White Paper on Guidelines for REST API specifications in OMA

Approved – 24 Jul 2012

Open Mobile Alliance

OMA-WP-Guidelines_for_REST_API_specifications-20120724-A

Contents

Figures

Tables

1. Scope

This White Paper (WP) contains guidelines for the development of REST API specifications in OMA.

2. References

[Fielding]	"Architectural Styles and the Design of Network-based Software Architectures" , Roy Fielding, 2000, URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[HTML FORMS]	"HTML Forms", W3C Recommendation, URL:http://www.w3.org/TR/html401/interact/forms.html
[JSON]	Java Script Object Notation, URL:http://www.json.org/
[OMADICT]	"Dictionary for OMA Specifications", Version 2.8, Open Mobile Alliance™, OMA-ORG-Dictionary-V2_8, URL:http://www.openmobilealliance.org/
[REST_TS_Common]	"Common definitions and specifications for OMA REST interfaces", Open Mobile Alliance™, OMA-TS-REST_Common-V1_0, URL:http://www.openmobilealliance.org/
[RFC1738]	"Uniform Resource Locations", ", T. Berners-Lee, L. Masinter, M. McCahill, December 1994, URL:http://www.ietf.org/rfc/rfc1738.txt
[RFC2388]	"Returning Values from Forms: multipart/form-data", L. Masinter, August, 1998, URL:http://www.ietf.org/rfc/rfc2388.txt
[RFC2616]	"Hypertext Transfer Protocol -- HTTP/1.1", R. Fielding et. al, June 1999, URL:http://www.ietf.org/rfc/rfc2616.txt
[RFC3986]	" URI Generic Syntax", T. Berners-Lee et al., January 2005, URL: http://www.ietf.org/rfc/rfc3986.txt
[RFC4627]	"Application/json media type", D. Crockford, July 2006, URL:http://www.ietf.org/rfc/rfc4627.txt

3. Terminology and Conventions

3.1 Conventions

This is an informative document, which is not intended to provide testable requirements to implementations.

3.2 Definitions

For the purpose of this WP, all definitions from the OMA Dictionary [OMADICT) apply.

3.3 Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CRLF	Carriage Return Line Feed
CRUD	Create, Read, Update, Delete
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	Java Script Object Notation
MIME	Multipurpose Internet Mail Extensions
MMS	Multimedia message Service
OMA	Open Mobile Alliance
REST	REpresentational State Transfer
SMS	Short Message Service
SP	SPace
TS	Technical Specification
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
UTF	Universal Transformation Format
WP	White Paper
XML	Extensible Markup Language

4. Introduction

This WP is intended to provide the guidelines for defining REST interfaces in OMA.

The REST (REpresentational State Transfer) architecture was defined in 2000 by Dr Roy Fielding [Fielding]. The key principles of REST are that clients and servers (typically in an HTTP system) interact via requests and responses. These requests/responses transfer representations of a resource; which is identified and addressed by a Uniform Resource Identifier (URI). REST promotes the use of HTTP verbs (GET, POST, PUT, and DELETE) to allow the client to query the current state of the resource, or to change it. By reusing these verbs, as well as HTTP principles of authentication, caching and content negotiation; it is possible to build relatively simple APIs based on existing Web standards [RFC2616].

5. Principles for defining REST APIs in OMA

1. A key guideline is that REST APIs are intended for use by typical web developers. These developers are assumed not to have a detailed understanding of telecoms services and will need to be able to leverage the OMA specified REST services as simply as they would leverage services from major web players, service providers or platforms.

   Therefore, OMA specified REST APIs should provide the same level of easy-to-use as other popular REST services provided on the Web. Wherever technically feasible, REST APIs would be used by applications acting on behalf of the end user (e.g. web site, portal), other specialized applications (SMS campaign managers, various notification services etc) or applications located on the end user device (e.g. mobile phone, dvd player). The cases where the OMA specified REST APIs specified do not serve well a particular client environment have to be identified, analyzed, documented and addressed (in the same Work Item, or a different Work Item, as deemed appropriate).

2. REST API specifications should conform to the REST & HTTP practices, in particular:

   a. Services should be defined in terms of resources that are addressable as URIs.

   b. Use of nouns in URIs is recommended over the use of verbs
      - URIs identify resources
      - HTTP methods identify Operations

   c. Use HTTP verbs, i.e. POST, GET, PUT, DELETE for CRUD (Create, Read, Update, Delete) operations, for all interfaces for which CRUD is a good fit , using the following mapping:
      - POST
        - POST maps to Create, if the HTTP client sends a request to the HTTP server to create a *subordinate* of the specified resource (a.k.a. creating a new member of the resource collection), using some server-side algorithm.
        - POST maps to Update if the HTTP client sends a request to the HTTP server to partially update the specified resource, or to update one or more *subordinates* of the specified resource
        - Note: In certain cases, POST may be used when the operation cannot be mapped to a CRUD operation. For example transformational update of the resource space is usually difficult to map to a CRUD operation (e.g. batch update, etc).
      - GET maps to Read. GET must be safe (i.e. it cannot change a resource), and must be idempotent (i.e. the outcome of calling it multiple times is the same as calling it once - unless somebody else changed the resource between calls)
      - PUT
        - In case the URI addressed by the PUT operation points to an existing resource, PUT maps to a complete Update of the resource, and must be idempotent.
        - In case the URI addressed by the PUT operation does not point to an existing resource, PUT maps to Create of that resource, if that operation is permitted.
      - DELETE maps to Delete, and must be idempotent

   d. Use standard HTTP Status codes in responses for both successful and failed operations. In the case of a failed operation additional status information (if available) will be returned in the body of the response.

   Use of HTTP status codes in response should be consistent with [RFC2616] and in case of successful operations it is recommended to use the following Status codes:

   POST: for successful response, these are the allowed values:

   200 (OK): when no resource URL is provided in the response but the body of the response includes the entity that describes the result.

   201 (Created): if a resource has been created on the origin server, the body of the message SHOULD contain an entity which describes the status of the request and refers to the new resource, and a Location header

   204 (No content): when no resource URL is provided in the response and it does not provide a body.

PUT:

200 (OK) or 204(No Content): they are used when the existing resource has been modified (idempotent).

201 (Created): MUST be used when a new resource is created.

GET: (idempotent)

200 (OK): successful response that includes the entity requested.

DELETE: (idempotent)

200 (OK): for a successful response if the response includes an entity describing the status.

202 (Accepted): if the action has not yet been enacted.

204 (No Content): if the action has been enacted but the response does not include an entity.

3. The content type used in responses is established using the following methodology:

   As a general rule, content type used in response message body must match content type used in request body. In case this is not possible, content type negotiation can be used. The methodology for content type negotiation is based on the "Accept" HTTP header in the request to signal the supported content types. A parameter of name "resFormat" can be given to override the information in this header. The methodology for content type negotiation is specified further in [REST_TS_Common].

   At least XML and JSON content types are supported, with other content types optionally supported on a case-by-case basis to be specifically documented (e.g. simple name-value pair parameters may be accepted in the URL when using GET and application/x-www-form-urlencoded may be supported for the **request** message body when using POST).

4. It is recommended to specify REST API versioning by inserting the API version in the resource URI path (e.g. a 2.0 version is a completely separate set of resources/endpoints from the previous 1.0 version).

   a. Minor API revisions are backwards compatible (in general, unknown parameters should be ignored for forwards compatibility) and major revisions are a distinct set of paths.
   b. If a change is made to the XML request/response format that is not backwards compatible, the major version number must be incremented, otherwise the minor version number is incremented.
   c. The namespace URN of the XML schemas only contains the major version number (e.g. urn:oma:xml:rest:common:1).
   d. The full version number (major and minor version number separated by a "." character) is given in the "version" attribute in the <schema> element of the XML schema.
   e. The resource URI only includes the major version number in the path.
   f. In the case that the API version is not present in the URL path the server will assume that the version is the latest supported by the implementation.

   Example: If service, for example "smsmessaging" supports version 1.0, 1.1, .. and 2.0, 2.1, etc, then the API versioning in the resource URL should be indicated as:
   http://example.com/exampleAPI/1/smsmessaging for the 1.0, 1.1, 1.x version and
   http://example.com/exampleAPI/2/smsmessaging for the 2.0, 2.1, 2.x version of the smsmessaging service.

5. Callback APIs specification and client implementations of the callback APIs have to comply with the remaining set of guidelines in this WP. Wherever necessary, callback functionality (i.e. the ability for the enabler to notify the application of particular events subscribed to) will be supported in the most appropriate manner consistent with the general REST architectural style chosen.

   a. For example, in the case when the client resides in a server-like environment a request URL may be passed by the client on which it can be notified of particular events that the client subscribed to.
   b. In all cases, other approaches may be followed on a case-by-case basis, using an analysis of specific client access particularities.

6. The API specifications should include examples. The example in the REST interface description should avoid using real host and real company name (use "www.example.com" instead of "www.carrier.com" and "myapp.developer.com").

7. If multiple attachments need to be sent as part of the client request or callback request from the server, then MIME Content-Type multipart/related should be used.

8. APIs should support ability to add extra data elements in the request/reply body and extra query parameters in the URL to enhance usability.

   Note: Client and server should ignore unrecognized parameters and data elements for forward compatibility reasons.

9. All URLs in the API specifications are for illustration purposes only. Particular implementation of the API can use different URLs structure and clients have to discover right URLs to use in runtime (no hard coding URLs into the client code) with the exception of the initial (home/starting URLs). This is needed to ensure client portability between different implementations of the API by different vendors. It would also allow server implementation to evolve without requiring clients to adopt new URL structure or hierarchy. Clients are free to cache URLs for the future use according to general HTTP/HTML practices; for a detailed description of the cache mechanism see [RFC 2616]. In other words: they don't have to start from the API home page all the time.

10. If a message contains sensitive data, such as passwords, account numbers, and card numbers (as in account management and payment APIs), security consideration to protect these information is required.

11. The HTTP protocol does not place any a priori limit on the length of a URI according to [RFC2616]. However, some old implementations have a limitation, that is, 256 bytes, while other implementations have at least 4000 characters limitation. GET-based forms with a URI above 255 bytes may get response including 414 (Request-URI Too Long) status code. In the case where the URL would exceed 4000 characters, the API design would consider using POST method instead of GET on a case by case basis.

5.1 API Documentation

Each REST API should be specified in a resource-oriented manner and the resources used by the API should be defined and explained. Use cases and sequence diagrams should be provided. Each REST API specification must include the following definitions:

- API resource definitions, together with an overall structure if multiple resources are defined in the API.
- Definition of HTTP operations (HTTP verbs: GET, POST, PUT, DELETE) for each resource:
- Data type definition, such as complex data type and enumeration type.
  - Description of the operation
  - Request
  - Response
  - Referenced faults

All parameters in URLs must be URL encoded, for example an endUserId and description parameters would be encoded as `endUserId=tel%3A%2B447990123456` and `description=Some%20billing%20information`. They should be listed in examples as unencoded for readability purposes.

5.1.1 API Data Types

REST API data types and enumeration types must be specified with an associated detailed description including optionality. This will enable a developer to understand how to use the parameter. API data type definitions must be consistent and follow recognized standard definitions; the following table gives an example:

Element	Type	Optional	Description

destinationAddress	xsd:anyURI	No	Number associated with the invoked messaging service, i.e. the destination address used by the terminal to send the message.
senderAddress	xsd:anyURI	No	Indicates message senderAddress.
message	xsd:string	No	Text of the message
dateTime	xsd:dateTime	Yes	Time when message was received by operator
resourceURL	xsd:anyURI	Yes	Self referring URL. The resourceURL SHALL NOT be included in POST requests by the client, but MUST be included in POST requests representing notifications by the server to the client, when a complete representation of the resource is embedded in the notification. The resourceURL MUST be also included in responses to any HTTP method that returns an entity body, and in PUT requests**.**
link	common:Link[0..unbounded]	Yes	Links to other resources that are in relationship with the resource
messageId	xsd:string	No	OPTIONAL server-generated message Identifier

Table 1: API Data Type Example

Furthermore, common data types should be reused consistently across multiple APIs.

5.2 Error Handling

After receiving and interpreting a REST request message, a server responds with an HTTP response message, as defined in [RFC2616].

```
Response      = Status-Line
                          *(( general-header
                          | response-header
                          | entity-header ) CRLF)
                          CRLF
                          [ message-body ]
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Standard values for “Status-Code” and “Reason-Phrase” above are used. For all faults additional information should when applicable, be returned to the requestor in the message body. The message body should contain the error details, such as an error code as well as an error description if available. The information returned should be self-contained, so the client does not need to save any state information. For examples provide tables with the supported resource formats.

5.3 Examples

The API specifications should include examples. Examples in the REST interface description should avoid using real host and real company names, for example use "www.example.com" instead of specifics such as "www.carrier.com" or "myapp.developer.com".

Furthermore the REST interface description should include detailed sample Request and Response messages, in HTTP-XML format for the convenience of the reader. For example, a sample REST <GetSmsDeliveryStatusRequest> Request should include:

```
GET /exampleAPI/1/smsmessaging/outbound/tel%3A%2B15555550151/requests/req123/deliveryInfos HTTP/1.1
Accept: application/xml
Host: example.com
```

And the resulting sample REST Response should include:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: nnnn
Date: Thu, 04 Jun 2009 02:51:59 GMT

<?xml version="1.0" encoding="UTF-8"?>
<sms:deliveryInfoList xmlns:sms="urn:oma:xml:rest:sms:1">
   <resourceURL>http://example.com/exampleAPI/1/smsmessaging/outbound/tel%3A%2B15555550151/requests/req123/deliveryInfos
   </resourceURL>
   <deliveryInfo>
     <address>tel:+15555550101</address>
     <deliveryStatus>MessageWaiting</deliveryStatus>
   </deliveryInfo>
   <deliveryInfo>
     <address>tel:+15555550104</address>
     <deliveryStatus>MessageWaiting</deliveryStatus>
   </deliveryInfo>
</sms:deliveryInfoList>
```

5.4 Common Data Formats

5.4.1.1 XML

POST and PUT requests may include data in XML format. An *application/xml* body should be used in these cases. This XML format needs to be compliant with the corresponding XML Schemas for the data types. If the XML contains pointers to the OMA SUP schema files, it can be validated online.

Responses may also include XML body.

5.4.1.2 JSON

POST and PUT requests may include data in JSON format [JSON]. Details on this format can be found in JSON [RFC4627]. Responses may also include bodies in JSON format. In [REST_TS_Common] serialization rules for JSON encoding in HTTP Request/responses are specified.

5.4.1.3 Application/x-www-form-urlencoded

As an alternative to XML or JSON, input data in requests (but not responses) may be submitted in application/x-www-form-urlencoded format as specified in [HTML_FORMS]. Usually, this format is used as the last portion of a URL as defined by [RFC2616]. In ParlayREST, this applies to GET/DELETE requests where this format can be used in query parameters.

In POST requests, this format can also be used, to support the use case of submitting a representation of a data structure directly from HTML forms by a web browser. This will imply the inclusion of an application/x-www-form-urlencoded body. As web browsers use POST to submit these forms, it usually does not make sense to use this format for the body of PUT requests. The format is subject to some restrictions in the character set of the exchanged information – unsafe and reserved characters must be escaped using "percent encoding" [RFC3986]. I.e., a character is replaced by the string as %HH where HH stands for the hexadecimal representation of the ASCII code of the character.

Most ParlayREST specifications define a application/x-www-form-urlencoded representation at least for some POST messages in an Appendix. In case no such message formats are defined in a particular specification, it is recommended to include information about why this has been omitted; otherwise, the following serialization guidelines apply.

5.4.1.3.1 Serialization guidelines for application/x-www-form-urlencoding in Requests.

The following are general rules for mapping between the XML and application/x-www-form-urlencoded formats:

a. When using this serialization in POST requests, data will be included in the body of the request and not in the URL. To do this, Content-Type: application/x-www-form-urlencoded will be used.

b. Where one of the elements is a complex type, only the simple type child sub (or sub-sub)-elements will be included in the URL encoded data.

c. In the absence of XML hierarchy issues, encoding shall look like:

   subelement1=valueA&

   subelement2=valueB&

   attribute=valueC

   The use of application/x-www-form-urlencoded should be specified for each API on a case-by-case basis. This should be documented by means of a table with the result of removing XML hierarchy levels.

Within application/x-www-form-urlencoded bodies, there is neither an indication of the first <?xml version="1.0" encoding="UTF-8" ?> nor the declaration of namespaces or schemaLocations.

5.5 Internationalization

XML Serialization: in REST requests/responses, internationalization comes through the use of UTF-8 encoding in XML bodies. This corresponds with a charset="utf-8".

```
Content-Type: application/xml; charset="utf-8"<?xml version="1.0" encoding="UTF-8"?>
<tns:example>
…..
</tns:example>
```

For JSON serialization, UTF-8 encoding will be used as default, as specified in application/json [RFC4627].

```
Content-Type: application/json;"
Content-Transfer-Encoding: 8bit
<json UTF-8 data>
```

For application/x-www-form-urlencoded serialization, internationalization support is more restricted. According to [RFC1738] and [HTML FORMS]**,** only alphanumeric ASCII characters [0-9, a-z, A-Z] and some other ($-_.+!*'()) may be included directly. Other unsafe and reserved characters may be exchanged too but must escaped (",?, etc.).

This applies to GET/DELETE query parameters and urlencoded bodies in POST/PUT requests, as in the example below.

```
Content-Type: application/x-www-form-urlencoded
message=quedar%EDamos+ma%F1ana&address=621444448
```

For the exchange of **binary data**, base64 will be taken as Content-Transfer-Encoding.

5.6 Backwards Compatibility

APIs evolution should offer backwards compatibility for clients using older versions of the API. Backwards compatibility should be guaranteed for previous upgrades (i.e. minor revisions) within the same release (i.e. major revisions).

Figure 1 Clients using older versions of the API

5.6.1 XML based APIs

In order to support for received API requests, at server's side, the following **guidelines** will be followed in APIs:

a. Data Types – Elements:

- A new version of a data type may be created, including new elements within a XML sequence/choice, but they will be always optional (minOccurs=0).
  - Example: a new element called "wapsupport" is included, but as optional. Former parameters (brand, model) are kept

```
<xsd:complexType name="UserTerminalInfoType">
    <xsd:sequence>
      <xsd:element name="brand" type="xsd:string"/>
      <xsd:element name="model" type="xsd:string"/>
      <xsd:element name="wapsupport" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
```

  - Example: a new, possible third choice is included

```
<xsd:complexType name="AChoiceType">
   <xsd:choice>
     <xsd:element name="choice1" type="xsd:string"/>
     <xsd:element name="choice2" type="xsd:string"/>
    <xsd:element name="choice3" type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>
```

- A new version of a data type may be created, changing the cardinality of some attribute or parameter, but always changing from mandatory to optional, never changing from optional to mandatory.
  - Example: brand and model are now made optional

```
<xsd:complexType name="UserTerminalInfoType">
    <xsd:sequence>
      <xsd:element name="brand" type="xsd:string" minOccurs="0"/>
      <xsd:element name="model" type="xsd:string" minOccurs="0"/>
```

```
  </xsd:sequence>
</xsd:complexType>
```

- New attributes may be defined for REST and SOAP. However, they will always be optional (absence of use=”required”).
  - Example: a new attribute called “lastUpdated” is included but as optional.

```
<xsd:complexType name="UserTerminalInfoType">
    <xsd:sequence>
      <xsd:element name="brand" type="xsd:string" minOccurs="0"/>
      <xsd:element name="model" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
<xsd:attribute name="lastUpdated" type="xsd:string" use="optional"/>
</xsd:complexType>
```

b. Data Types – Enumerations:

- New enumerated values may be included, but always maintaining the former ones.
  - Example: a new value is included in the enumeration (pound), keeping the two other, existing formerly.

```
 <xsd:simpleType name="CurrencyType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="euro"/>
      <xsd:enumeration value="dollar"/>
      <xsd:enumeration value="pound"/>
    </xsd:restriction>
</xsd:simpleType>
```

c. Operations:

- Operations may be evolved, adding new parameters. But new parameters will always be optional. Existing parameters will always be kept, for compatibility.
  - Example: over an existing operation, a new, optional input parameter is included, “maxItems”.

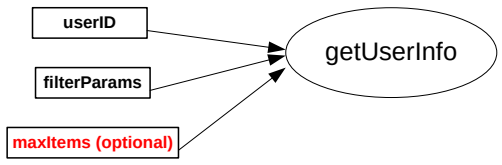


Figure 2 Backwards compatibility for operations

- New operations may be added, but existing operations will always be kept, for compatibility.

5.6.2 JSON based APIs

Above considerations are given for XML based API requests. For the JSON case, existing parameters in previous versions of the API will be kept in API specifications, for backwards compatibility.

5.7 Forward Compatibility, Extensibility

APIs should be designed to offer forwards compatibility towards new versions of the API. This compatibility will typically apply between upgrades under a same Release, in two ways:

- Upgraded servers returning a response to legacy clients
- New versions of clients making requests to existing, non upgraded API servers

Figure 3 Upgraded servers returning a response to legacy clients

Figure 4 New versions of clients making requests to existing, non-upgraded servers

5.7.1 XML based APIs

Considering XML format, extensibility and evolution of the data exchanged over APIs is possible by means of **extensible XML Schemas.**

Thus, the following technical guidelines will be followed for the **design of extensible APIs**:

a. Extensions in sequences.

- It is recommended to include extensibility points in root XML types or any other which is expected to evolve in the future, with “processContents=lax” processing model, so that receivers are not forced to validate these extended elements. Extensions over the same namespace will go under a wrapper and extensions over other namespaces may go directly under parent data. However, if the elements included belong to a known namespace, server will try to parse these XML elements.
    - Example

```
<s:complexType name="ExtensionType">
 <s:sequence>
    <s:any processContents="lax" minOccurs="0" maxOccurs="unbounded" namespace="##any"/>
 </s:sequence>
 <s:anyAttribute/>
</s:complexType>
```

And then, in the complex element definitions include also a direct wildcard to include directly additional elements from other namespaces (to avoid XML determinism problems):

```
<xsd:complexType name="MyType">
```

```
  <xsd:sequence>
    <xsd:element name="e1" type="xsd:string"/>
    <xsd:element name="e2" type="xsd:string"/>
                    <xsd:element name="Extension" type="tns:ExtensionType"
                    minOccurs="0" maxOccurs="unbounded"/>
    <xsd:any namespace="##other" processContents="lax" minOccurs="0"   maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:anyAttribute/>
</xsd:complexType>
```

Note: “choice” and “all” complex data types are not extensible and thus can not be modified within a release.

b. Extension of attributes:

- The possibility of any future attribute is given by means of the inclusion of the anyAttribute wildcard, as indicated in the example above.

c. Extension of enumerations

- The possibility of any future value in the enumeration is given by means the definition of the enumeration as a union of the current enumerated values plus a possible string.
  - Example:

```
<xsd:simpleType name="DeliveryStatusType">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="DeliveredToTerminal"/>
         <xsd:enumeration value="DeliveryImpossible"/>
      </xsd:restriction>
</xsd:simpleType>

<xsd:element name="deliveryStatus">
       <xsd:simpleType>
           <xsd:union memberTypes="tns:DeliveryStatusTypes xsd:string"/>
       </xsd:simpleType>
</xsd:element>
```

This procedure consists in the “Must Ignore” rule, in which receivers may omit extended elements which they don’t understand in syntactically correct XML documents (typically, the validation should then be performed by the application logic on top of the API).

Note: Whether to follow this Must Ignore Rule - along with the extensibility mechanisms above - or not is a design decision that must consider deployment dependent aspects as well as the specific usage that is desired for the API itself.

5.7.2 JSON based APIs

Above considerations are given for XML based API requests. However, JSON is an inherently extensible serialization format. As a string, any data may be additionally included, although if server is not upgraded they will merely be ignored

5.8 Encoding and Serialization Details for MIME format

A MIME multipart message is used in some ParlayREST APIs (e.g. the MMS API) to represent content that consists of several parts. ParlayREST suggests for simplicity purposes and better suitability to the internet developer community and browsers to use multipart/form-data [RFC2388] and [HTML_FORMS].

A MIME multipart message contains the root structure which carries the ParlayREST parameter set, and one or more multimedia content attachments expressed as MIME body parts within the HTTP request or response.

Rules how to compose a multipart message are given in [REST_TS_Common]. Messages with one attachment are represented differently than those with multiple attachments.

The following uses the example of MMS to illustrate how to encode and send a MIME multipart message using forms when the root fields are represented in XML and when more than one MMS content is sent:

```
POST http://{serverRoot}/{apiVersion}/messaging/{senderAddress}/outbound/requests HTTP/1.1

Other http headers
Content-Type: multipart/form-data, boundary=asdfa487

--asdfa487
Content-Disposition: form-data; name="root-fields"
Content-type: application/xml

Here the XML representation of the MMS root fields "Inbound/OutboundMMSMessage"

--asdfa487
Content-disposition: multipart/form-data; name="attachments"
Content-type: multipart/mixed, boundary=BbC04y

    --BbC04y
    Content-disposition: attachment; filename="textBody.txt"
    Content-Type: text/plain; charset= "UTF-8"
    Content-Transfer-Encoding: 8-bit


    ... text of the MMS ...
    --BbC04y
    Content-disposition: attachment; filename="file2.gif"
    Content-type: image/gif
    Content-Transfer-Encoding: base64

      ...contents of file2.gif...

        --BbC04y

        Other attachment may come here (correctly delimited by the boundary string)

         --BbC04y--


--asdfa487--
```

The following uses the example of MMS to illustrate how to encode and send a MIME multipart message using forms when the root fields are represented in JSON, and when a single content is sent:

```
POST http://{serverRoot}/{apiVersion}/messaging /{senderAddress}/outbound/requests HTTP/1.1

Other http headers

Content-Type: multipart/form-data, boundary=asdfa487
```

--asdfa487
Content-Disposition: multipart/form-data; name="root-fields"
Content-type: application/json

Here the JSON representation of the MMS root fields "Inbound/OutboundMMSMessage"

--asdfa487
Content-disposition: multipart/form-data; name="attachments"; filename="picture.jpeg"
Content-type: image/jpeg

...contents of picture.jpeg...

--asdfa487--

5.9 Light-weight resources

A term called "light-weight resources" is used to describe resources that enable access to a part of a data structure or individual elements in a data structure. This is in a contrast to other resources that operate on the entire data structure and are regarded as heavy-weight resources. A data structure could be any kind of XML/JSON structure representing the heavy-weight resource that is created using POST or PUT. A light-weight resource is basically a URL pointing out a resource representing a sub-structure inside the data structure.

For light-weight resources the following apply.

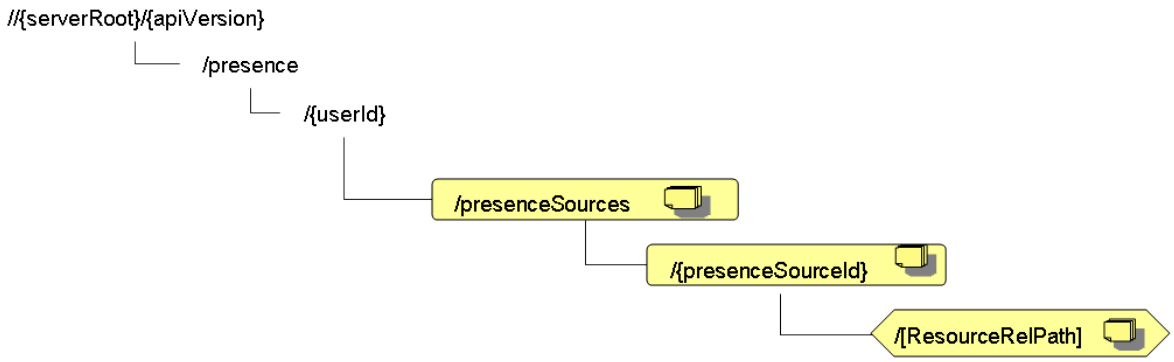
- Only PUT, GET and DELETE operations can be used (PUT will create the resource if it does not exists).
- Precondition for using light-weight resources is that the ancestor heavy-weight resource exists.
- There may be several levels of Light-weight resources below the ancestor heavy-weight resource, depending on the data structure (i.e. ../parent/child/grandchild)
- The entire light-weight resource URL is built up of the heavy-weight URL path and the relative resource path for light-weight resource.
- HTTP Etag value MAY be reused from the ancestor heavy-weight resource. Applications MAY also assign individual ETag values per light-weight resource.

The following text (steps 1-3) describes how light-weight resources should be described in API technical specifications.

1. Resources Summary (Section 5.1 in TS):

The light-weight resources should be illustrated in the resource tree as [ResourceRelPath] (i.e. a relative path of the resource) surrounded by a hexagon shape. The heavy-weight resources are illustrated by using a rectangular shape.

Example:



The resource table should describe the name of the light-weight resource as well as applicable operations:

Example:

Resource	**Base URL: http://{serverRoot}/{apiVersion}/xyz**	**Data Structures**	**HTTP verbs**			
			GET	**PUT**	**POST**	**DELETE**
Management of individual data of some kind	/{userId}/Heavy-weight-resource path/[ResourceRelPath]	*The data structure corresponds to the element pointed out by the request-URI.* (Used for GET/PUT)	Retrieves the value of the specified data item	Updates a data item	no	Removes data item.

2. Data Structure (Section 5.2.x in TS):

Data structures that contain elements that could be accessed by using light-weight resources should include a column called [ResourceRelPath]. This column includes string(s) and each of these strings represents a resource relative path for light-weight resource that needs to be appended to the corresponding heavy-weight resource URL in order to create light-weight resource URL for accessing corresponding element in the data structure. The root element and data type of the resource associated with the [ResourceRelPath] are defined by the Element and Type columns in the row that defines the [ResourceRelPath].

Example 1:

5.2.x Type:Presence

Element	Type	Optional	[ResourceRelPath]	Description
person	PersonAttributes	Yes	person	The presence attributes related to person.

5.2.x Type:PersonAttributes

Element	Type	Optional	[ResourceRelPath]	Description
mood	Mood	Yes	person/mood	The user's mood (angry, confused, happy, etc.) [RFC4480]

root element when accessed as light-weight resource

light-weight resource

Sub-elements in a data structure that are used to identify a particular instance of the parent element are regarded as key properties (keys) of the element; for example for element service, key properties are service Id and service version. In case the key(s) are used to identify a light-weight resource representation, it should be indicated in both the [ResourceRelPath] and in the description part of the corresponding data structure(s) In addition, for keys: the following apply:

- Keys are not accessible individually using light-weight resources ([ResourceRelPath] column should indicate "Not applicable")
- When assessing a parent element with light-weight resource, the key(s) shall not be altered (this should be stated in the description column of the corresponding data structure(s).
- Where applicable, keys in [ResourceRelPath] should be surrounded by curly brackets ({..}).

Example 2:

5.2.x Type: Presence

Element	Type	Optional	[ResourceRelPath]	Description
service	ServiceAttributes [0..unbounded]	Yes	service/{serviceId}/{version}	The presence attributes related to services. For description of "serviceId" and "version" see 5.2.y. The sub-elements "serviceId" and "version" of the type ServiceAttributes are key properties for service element and SHALL NOT be altered when this element is accessed as a light-weight resource.

5.2.z Type: ServiceAttributes

Element	Type	Optional	[ResourceRelPath]	Description
serviceId	xsd:token	No	Not applicable	Identifier of the service. It is a key property of the service.
version	xsd:token	No	Not applicable	The version of the specified service. It is a key property of the service.
statusIcon	StatusIcon	Yes	service/{serviceId}/{version}/statusIcon	Contains a link to an icon of the user. [RFC4480]

key properties

3. Detailed resource operation description (starts with Section 5.4 until 5.X in TS)

Typically a URL for light-weight resource should look like

http://{Heavy-weight resource path}/[ResourceRelPath

Table 5.X.1 Request URI variables, should include description for [[ResourceRelPath]

Example:

The following request URI variables are common for all HTTP commands:

Name	Description
serverRoot	server base url: hostname+port+base path. Example: http://example.com/exampleAPI
apiVersion	version of the ParlayREST API clients want to use (e.g. 1 for version 1.x)
xyz	Some data...
[ResourceRelPath]	Relative resource path for a light-weight resource, consisting of a relative path down to an element in the data structure. For more information about the applicable values (strings) for this variable see 5.X.1.1

The description part of [ResourceRelPath] refers to another section-table (5.X.1.1) that is specific for light-weight resources and that should to be created too. The table should describe what types of light-weight resources can be accessed by that particular heavy-weight resource, what methods are available, and the link to the data structure (section 5.2.X) that contain possible strings (relative resource paths) that could be used for [ResourceRelPath].

Example:

5.X.1.1 Light-weight relative resource paths

The following table describes the types of light-weight resources that can be accessed by using this resource, applicable methods, and links to data structures that contain values (strings) for those relative resource paths.

.

Light-weight resource type	Method supported	Description
A type of the light-weight resource	GET, PUT, DELETE	Description of the type of light-weight resource that can be accessed. Here also shall be included a reference to the section with the Data Structure where such light-weight resource type is specified. The data structure in the column [ResourceRelPath] contains values (strings) for relative resource path [ResourceRelPath].

Appendix A. Change History (Informative)

Document Identifier	Date	Sections	Description
OMA-WP-Guidelines_for_REST_API_specifications-20100818-D	18 Aug 2010	All	Generalized REST White Paper, created from version 1_0 WP
OMA-WP-Guidelines_for_REST_API_specifications-20100827-D	27 Aug 2010	3, 5	Implement CR OMA-ARC-REST-2010-0420
OMA-WP-Guidelines_for_REST_API_specifications-20101026-D	26 Oct 2010		Implement the following CRs • OMA-ARC-REST-2010-0423R02 • OMA-ARC-REST-2010-0582 Address CONRR comments K002, K005, K006, and K008 Make some minor clerical corrections.
OMA-WP-Guidelines_for_REST_API_specifications-20101208-D	08 Dec 2010	5.9	Implement the following CRs • OMA-ARC-REST-2010-0715R01 • OMA-ARC-REST-2010-0718
OMA-WP-Guidelines_for_REST_API_specifications-20101214-D	14 Dec 2010	All	Editorial fixes: heading styles, Cover page and history table
OMA-WP-Guidelines_for_REST_API_specifications-20110111-C	11 Jan 2011	All	Status changed to Candidate by TP: OMA-TP-2010-0531R01-INP_ParlayREST_2_0_for_Candidate_approval
OMA-WP-Guidelines_for_REST_API_specifications-20110808-D	08 Aug 2011	5.9	Incorporated CR OMA-ARC-REST-M-2011-0020-CR_LW_resource_keys_as_elements_WP_ParlayREST2
OMA-WP-Guidelines_for_REST_API_specifications-20120625-D	25 Jun 2012	1, 2, 3.2, 3.3, 4, 5, 5.1, 5.1.1, 5.2, 5.3, 5.4.1.3, 5.4.1.3.1, 5.5, 5.6, 5.6.1, 5.7.1, 5.8, 5.9	Incorporated CR OMA-ARC-REST-M-2012-0010-CR_WP_Guidelines_fixing_validation_errors Editorial changes
OMA-WP-Guidelines_for_REST_API_specifications-20120724-A	24 Jul 2012	All	Status changed to Approved by TP Ref TP Doc# OMA-TP-2012-0280-INP_ParlayREST_2_0_for_Final_Approval