:atom="http://www.w3.org/2005/Atom" xmlns:vsc="http://schemas.emc.com/vs-
xml/namespace/Common/1.0">
<atom:title>/slm/msa/uim/instances/NAS::10021</atom:title>
<atom:updated>2012-05-17T09:58:49-04:00</atom:updated>
<atom:author>
<name>msa framework</name>
```

```
</atom:author>
<atom:id>9b85214d-c0de-45ef-81b5-ec3f49fcde97</atom:id>
<atom:link rel="self" href="https:.../slm/msa/uim/instances/NAS::10021"/>

<atom:entry>
  <atom:title type="text">NAS - GladNAS236</atom:title>
  <atom:id>https:.../slm/msa/uim/instances/NAS::10021</atom:id>
  <atom:updated>2012-05-17T09:58:49-04:00</atom:updated>
  <atom:link rel="self" href="https:.../slm/msa/uim/instances/NAS::10021"/>
  <atom:link rel="alternate" href="https:.../msa/uim/instances/NAS::10021"/>
  <atom:link rel="http://schemas.emc.com/msa/common/reln/type" href="https:.../slm/msa/uim/types/NAS"/>
  <atom:content type="application/xml">
    <vsc:NAS xmlns:inst="http://schemas.emc.com/msa/uim/1.0">
      <vsc:displayName>GladNAS236</vsc:displayName>
      <inst:Model>Celerra VNX5700</inst:Model>

      ... etc. for the rest of the attributes

      <atom:link rel="self" href="https:.../slm/msa/uim/instances/NAS::10021"/>
      <atom:link rel="http://schemas.emc.com/msa/common/reln/type" href="https:.../slm/msa/uim/types/NAS"/>
      <atom:link rel="http://schemas.emc.com/msa/uim/1.0/NAS/relationship/PortGroup"
href="https:.../slm/msa/uim/instances/NAS::10021/relationships/PortGroup"/>

      ... etc. for the rest of the relationships

    </vsc:NAS>
</atom:content>
</atom:entry>
</atom:feed>
```

/instances/{id}/relationships

Because the default representation of relationships in EDAA is to use a single atom:link element to refer to the resource containing the collection of related resources, the response to /instances/{id}/relationships is essentially the same as /instances/{id}.

The /instances/{id}/relationships URI pattern was introduced in EDAA for sake of completeness of the URI patterns.

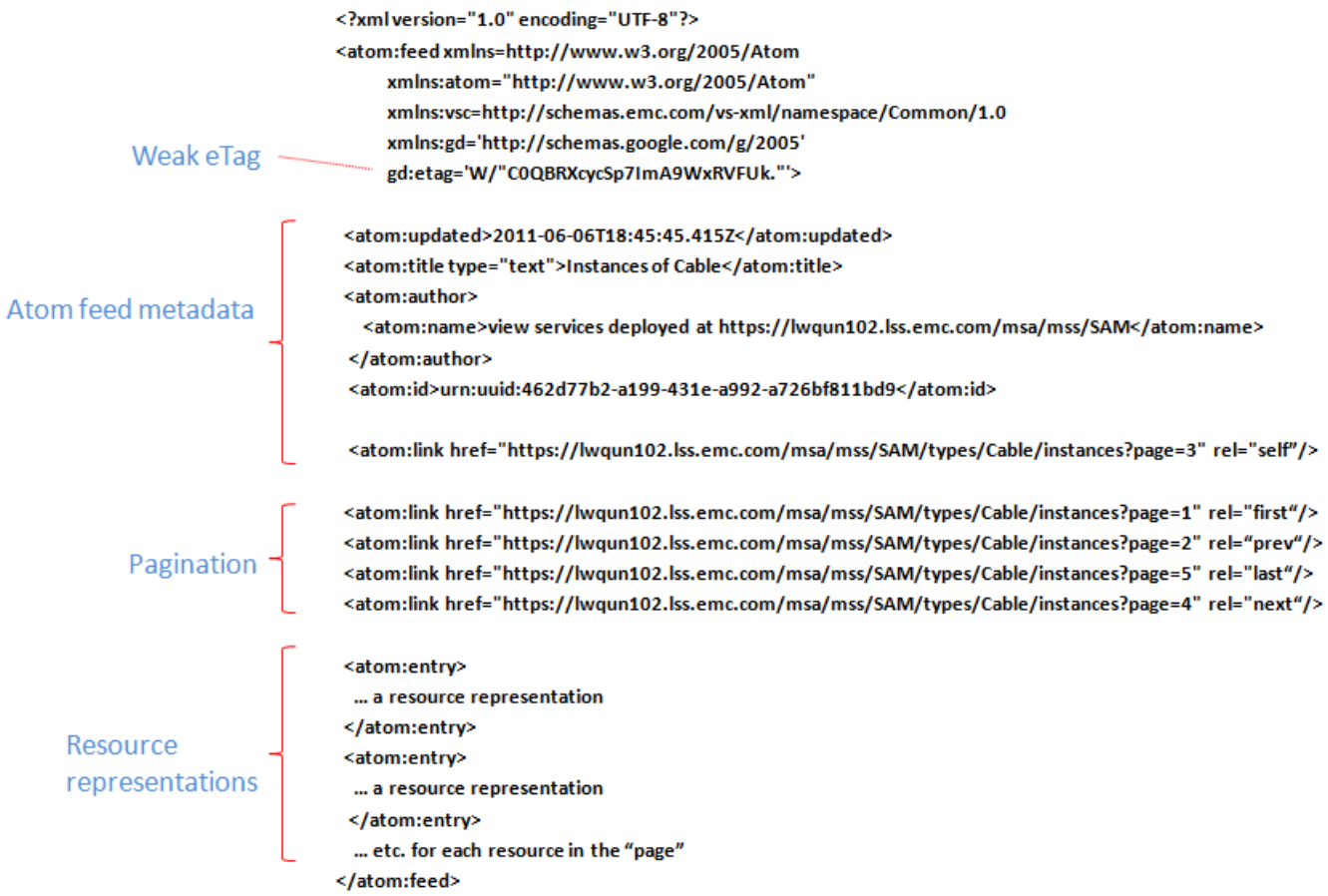
Feeds of Resources

There are several URI patterns where it is expected that possibly many resource representations will be returned in response to a GET operation.

We use RFC 5005 Atom Pagination to provide a "chunking" or "paging" mechanism so that "reasonable" sized responses can be generated and returned to the consumer, and the consumer can use multiple GET requests to retrieve various related "pages" or "chunks" of the feed.

In the absence of a pagination or chunking approach, any REST API, including EDAA, would be periodically unusable, if the client issued a simple GET operation that might involve thousands of resources.

The following figure shows atom-feed specific information commonly seen within EDAA:




```
<?xml version="1.0" encoding="UTF-8"?>
<atom:feed xmlns=http://www.w3.org/2005/Atom
      xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns:vsc=http://schemas.emc.com/vs-xml/namespace/Common/1.0
      xmlns:gd='http://schemas.google.com/g/2005'
      gd:etag='W/"C0QBRXcycSp7ImA9WxRVFUk."'>

 <atom:updated>2011-06-06T18:45:45.415Z</atom:updated>
 <atom:title type="text">Instances of Cable</atom:title>
 <atom:author>
   <atom:name>view services deployed at https://lwqun102.lss.emc.com/msa/mss/SAM</atom:name>
 </atom:author>
 <atom:id>urn:uuid:462d77b2-a199-431e-a992-a726bf811bd9</atom:id>

 <atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=3" rel="self"/>

 <atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=1" rel="first"/>
 <atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=2" rel="prev"/>
 <atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=5" rel="last"/>
 <atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=4" rel="next"/>

 <atom:entry>
  ... a resource representation
 </atom:entry>
 <atom:entry>
  ... a resource representation
 </atom:entry>
  ... etc. for each resource in the "page"
</atom:feed>
```

Feed level eTag

Following the recommendations in the EDAA spec, the atom:feed should be associated with an eTag. The ETag approach is a feature of http that provides a convenient mechanism of determining versions of web resources, and a convention in http to avoid retrieving copies of a resource's representation if there has been no change since the last retrieval of that resource.

Although eTag data usually appears within http headers, Google's gData pioneered an approach to decorate atom:feed and atom:entry elements with eTag information.

An eTag appearing in an atom:feed is a "weak eTag", meaning it should be used only for "conditional retrieval".

Atom Feed Required Metadata

Per Atom Syndication Format the atom:feed must contain an id tag, a title tag and an updated tag. We note that the atom:feed in EDAA contains the author tag, thereby removing the obligation that each atom:entry contain an author tag. We also follow the recommendation in RFC 4287 to include an atom:link with @rel="self" in all EDAA atom:feeds.

Pagination Links

Following the RFC 5005 Atom Pagination most atom:feeds in EDAA, and certainly all atom:feeds that deal with multiple resources, include 1 or more of the atom:link elements containing @rel="first" or "next" or "prev" or "last". The semantics of these atom:link elements are described in RFC 5005 .

In EDAA we define two query parameters EDAA spec called ?page and ?per_page. These query parameters control the pagination related aspects of an atom:feed.

The ?per_page query parameter allows the consumer to control the maximum number of resources to appear in any given "page". If this query parameter is not specified, the server determines a "default" value, usually around 20 resources per page.

The ?page allows the consumer to specify, given a particular combination of ?per_page and optionally the ?orderby and ?filter query parameters, a particular page to retrieve. Typically ?page is used within the RFC 5005 style atom:link elements as seen in the example above:

```
<atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=1" rel="first"/>
<atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=2" rel="prev"/>
<atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=4" rel="next"/>
<atom:link href="https://lwqun102.lss.emc.com/msa/mss/SAM/types/Cable/instances?page=5" rel="last"/>
```

The semantics of RFC 5005 and the ?page query parameter are pretty straight forward. This combination allows a consistent, HATEOAS style interaction for the consumer to iterate over a collection of resources.

GET Operations Retrieving Feeds of Resources

The following URI patterns are expected to return an atom:feed containing a collection of multiple resource representations. As with any atom:feed in EDAA containing a multiplicity of representations, it is paginated according to RFC 5005 Atom Pagination .

See here for more explanation on the content of the atom:feed metadata and here for details on how individual resources are represented within an atom:entry.

/types/{typeName}/instances

The purpose of this URI pattern is to retrieve a collection of resources, one for each instance of the resource type identified by {typeName}. Note, that if the type identified by {typeName} as "sub types", then this URI pattern will return instances of those sub types as well.

If {typeName} does not correspond to a type known to the EDAA (eg no such named type appears in the response to a GET operation on /types), then a GET operation will respond with an http error code 404 (not found).

If the {typeName} does correspond to a type known to the EDAA, then the response is a paginated atom:feed containing an atom:entry for each resource instance of that type.

This URI pattern is a very commonly used pattern.

This example shows /types/vCenter/instances:

```
<?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/instances"/>
  <entry>
    <title type='text'>vCenter - lglan195</title>
    <id>http://localhost:8080/slm/msa/instances/vCenter::1</id>
    <updated>2011-05-24T08:20:55-05:00</updated>
```

```
  <link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::1' />
  <link rel='alternate' href='http://localhost:8080/slm/msa/instances/vCenter::1' />
  <link rel='edit' href='http://localhost:8080/slm/msa/instances/vCenter::1' />
  <link rel='related' href='http://localhost:8080/slm/msa/types/vCenter' />
  <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'>
      <vsc:displayName>lglan195</vsc:displayName>
      <inst:ipAddress>10.247.64.195</inst:ipAddress>
      <inst:id>1</inst:id>
      <inst:name>lglan195</inst:name>
      <inst:description>null</inst:description>
      <inst:vCenterVersion>null</inst:vCenterVersion>
      <inst:userName>Administrator</inst:userName>
      <inst:port>443</inst:port>
      <inst:password>[B@1827245</inst:password>
      <inst:connectionStatus>REACHABLE</inst:connectionStatus>
      <atom:link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::1' />
      <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' />
      <atom:link rel='http://schemas.emc.com/msa/uim/1.0/vCenter/action/verifyConnection'
                 href='http://localhost:8080/slm/msa/instances/vCenter::1/action/verifyConnection' />
    </inst:vCenter>
  </content>
</entry>
<entry>
  <title type='text'>vCenter - xxxbarfxxxxxxx</title>
  <id>http://localhost:8080/slm/msa/instances/vCenter::8</id>
  <updated>2011-05-24T08:20:55-05:00</updated>
  <link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::8' />
  <link rel='alternate' href='http://localhost:8080/slm/msa/instances/vCenter::8' />
  <link rel='edit' href='http://localhost:8080/slm/msa/instances/vCenter::8' />
  <link rel='related' href='http://localhost:8080/slm/msa/types/vCenter' />
  <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'>
      <vsc:displayName>xxxbarfxxxxxxx</vsc:displayName>
      <inst:ipAddress>127.0.0.1</inst:ipAddress>
      <inst:id>8</inst:id>
      <inst:name>xxxbarfxxxxxxx</inst:name>
      <inst:description>something</inst:description>
      <inst:vCenterVersion>null</inst:vCenterVersion>
      <inst:userName>me</inst:userName>
      <inst:port>80</inst:port>
      <inst:password>[B@1708099</inst:password>
      <inst:connectionStatus>UN_VERIFIED</inst:connectionStatus>
      <atom:link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::8' />
      <atom:link rel='http://schemas.emc.com/msa/uim/1.0/vCenter/relationship/Datacenters'
                 href='http://localhost:8080/slm/msa/instances/vCenter::8/relationships/Datacenters' />
      <atom:link rel='http://schemas.emc.com/msa/uim/1.0/vCenter/action/verifyConnection'
                 href='http://localhost:8080/slm/msa/instances/vCenter::8/action/verifyConnection' />
    </inst:vCenter>
  </content>
</entry>
<entry>
  <title type='text'>vCenter - matt 111</title>
  <id>http://localhost:8080/slm/msa/instances/vCenter::9</id>
  <updated>2011-05-24T08:20:55-05:00</updated>
  <link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::9' />
  <link rel='alternate' href='http://localhost:8080/slm/msa/instances/vCenter::9' />
  <link rel='edit' href='http://localhost:8080/slm/msa/instances/vCenter::9' />
  <link rel='related' href='http://localhost:8080/slm/msa/types/vCenter' />
  <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'>
      <vsc:displayName>matt 111</vsc:displayName>
      <inst:ipAddress>127.0.0.1</inst:ipAddress>
      <inst:id>9</inst:id>
      <inst:name>matt 111</inst:name>
      <inst:description>something</inst:description>
      <inst:vCenterVersion>null</inst:vCenterVersion>
      <inst:userName>me</inst:userName>
      <inst:port>80</inst:port>
      <inst:password>[B@1cb79b7</inst:password>
      <inst:connectionStatus>UN_VERIFIED</inst:connectionStatus>
      <atom:link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::9' />
      <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' />
```

```
        <atom:link rel='http://schemas.emc.com/msa/uim/1.0/vCenter/action/verifyConnection'
                   href='http://localhost:8080/slm/msa/instances/vCenter::9/action/verifyConnection' />
      </inst:vCenter>
    </content>
  </entry>
</feed>
```

/instances

This URI pattern returns all instances known to an EDAA. This collection may be very large, hence the utility of RFC 5005 pagination.

This URI pattern may be used, for example, to bulk upload the content of an EDAA, for example to load it into an alternative data store for purposes of, say analytics, or topology graph traversal.

/instances/{id}/relationships/{relName}

This URI pattern is used to retrieve representations of each resource related to the identified resource over the relationship identified by {relName}.

A GET operation on this URI will respond with an http 404 (not found) if the resource identified by {id} is not available to the user.

A GET operation on this URI will respond with an http 404 (not found) if the value of {relName} does not correspond to a relationship modeled for the resource type associated with the resource identified by {id}.

If there are no errors, the response to a GET operation on this URI is a paginated atom:feed, where each entry in the atom:feed contains a representation of a resourced related to the resource identified by {id} over the relationship identified by {relName}.

EDAA Primer - Details - Query Parameters

Many REST API approaches, such as Google's gdata or Microsoft's odata, leverage http query parameters (eg the things that start with ? or & at the end of a URL). This topic is a discussion of how EDAA uses a small set of query parameters to provide richer customization of the EDAA for consumers to tailor.

Contents

1 Query Parameters in EDAA
2 page and per_page
  2.1 Interpretation by URI Pattern
3 alt
  3.1 Interpretation by URI Pattern
4 fields
  4.1 Interpretation by URI Pattern
5 expand
  5.1 Interpretation by URI Pattern
6 orderby
  6.1 Interpretation by URI Pattern
7 filter
  7.1 Filter Expressions
  7.2 filter expression terms
  7.3 filter expression operator precedence
  7.4 Missing Properties and Filter Expressions
  7.5 Interpretation by URI Pattern
8 languages
  8.1 Interpretation by URI Pattern

Query Parameters in EDAA

The following table is the set of query parameters defined by EDAA Spec.

Parameter	Meaning	Details
page and per_page	Pagination related	see here
alt	Alternative format of the resource representation (eg atom or JSON)	see here
fields	Specify the subset of fields (properties) of the resource that should be returned in the response (like SELECT in SQL)	see here
expand	Augment relationship representations with an in line feed of related resources	see here
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)	see here
filter	simple boolean predicate expression to describe a filter, or subset, of the resources to return in a GET operation (like WHERE in SQL).	see here
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.	see here

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/instances?page=1&per_page=20&alt=atom&fields=displayName,id,connectionStatus&orderby=id&filter=connectionStatus%20eq%20DOWN
```

may be an appropriate (but complicated) request.

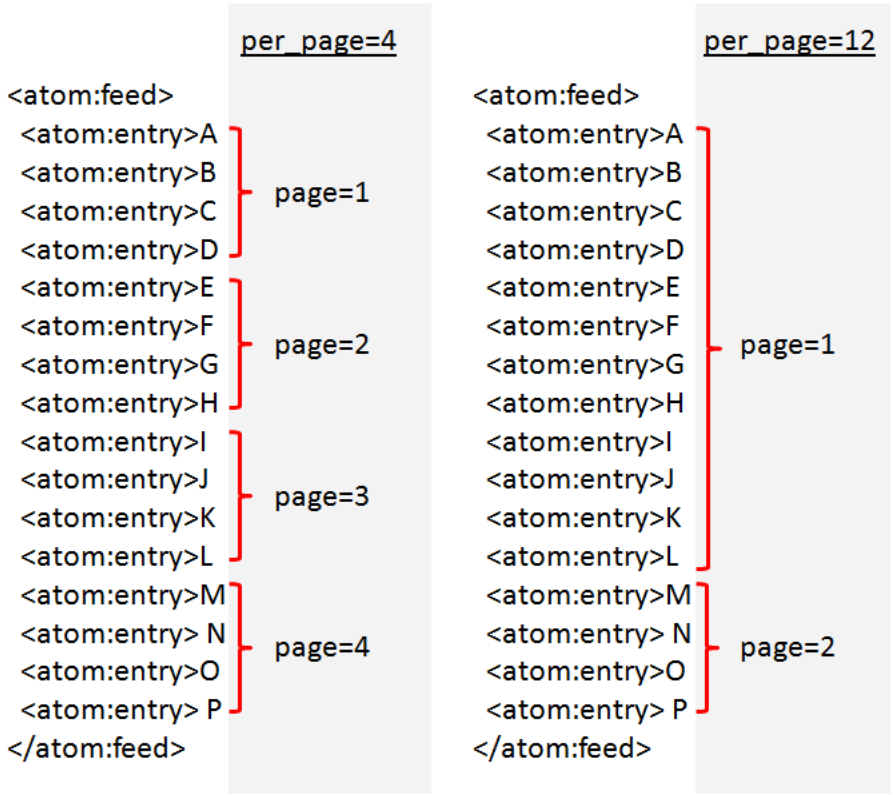
Note also that ?filter and ?orderby affect the interpretation of ?page because those ?filter changes which atom:entry elements are included in the "conceptual" atom:feed over which the request operations and ?orderby changes the order in which those atom:entry elements appear.

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.

The following figure shows a "conceptual" atom:feed of all atom:entries that meet a certain criteria (say, for example, resources of a given type). The figure shows that "conceptual" atom:feed chunked with different page sizes (different values of ?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>H
</atom:feed>
```

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).

The ?page and ?per_page are integral to the processing of RFC 5005 Atom Pagination style atom:link elements in MSA.

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=atom, the serialization of the response is governed by the RFC 4287 - Atom Syndication Format] and rules to represent type resources and instance resources

specified in EDAA.

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/instances?fields=displayName,id,Datacenters
```

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/instances?fields=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::1?fields=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>
```

```
    <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.

Note that although Google's Gdata Partial response includes the possibility of expressing xpath-like expressions in the value of ?fields, in EDAA this sophistication is not currently supported.

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.
/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.
/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
```

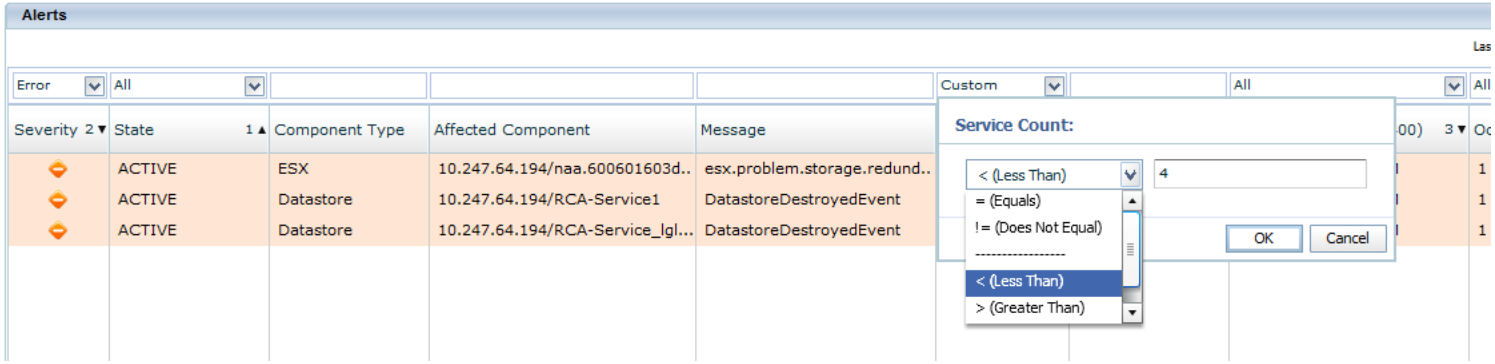
would return a subset of the atom:entry elements for which the expression "attr2 less than 8" evaluates true:

```
<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:



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/instances?filter=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.
```

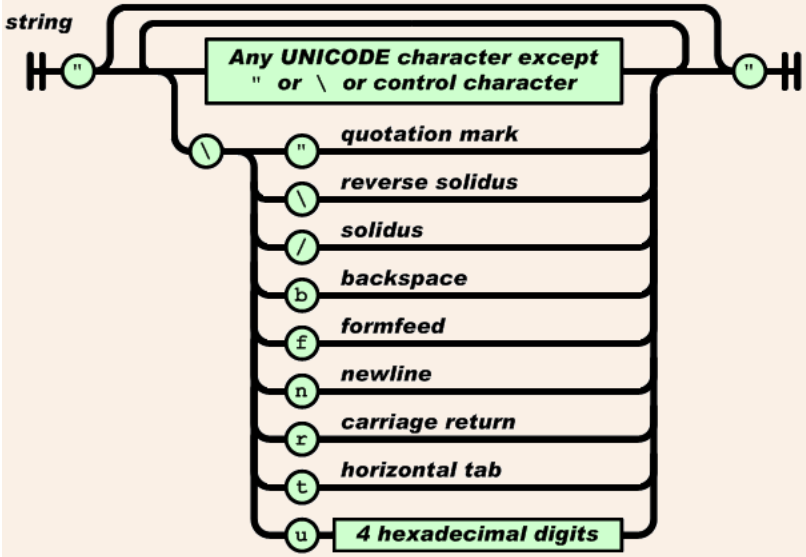
filter expression terms

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.

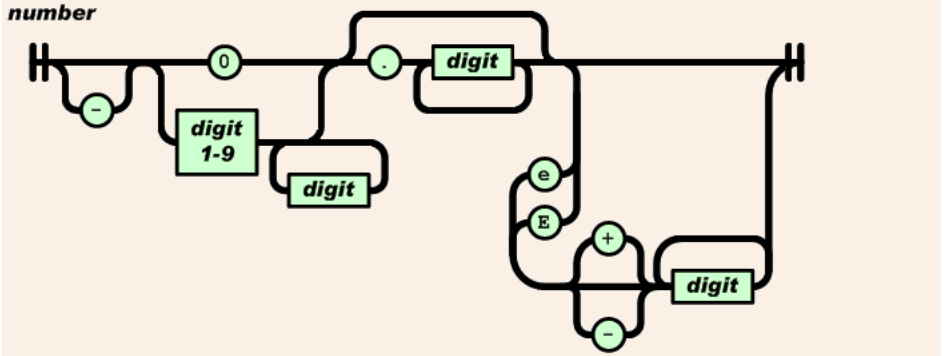
A string is defined by JSON as:

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/instances?filter=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/instances?filter=p1 eq "a" AND p2 eq "b" AND p3 eq 8 OR p3 eq 9
```
2.
```
GET /types/T/instances?filter=(p1 eq "a" AND p2 eq "b" AND p3 eq 8) OR p3 eq 9
```
3.
```
GET /types/T/instances?filter=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/instances?filter=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.

EDAA Primer - Details - Relationships

Contents

1 Relationships
2 Representing Relationships
2.1 Normal Form of Relationship Representation
2.2 Expanded Form of Relationship Representation
2.3 Tradeoffs between the forms of Relationship Representation

Relationships

Recall from the primer discussion on resource representations that a key notion in EDAA is the relationships between resources. A fundamental concept of a resource is that it has properties (things like name, ip address, etc.) and it is related to other resources (things like containment, physical connectivity, etc.). These properties are modeled at the resource type level and at the resource instance level.

Relationships appear in resource instances as atom:link elements, like the example below:

```
<atom:feed>
...
  <atom:entry>
  ...
    <atom:link rel='self' href='http://localhost:8080/slm/msa/instances/vCenter::9' />
  ...
    <atom:content>
    ...
        <some resource instance ...

          <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' />
...
```

Although atom:link elements are used for many other things in EDAA (eg links to action URLs, links to eventing, standard atom thinks like pagination, etc.) an atom:link associated with relationships can be determined by examining the value of @rel. As defined in EDAA spec, if the @rel of the atom:link contains a pattern like {common}/{product}/relationship/{relName}, where {common} currently binds to "http://schemas.emc.com/msa", then the consumer should interpret the atom:link as representing a relationship. The value of @href is a URL to a resource containing a collection of zero or more related resources.

In the example above, we see two atom:link elements. The first one, with @rel='self', is not a relationship atom:link -- this atom:link specifies the identity of the resource being represented in the atom:entry. For convenience of discussion, we will call this resource "vCenter::9".

The second atom:link in the example is a relationship link. What this means is that the resource "vCenter::9" is related to a collection of resources via the "Datacenters" relationship. The collection of related resources can be found at http://localhost:8080/slm/msa/instances/vCenter::9/relationships/Datacenters as contained in the value of @href. If a consumer wanted to know what resources are linked

If the consumer wanted to know more information about what relationships the "vCenter::9" resource has, it can examine the representation of the resource for more atom:links with @rel containing the relationship pattern, or it could examine the type resource associated with "vCenter::9" and see what relationships instances of the "vCenter" type can have. As discussed in the primer discussion on resource types, relationship information at the type level contains all sorts of useful information, for example the type of the resource related through the "Datacenters" relationship. Here is a snippet from the type resource for vCenter:

```
<atom:feed>
...
  <atom:entry>
  ...
    <atom: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>
         ...
         <vsc:relationship relType="vCenterDatacenter" type="http://schemas.emc.com/msa/common/contains" minOccurs="1" maxOccurs="unbounded"
                           description="List of Datacenters">Datacenters</vsc:relationship>

...
```

From the relationship declaration in the vCenter type, we can see that resources related to a vCenter resource in by the Datacenters relationship are of type "vCenterDatacenter".

Representing Relationships

In EDAA, there are two styles of representing relationships with atom:link elements:

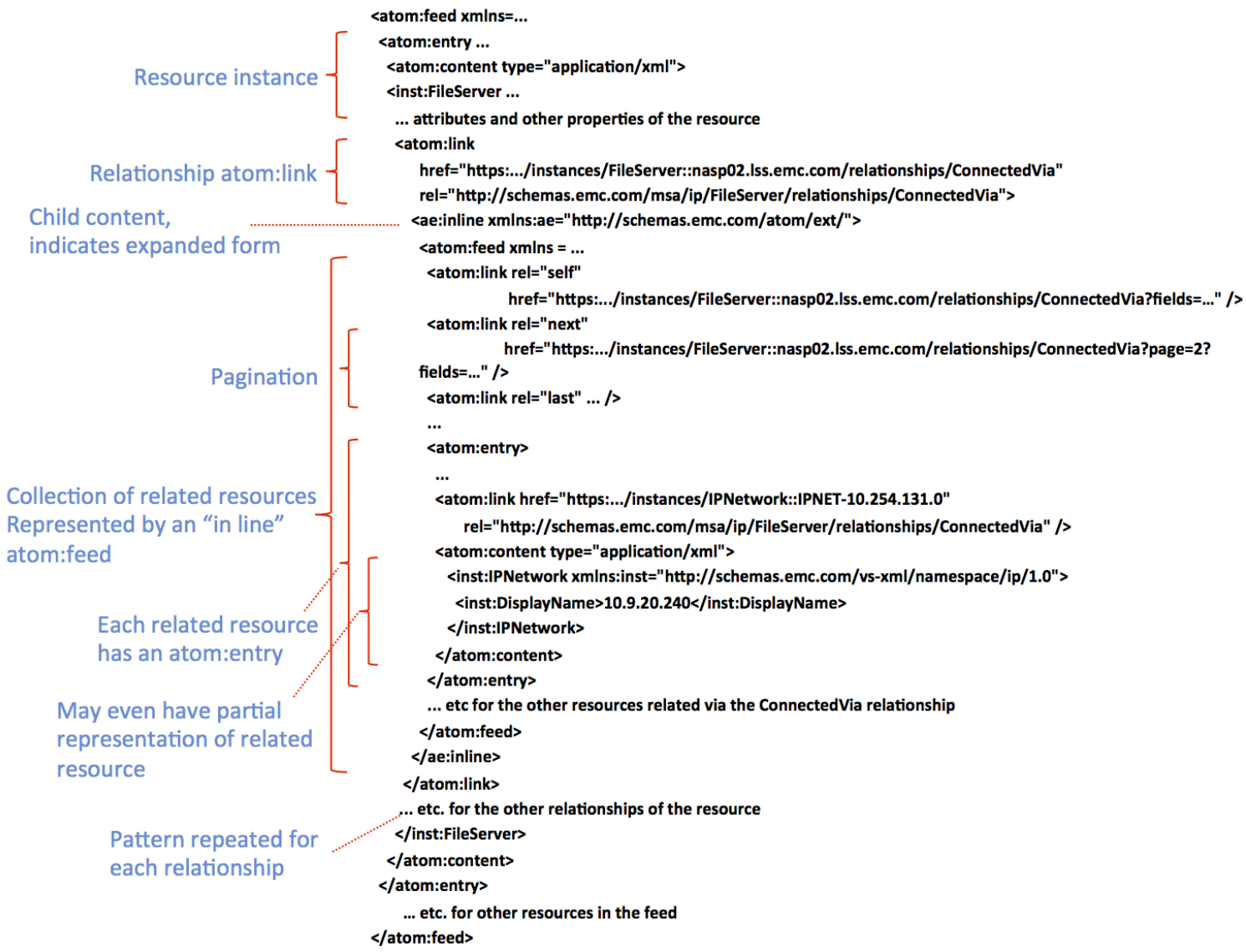
1. the "normal" form, which represents the relationship using an atom:link or its JSON equivalent
2. the "expanded" form, which includes child content of the "normal" form, an inline feed of related resources.

Normal Form of Relationship Representation

We have already seen the normal form of using an atom:link to represent relationships in the example above. The relationship is represented by an atom:link with no child elements. If the consumer wishes to examine the collection of related resources, it must resolve the URL contained in @href and process the atom:feed of related resources.

Expanded Form of Relationship Representation

A consumer would request an "expanded" style for one or more relationships by using the ?expand query parameter. This "expanded" style of relationship representation adds additional information about the related resources to the base atom:link element representing the relationship itself. The expanded form of relationship representation adds a child element to the relationship atom:link containing an atom:feed of the related resources, as shown in the example below:



.

The idea with the expanded form of relationship representation is to trade off an increase in the size of a resource's representation to potentially save a few round-trips from the client to the server. It does this by essentially caching some information about the relationship "resource" as child content of the relationship's atom:link.

The resource's representation is still the same, you have attributes and relationships just like in any resource representation.

The relationships are still modeled by an atom:link element, but, in the expanded form of relationship representation, the relationship atom:link has a child element. EDAA leverages a proposed atom:inline extension to add an atom:feed element as a child of the atom:link element. It is the presence of this "ae:inline" element that distinguishes the normal form of relationship representation from the expanded form of relationship representation.

Think of this subordinate atom:feed as an "in line" cache of the collection of the related resource you would get if you resolved the @href of the atom:link is resolved.

The subordinate atom:feed is paginated, following RFC 5005, so that the size of the response doesn't get too huge. Pagination of the subordinate atom:feeds are under the control of the server and are not influenced by ?page or ?per_page query parameters that may be included in the request.

The subordinate atom:feed itself contains a set of atom:entry elements, one for each related resource via the given relationship. The atom:entry typically contains an atom:link to the resource, but it may also contain an atom:content element containing a

partial representation of that resource. Clearly the size of the partial representation should be kept small, just a few properties of the resource that are very commonly shown in UIs, for example.

Tradeoffs between the forms of Relationship Representation

The normal form of relationship representation is the default form. By default, servers will use this style to represent relationships in resource representations. The benefits of this style is its terseness, the entire relationship is represented in the atom:link with no further information needed. For resources that have lots of relationships, or for responses that include lots of resource representations, the link by relationship style may be the best approach.

The expanded form of relationship representation style is available via the ?expand query parameter. The benefit with this style is that much of the information about the related resources is already present in the response of one GET operation. Also, it is easy for a client to determine if a relationship contains zero related resources (the ae:inline element has no children). This form of relationship representation can expand the size of a resource representation greatly. For a resource that has lots of relationships and/or a resource that has several relationships that have large cardinality (eg 100s of related resources), the expanded form of relationship representation may not be a good choice, as the size of the responses may increase latency of the request/response, use a lot of network bandwidth and require too much memory/cpu to process for both the client and the server.

EDAA Primer - Details - JSON

This topic discusses how EDAA supports JSON as an additional serialization mechanism, augmenting atom/xml support in EDAA.

Contents

1 atom/xml vs JSON
1.1 atom/xml
1.1.1 Benefits of Atom and XML
1.2 JSON
1.2.1 Benefits of JSON
2 EDAA support of Serialization formats
3 Consumer choice

atom/xml vs JSON

In any REST API, or distributed data computing environment in which data is being transferred between systems over a network, there is an important question to resolve: "in what form is the data serialized"? Clearly there are many choices for data serialization format, eg ascii-XML, JSON, comma separated variables (CSV), various binary representations, proprietary schemes etc. Tradeoffs are often made between compactness of the representation (for network efficiency), serialization/deserialization performance, readability by humans, ease of use, etc.

In practice within the REST-based Web services community there are two popular choices for data serialization format:

1. XML, particularly combined with Atom Syndication Format, and
2. JSON

EDAA standardizes on 2 serialization formats: atom/xml and JSON. EDAA doesn't restrict implementations from having additional serialization formats, each EDAA implementation MUST have one of atom/xml or JSON and SHOULD support both.

There is a growing conversation within the web community about whether to use XML or JSON. Norman Walsh, in his blog commented on a trend of some web properties (eg Twitter, Foursquare) to support JSON only, giving a nice summary of pros/cons . Justin Cormack in his blog says the JSON vs XML tradeoff is not clear cut.

Net/net, neither form of serialization is superior. A lot depends on what the client environment looks like (eg which browser, which Javascript library) and aspects of the application itself.

atom/xml

The idea with atom/xml (Content-type - "application/atom+xml") is to serialize data in terms of feeds (collections) and entries (collection members). Atom, is short form for Atom Syndication Format defines a standard way to represent collections and members using XML. There are a set of related or derivative specifications that are included in the atom "family", specifically RFC5005 atom

pagination , that are important to REST API styles like ODAA.

ODAA also uses the atom:link construct from atom to represent links between resources.

A good introduction to atom can be found here. Of course, to understand atom you need a good grounding in XML. A good introduction to XML can be found here.

Benefits of Atom and XML

- XML is a better fit for semi-structured data than is JSON (ie mixed content)
  - that being said, not a lot of applications of EDAA have a strong need to represent semi-structured data
- XML (via XML Schema Definitions or XSDs) has a mechanism to validate instance documents against a definition of a "valid" structure
- XML has tons of interesting standards (XPath, XQuery, XProc, XSLT, etc.) around which really useful tools have been built
- Atom has nice standards for things like pagination (RFC 5005), in JSON things like pagination are basically "roll your own"
- Some perceive XML to be more secure than JSON, especially in browser-client situations, because the way JSON is deserialized into objects, using Javascript eval(), can be a potential security hole (you have to trust that the provider, or some malicious intermediary, hasn't included dangerous JavaScript code in the JSON payload).
  - That being said, there are a growing number of JSON libraries that deserialize in a safer fashion.

JSON

JSON is a simple data serialization format derived from Javascript. The idea is that JSON is a text-based representation of objects that can be parsed into all sorts of different programming languages in addition to Javascript.

A quick introduction to JSON can be found here.

JSON is a simple data representation format based on two constructs:

1. collections of name/value pairs, and
2. ordered lists (arrays) of things

It really is that simple. Of course, given a simple set of "building blocks" rich, sophisticated data structures can be represented in JSON.

Examples of how JSON is used in EDAA can be found in the EDAA Spec and elsewhere in the Primer.

Benefits of JSON

- natural fit with Javascript and browser based applications, simple to serialize/deserialize
- simpler than XML for simple types, collections of simple types, etc.
- overall simplicity, smaller spec, fewer moving parts
- There are suggestions that JSON is faster to parse than XML, but that varies with different tools, browsers etc. One article, [1] has some performance comparisons that favor XML.

EDAA support of Serialization formats

An EDAA implementation MUST support at least one of the EDAA standardized data serialization format (either atom/xml or JSON). An

EDAA implementation SHOULD support both atom/XML and JSON. An EDAA implementation MAY support additional data serialization formats.

The way the atom/xml data serialization format is used in EDAA is governed by the RFC 4287 - Atom Syndication Format and the EDAA rules to represent type resources and instance resources.

The way the JSON data serialization format is used in EDAA is governed by the JSON conventions in EDAA and the EDAA rules to represent type resources and instance resources.

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
```

EDAA Primer - Details - Modifying Resources

This topic reviews a set of "patterns" that developers should consider when they attempt to enhance the EDAA capabilities of their product to support operations that create/update and delete resources or otherwise allow consumers to directly modify data within their product through an EDAA interface.

Contents

1 Thinking about Read/Write EDAA - Patterns for EDAA Developers
2 Keep it simple - Wherever possible, use the simplest possible solution to satisfy the use case
  2.1 Discussion
  2.2 Simple "Single Resource" Examples
  2.3 Operations on Relationships Examples
3 Complex Context - Where simple POST/PUT/PATCH/DELETE on a single resource don't quite fit
  3.1 Discussion
  3.2 Aggregate Resource Design Pattern
    3.2.1 Factory Pattern
    3.2.2 Plan Partial Failure
  3.3 ZoneSet, Zone and ZoneMember Example
    3.3.1 Incremental Approach
    3.3.2 Factory Pattern Approach
4 Keep it clear -- Metadata and Partial Representations
  4.1 Discussion
  4.2 Examples
5 Don't keep the user waiting - use Task resources
  5.1 Discussion

Thinking about Read/Write EDAA - Patterns for EDAA Developers

Here is an approach to address create/update/delete operations in EDAA. Because EDAAs will be used in a wide variety of situations, we identify some patterns of applying REST to address various design challenges. We have identified four patterns to be used when attempting to add create/update/delete or action semantics to an ODAA

2. Keep it simple -- if a straight forward application of the REST uniform interface (PUT/POST/PATCH/DELETE) works, then use it
3. Cover context thoughtfully -- beyond simple PUT/POST/PATCH/DELETE
4. Keep it clear -- Clarify consumer requirements by describing create and update partial representations
5. Don't keep the user waiting -- Asynch notification and the Task Resource

We examine each pattern enumerated above, discussing details/motivation and outlining an example use case and solution to illustrate the techniques.

Keep it simple - Wherever possible, use the simplest possible solution to satisfy the use case

If it is possible to solve the problem using a direct application of the uniform interface (PUT/POST/PATCH/DELETE) on a resource, then do it. Simple good, complex bad. Complexity should be introduced only when it is agreed the simple application of the REST uniform interface is not appropriate.

Discussion

Try to keep a RESTful approach wherever possible; use the uniform interface: POST (or PUT) to create resources, PUT or PATCH to modify resources, DELETE to delete resources.

Remember that PUT and DELETE should be idempotent.

PUT, when used for update, means replace then entire representation of a resource with the body of the message. Clearly, some properties of the representation (eg identifiers, computed values, some relationships, etc.) cannot be modified by consumers.

Simple "Single Resource" Examples

1. Simple create of a resource of a given type
   - e.g. to create a new VCenter resource
     - POST /types/VCenter/instances with the body of the message containing a valid partial representation (for create) of a vCenter resource
     - See also EDAA Read/Write - Creating New Resources
2. Update of a resource's representation (complete overwrite of state)
   - e.g. to modify the entire state of a given Alert resource
     - PUT /instances/Alert::1234, with the body of the message containing a valid representation of an Alert resource, any property that is not valid to be modified by third parties will be ignored.
3. Update of a subset of a resource's representation
   - e.g. to modify the assignedTo property of an Alert resource
     - PATCH /instances/Alert::1234, with a small number of properties that can be modified by third parties (e.g. cannot change the created date/time property). See also EDAA Read/Write - PUT vs PATCH.
4. Simple delete of a resource of a given type
   - e.g. close out an Alert resource and remove it from the system (either hard delete or soft delete)
     - DELETE /instances/Alert::1234, removes the resource from visibility in the system. The resource may still be there (e.g. soft delete) for purposes of audit trails etc, but it will not be visible to any of the REST API operations.
   - It is up to the EDAA implementation to cascade this delete operation to any relationships the deleted resource may have participated in. EDAA does not require this cascading behavior, but it is recommended that the EDAA SHOULD modify those relationships as part of the processing of the DELETE operation.

Operations on Relationships Examples

We can observe that relationships are also resources that can be operated on using the REST uniform interface.

1. Simple add a resource across a relationship
   - e.g. add a "user" resource to a "group" resource (eg in an Authentication/Authorization domain)
     - POST /instances/Group::powerUsers/relationships/includesUser, where the body of the POST message contains a URI to the User resource to be added to the Group resource
2. Create a new resource in the context of a relationship
   - e.g. create a new User resource in the context of that User being a member of a given Group
     - POST /instances/Group::powerUsers/relationships/includesUser, where the body of the POST message contains a partial representation of a User resource to be created and added to the identified group.
3. Add an existing resource to a relationship
   - e.g. add an existing User resource in the context of that User being added as a member of a given Group
     - POST /instances/Group::powerUsers/relationships/includesUser, where the body of the POST message contains a representation of a User or an atom:link to a User resource
4. Delete all associations from a relationship
   - e.g. remove all the User resources in the "includesUser" from the powerUsers group
     - DELETE /instances/Group::powerUsers/relationships/includesUser
   - This will disassociate all users from the includesUser relationship for the powerUsers group. The User resources themselves are NOT modified. Upon successful completion of the above operation, a GET operation on /instances/Group::powerUsers/relationships/includesUser returns an atom:feed with zero atom:entry elements.
   - Note, sometimes the DELETE will have sideffects. For example if the "includesUser" relationship from Group->User was part of a bi-directional relationship, where the User->Group part of the bi-directional relationship was modeled by a "partOfGroup" relationship on the User type, then the DELETE mentioned in the above bullet would have a side effect of deleting the "partOfGroup" associations.
   - It is the responsibility of the service implementation to ensure that DELETE is idempotent.
   - Similarly, in the case of managing storage provisioning for clusters, DELETE a Server<->ServerGroup relationship and all of the Server<->StorageVolume relationships are deleted as a side effect, even though the Server itself may not be deleted. Likewise, because a Server resource should exist independent of a ServerGroup, then assuming it exists first, then adding the Server<->ServerGroup relationship would have a side effect of adding relationships from that Server to all of the StorageVolumes in the associated StorageGroup for the cluster.
5. Removal of an individual resource from a relationship
   - In order to support this, each association would need its own edit URI. The current version of EDAA does not model associations with their own, separate URIs. For those relationships that are implemented to support deletion of individual associations, each association MUST be assigned its own URI in order that it be the object of a DELETE message to be sent to that association resource, thereby removing an individual resource from the relationship.
   - Consider the /instances/Group::powerUsers/relationships/includesUser relationship resource. If it contained association URIs, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:vsc= ...
<entry>
  <link rel="edit" href="https:.../instances/e7b7fd5e-c5f7-4674-be2f-1af80c4ad1b8" />
```

```
  <content type="application/xml">
    <inst:Interface xmlns:inst="...
       <atom:link rel="self" href="https:.../instances/User::sgg"/>
       <inst:Name>sgg</inst:Name>
    ...
  </content>
</entry>
<entry>
  <link rel="edit" href="https:.../instances/7462a872-3be8-4c6a-b309-44f11e34e076" />
  <content type="application/xml">
    <inst:Interface xmlns:inst="...
       <atom:link rel="self" href=".../instances/User::Edgar"/>
       <inst:Name>Edgar St. Pierre</inst:Name>
    ...
  </content>
</entry>
...
```

- A consumer could examine the entry corresponding to user "sgg" and examine that entry to see if it contains a URI with @rel containing "edit". If it does, then sending a DELETE message to the corresponding URI would accomplish the consumer's goal of deleting the user "sgg" from the relationship.
  - See EDAA read/write spec for more details.

Complex Context - Where simple POST/PUT/PATCH/DELETE on a single resource don't quite fit

Context is often more complex than just a single resource.

Discussion

From the requirements listed above there are four major categories of context we need to think about:

1. Context isolated to a single resource
   - Use simple POST/PUT/PATCH/DELETE operations directly on the resource's URI (straight forward, simple)
   - Consumer knows what is possible by the OPTIONS verb or by atom:link elements in the resource's representation
   - See above for examples
2. Context involves a single relationship
   - Many richer examples have Resource1–> RelationshipA--> Resource2 as sufficient context
   - In this case, the context is pretty straight forward to represent in REST.
   - Add existing resources to a relationship, create a new resource in the context of a relationship, remove a resource from a relationship
   - See above for examples
3. Verb-style actions (REST-hybrid)
   - Sometimes PUT/POST/PATCH/DELETE on a single resource, or relationship doesn't make sense, e.g. for an operation like "reboot a server"
   - In these situations, as a last resort, use an "action-verb" pattern
   - The resource representation contains a set of atom:link elements, one for each "action" that is applicable for resources of that type
     -

with atom:link rel="{URI of action}" elements to identify the action and the value of @href indicates how to achieve the action.
- The /type/{typename} resource will also include a list of atom:link elements that suggest actions that MAY be available on resources of that given type

4. Richer context, where there are multiple relationships involving multiple resources.
   - This is where REST design will get tricky.
   - This is where the "aggregate resource" pattern comes in handy, to have the new (sometimes artificial) aggregate resource encapsulate the complicated resource configuration, and then that configuration is somehow "activated" or "provisioned" to modify the IT Infrastructure in a complex manner.
   - Also associated with the notion of a richer context is the "factory pattern"
   - See below for an example

Aggregate Resource Design Pattern

Sometimes an operation requires a rich orchestration of changes to a collection of related IT Infrastructure Resources. The notion of an "Aggregate Resource" allows us to maintain simplicity at the interface level by building an abstraction that can be implemented within the service implementation to hide the complex details. This pattern can also be known as a "controller resource pattern". The trick here is to identify the right abstraction.

The UIM product, for example, deals with the notion of provisioning host, switch and storage resources in support of management of VBlock installations. In UIM, they developed the notion of the "Service" resource and "ServiceOffering" resource. These resources represent "abstractions" or aggregations. A consumer uses standard REST interfaces on a ServiceOffering to configure it with service profiles. A consumer can create a Service resource, based on a ServiceOffering using standard REST approaches. The REST interface supports an action on a Service, called "activate", that causes the pattern configured in the Service resource to be "realized" or "provisioned" on the underlying physical resources and virtualized resources.

This notion of an aggregate resource design pattern, provides a mechanism for an API designer to control how certain sets of provisioning actions can be combined. this simplifies the REST API (no need for distributed transactions over multiple operations or complicated box-carring approaches to batch operations into a single unit of work). This also simplifies the service implementation in that it constrains what set of operations on the underlying IT Infrastructure can be achieved to those operations defined on the aggregated resource.

Factory Pattern

Associated with the Aggregate Resource Design Pattern, is the possibility of creating a "collection" or "network" of linked resources in one request. The notion of a "Factory" that, given a "plan" of the resources and their interconnections, could create a resource and all its related resources in one request.

For an example of using the factory pattern, see ZoneSet example.

As noted in the ZoneSet example, support for the "factory pattern" style of creating resources requires EDAA support of the Alias concept.

Plan Partial Failure

When using a Factory Pattern to building up a network of related resources, the plan author needs to be aware of the possibility of partial failure situations.

As discussed in the EDAA read/write spec, the response to the POST action MAY be a reference to a Task resource. The status of the plan creation, whether the activity is still in progress, whether the action completed or whether there were errors in the processing of the request are all contained in the Task resource. The plan author would examine the Task resource to determine the state of the processing and to understand if there are any error conditions associated with the plan processing.

From the perspective of the consumer, the cleanest approach to "partial failure" is an "all or nothing" semantic. By this we mean that when a plan is submitted and the execution of that plan is not successful, then all resources created in the process of executing the plan are removed from the system and not exposed to the user. It will be as if the plan was never attempted as far as the set of resources in the system is concerned. The Task resource can contain error information to inform the plan author where the problem is. The plan author can then make modifications to the original plan and resubmit. There is no onus on the plan author to attempt to piece together which parts of the plan got executed, which resources were created and then rework the original plan to not create those resources, but only attempt the resources that were not successfully created. This is a lot of work for the plan author.

ZoneSet, Zone and ZoneMember Example

This example is based on a simplified information model of ZoneSets, Zones and ZoneMembers. These resources are important objects in a SAN Fabric. We choose this example because it illustrates a non-trivial collection of interrelated objects and illustrates a situation where the design needs to separate the creation of the "plan" (ie collection of resources and their inter relationships) from the realization of this plan by configuring the underlying SAN Fabric. The creation of the plan uses a series of simple POST operations to build up the network of related resources and the realization of the plan uses an aggregate resource pattern operation.

The relationships between these resource types is fairly simple. A ZoneSet resource can have zero or more related Zones. A Zone can have zero or more related ZoneMembers. A Zone can be in zero or more ZoneSets. A ZoneMember can be in zero or more Zones.

This example will not dive into other details of the resource model (eg the properties of each resource, other relationships, etc.).

The consumer wishes to create a ZoneSet with the following "plan":

- ZoneSet is named "ZS1"
- ZS1 has 2 Zones: named "Z1" and "Z2"
- Z1 has 2 ZoneMembers ZMa and ZMb
- Z2 has 3 ZoneMembers ZMb, ZMc and ZMd

and to illustrate "link by value" and "link by reference", assume ZoneMember ZMb already exists as a resource.

This example shows two major patterns of creating networks of inter-related resources:

1. **incremental approach** - where the consumer issues a series of individual resource creation requests
2. **factory pattern approach** - where the consumer creates a single plan representation and POSTs that plan to a factory resource

Incremental Approach

This approach uses simple POST operations to type resources and relationship resources. It is simple, natural application of REST patterns.

1) Create the ZoneSet resource

```
POST /types/ZoneSet/instances HTTP/1.1
...
<inst:Name>ZS1</inst:Name>
... other properties
```

and the response would be

```
HTTP/1.1 201 Created
Location: http://.../instances/ZoneSet::ZS1
...
```

2)Create the Zone resource "Z1" using POST to the ZoneSet --hasZones--> Zone relationship

```
POST /instances/ZoneSet::ZS1/relationships/hasZones HTTP/1.1
...

<inst:Name>Z1</inst:Name>
... other properties
```

and the response would be a reference to the newly created zone (which, because the POST was to the relationship, is added to the ZoneSet --hasZones--> Zone relationship for ZoneSet::ZS1)

```
HTTP/1.1 201 Created
Location: http://.../instances/Zone::Z1
...
```

3)Create the Zone resource "Z2" using POST to the ZoneSet --hasZones--> Zone relationship
Essentially the same as step 2.

4)Create the ZoneMember resource "ZMa" using POST to the Zone --hasMember--> ZoneMember relationship with Zone::Z1 created in step 2.

```
POST /instances/Zone::Z1/relationships/hasMembers HTTP/1.1
...

<inst:Name>ZMa</inst:Name>
... other properties
```

and the response would be a reference to the newly created ZoneMember, and of course this ZoneMember has been added to the Zone --hasMember--> ZoneMember relationship for Zone::Z1

```
HTTP/1.1 201 Created
Location: http://.../instances/ZoneMember::ZMa
...
```

Note, an alternative approach to accomplishing this is:

```
POST /types/ZoneMember/instances HTTP/1.1
```

```
<inst:Name>ZMa</inst:Name>
... other properties of the ZoneMember
<atom:link rel="http://schemas.emc.com/msa/san/ZoneMember/relationships/memberOfZone"
           href="http://.../instances/Zone::Z1"
/>
```

and the response would be

```
HTTP/1.1 201 Created
Location: http://.../instances/ZoneMember::ZMa
...
```

This would create the ZoneMember and also connect it up to the Zone::Z1 via the ZoneMember --memberOf--> Zone relationship. Note, we are not implying that ZoneMember --memberOf--> Zone and Zone --hasMembers--> ZoneMembers are automatically bi-directional relationships. The information model would need to decide which relationship (or both) is part of the model. If both relationships need to be maintained, then the above message body would need to be posted to /instances/Zone::Z1/relationships/hasMembers.

5)Link the pre-existing ZoneMember resource "ZMb" using POST to the Zone --hasMember--> ZoneMember relationship

```
POST /instances/Zone::Z1/relationships/hasMembers HTTP/1.1
...
<atom:link rel="http://schemas.emc.com/msa/san/Zone/relationships/hasMembers" href="http://.../instances/ZoneMember::ZMb" />
```

and the response would be

```
HTTP/1.1 200 Ok
Location: http://.../instances/Zone::Z1/relationships/hasMembers
...
```

6) Populate Zone Z2 with ZoneMember ZMb
Essentially the same as step 5.

7) Populate Zone Z2 with ZoneMember ZMc and ZMd
Essentially step 4 repeated two times, once for ZMc and once for ZMd.

Now, the ZoneSet is populated as required. The ZoneSet topology can be examined and confirmed by the consumer (through GET operations) prior to the consumer doing an "activate" action (POST /instances/ZoneSet::ZS1/action/activate) to activate the ZoneSet and related resources.

Note, this example does illustrate the need for adding a property of ZoneSet, Zone and ZoneMember to distinguish "activated" resources from those resources that are merely in the planning stage. This sort of "state" property could be used by the consumer to distinguish those ZoneSet, Zone and ZoneMember resources that are "activated" from those that are not.

Factory Pattern Approach

This approach involves having a separate resource, a so-called Factory resource, that is responsible for "manufacturing" a resource and its network of related resources based on a "plan" supplied by the consumer in the body of a POST operation on a URI to the

factory.

For our running ZoneSet creation example, here is a payload, that when POSTed to the ZoneSetFactory resource, would cause the ZoneSet "ZS1" containing two Zones (Z1 and Z2) and their ZoneMembers to be created and linked together.

```
<inst:ZoneSet xmlns:inst="http://schemas.emc.com/UIM/1.0">
  <atom:link rel="self" href="http://schemas.emc.com/msa/Alias/ZS1" > (See Note: (1))
  <inst:Name>ZS1</inst:Name>
  ... etc for the rest of the properties definable through the partial representation (for create) of a ZoneSet resource

  <atom:link rel="http://schemas.emc.com/UIM/ZoneSet/relationships/hasZone" href="http://schemas.emc.com/msa/Alias/Z1" >  (See
Note: (1))
    <inst:Zone xmlns:inst="http://schemas.emc.com/UIM/1.0">
      <inst:Name>Z1</inst:Name>
          ... etc for the rest of the properties definable through the partial representation (for create) of a Zone resource

      <atom:link rel="http://schemas.emc.com/UIM/Zone/relationships/hasMember" href="http://.../instances/ZoneMember::ZMa" >
        <inst:ZoneMember xmlns:inst="http://schemas.emc.com/UIM/1.0">
          <inst:Name>ZMa</inst:Name>
             ... etc for the rest of the properties definable through the partial representation (for create) of a ZoneMember
resource
        </inst:ZoneMember>
      </atom:link>
      <atom:link rel="http://schemas.emc.com/UIM/Zone/relationships/hasMember" href="http://.../instances/ZoneMember::ZMb" >
    </inst:zone>
  </atom:link>

  <atom:link rel="http://schemas.emc.com/UIM/ZoneSet/relationships/hasZone> href="http://schemas.emc.com/msa/Alias/Z2" >  (See
Note: (1))
    <inst:Zone xmlns:inst="http://schemas.emc.com/UIM/1.0">
      <inst:Name>Z2</inst:Name>
         ... etc for the rest of the properties definable through the partial representation (for create) of a Zone resource

      <atom:link rel="http://schemas.emc.com/UIM/Zone/relationships/hasMember> href="http://.../instances/ZoneMember::ZMb" >
      <atom:link rel="http://schemas.emc.com/UIM/Zone/relationships/hasMember> href="http://schemas.emc.com/msa/Alias/Zmc" >
(See Note: (1))
        <inst:ZoneMember xmlns:inst="http://schemas.emc.com/UIM/1.0">
          <inst:Name>ZMc</inst:Name>
               ... etc for the rest of the properties definable through the partial representation (for create) of a
ZoneMember resource
          </inst:ZoneMember>
      </atom:link>

      <atom:link rel="http://schemas.emc.com/UIM/Zone/relationships/hasMember> href="http://schemas.emc.com/msa/Alias/Zmd" >
(See Note: (1))
        <inst:ZoneMember xmlns:inst="http://schemas.emc.com/UIM/1.0">
          <inst:Name>ZMd</inst:Name>
             ... etc for the rest of the properties definable through the partial representation (for create) of a ZoneMember
resource
        </inst:ZoneMember>
      </atom:link>

      </inst:zone>
  </atom:link>

</inst:ZoneSet>
```

Note:
(1): When building plans for submission to Factory resources, it is often useful to be able to reference resources about to be created by the plan in various other parts of the plan. In this example, the ZoneSet about to be created is aliased by <atom:link rel="self" href="http://schemas.emc.com/msa/Alias/ZS1" >. Allowing that resource to be referenced in other parts of the plan. In this case, this resource is not referenced elsewhere in the plan. See EDAA read/write spec for a further description of the Alias concept.

Keep it clear -- Metadata and Partial Representations

How does a consumer know what to put in the body of a POST message to create a resource? What goes into the body of a PUT message to modify a resource?

Discussion

As was noted in the requirements, many properties (and relationships) of a resource can be created or updated at any time, some properties (and relationships) can be specified at create time only, some properties cannot be created or updated by consumers. Typically what happens is that the information modeling exercise becomes a bit richer. For any given resource, a “full” (read) representation schema is declared, and a create (partial) representation and update (partial) representation is also declared. These partial representations provide clarity to application designers on what can be modified where.

Even though we provide rich mechanisms in EDAA to describe the API, there's simply no substitute for good human readable documentation describing the API behavior.

Several common needs:

1. The consumer needs to know what operations are valid on a given resource
2. The consumer needs to know what the valid create partial representation is (what properties and relationships need to be specified)
3. The consumer needs to know what the valid update partial representation is (what properties and relationships need to be specified)

Examples

What operations are supported

- Use an HTTP OPTIONS message, to determine which HTTP verbs are supported by the resource
- Do a GET on /types/{typeName} resource to see what actions can be typically performed on instances of that type
- Use the atom:link rel="edit" to give the url (often the same one as rel="self") to which PUT/POST/PATCH/DELETE messages may be sent
  - e.g. If the Alert Type permits consumers to modify existing instances, then a representation of that Alert resource would contain
    - <atom:link rel=”edit” href=”https://.../instances/Alert::{ID}"/>
- Use the atom:link with rel="{uri of action}" where the uri of action identifies exactly what action is invoked when a POST message is sent to the URI contained in the href attribute.
  - e.g. If a Server resource permits consumers to issue a reboot action, then a representation of that Server resource would contain
    - <atom:link rel=”http://schemas.emc.com/msa/Server/action/reboot” href=”https://.../instances/Server::foo/action/reboot"/>

What create partial representations are permitted

- See EDAA Spec - Partial Representations for an example of how the consumer can determine what properties of a resource MUST

be included in a *creation* operation for a given resource Type.

Don't keep the user waiting - use Task resources

Don't force the consumer's application to block if the response to a request is going to take time to complete.

Discussion

Some operations have little or no latency (eg less than 1 second); in these cases, implementing a synchronous request/response message exchange pattern is simplest and best.

However some operations may take more time than is reasonable to make the consumer block for a response to his/her request. In these cases, EDAA implementations should take advantage of the HTTP 202 (Accepted) response code and immediately respond with a Task resource to encapsulate the long running process associated with fulfilling the initial request from the consumer.

When a consumer receives an HTTP 202 response code, the client logic can examine the Task resource to find out more information about how it can keep track of status (polling or subscription to event notification) and what is the actual status of the operation. When the operation completes, the status of the Task resource changes and the resource will contain either error information (if the task failed) or a URI to the resource created/modified as a result of the operation. The client can then display the error message, or do a GET operation on the URI to examine the results of the operation.