



White Paper on Guidelines for REST API specifications in OMA

Candidate – 23 Nov 2010

Open Mobile Alliance
OMA-WP-Guidelines_for_REST_API_specifications-20101123-C

Use of this document is subject to all of the terms and conditions of the Use Agreement located at <http://www.openmobilealliance.org/UseAgreement.html>.

Unless this document is clearly designated as an approved specification, this document is a work in process, is not an approved Open Mobile Alliance™ specification, and is subject to revision or removal without notice.

You may use this document or any part of the document for internal or educational purposes only, provided you do not modify, edit or take out of context the information in this document in any manner. Information contained in this document may be used, at your sole risk, for any purposes. You may not use this document in any other manner without the prior written permission of the Open Mobile Alliance. The Open Mobile Alliance authorizes you to copy this document, provided that you retain all copyright and other proprietary notices contained in the original materials on any copies of the materials and that you comply strictly with these terms. This copyright permission does not constitute an endorsement of the products or services. The Open Mobile Alliance assumes no responsibility for errors or omissions in this document.

Each Open Mobile Alliance member has agreed to use reasonable endeavors to inform the Open Mobile Alliance in a timely manner of Essential IPR as it becomes aware that the Essential IPR is related to the prepared or published specification. However, the members do not have an obligation to conduct IPR searches. The declared Essential IPR is publicly available to members and non-members of the Open Mobile Alliance and may be found on the “OMA IPR Declarations” list at <http://www.openmobilealliance.org/ipr.html>. The Open Mobile Alliance has not conducted an independent IPR review of this document and the information contained herein, and makes no representations or warranties regarding third party IPR, including without limitation patents, copyrights or trade secret rights. This document may contain inventions for which you must obtain licenses from third parties before making, using or selling the inventions. Defined terms above are set forth in the schedule to the Open Mobile Alliance Application Form.

NO REPRESENTATIONS OR WARRANTIES (WHETHER EXPRESS OR IMPLIED) ARE MADE BY THE OPEN MOBILE ALLIANCE OR ANY OPEN MOBILE ALLIANCE MEMBER OR ITS AFFILIATES REGARDING ANY OF THE IPR'S REPRESENTED ON THE “OMA IPR DECLARATIONS” LIST, INCLUDING, BUT NOT LIMITED TO THE ACCURACY, COMPLETENESS, VALIDITY OR RELEVANCE OF THE INFORMATION OR WHETHER OR NOT SUCH RIGHTS ARE ESSENTIAL OR NON-ESSENTIAL.

THE OPEN MOBILE ALLIANCE IS NOT LIABLE FOR AND HEREBY DISCLAIMS ANY DIRECT, INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR EXEMPLARY DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE USE OF DOCUMENTS AND THE INFORMATION CONTAINED IN THE DOCUMENTS.

© 2010 Open Mobile Alliance Ltd. All Rights Reserved.

Used with the permission of the Open Mobile Alliance Ltd. under the terms set forth above.

Contents

1. SCOPE	4
2. REFERENCES	5
3. TERMINOLOGY AND CONVENTIONS	6
3.1 CONVENTIONS	6
3.2 DEFINITIONS.....	6
3.3 ABBREVIATIONS	6
4. INTRODUCTION	7
5. PRINCIPLES FOR DEFINING REST APIS IN OMA	8
SomeExampleType.....	10
5.1 API DOCUMENTATION.....	10
5.1.1 API Data Types.....	11
5.2 ERROR HANDLING	11
5.3 EXAMPLES	12
5.3.1 Common Data Formats	13
5.4 INTERNATIONALIZATION	14
5.5 BACKWARDS COMPATIBILITY	14
5.5.1 JSON based APIs.....	16
5.6 FORWARD COMPATIBILITY, EXTENSIBILITY.....	16
5.6.1 JSON based APIs.....	18
5.7 ENCODING AND SERIALIZATION DETAILS FOR MIME FORMAT.....	18
5.8 LIGHT-WEIGHT RESOURCES	20
APPENDIX A. CHANGE HISTORY (INFORMATIVE).....	25

Figures

Figure 1 Clients using older versions of the API.....	14
Figure 2 Backwards compatibility for operations.....	16
Figure 3 Upgraded servers returning a response to legacy clients	16
Figure 4 New versions of clients making requests to existing, non-upgraded servers	16

Tables

Table 1: API Data Type Example.....	11
-------------------------------------	----

1. Scope

This white paper contains guidelines for REST API specifications developed 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 2.7, Open Mobile Alliance™, OMA-ORG-Dictionary-V2_7, 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

[N/A] [N/A]

3.3 Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
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
TS	Technical Specification
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

4. Introduction

This document is the OMA Guidelines for REST API specification development and 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 [RFC 2616] 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 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:xsd: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 X supports version 1.0, 1.1, .. and 2.0, 2.1, etc, then you use:

http://example.com/service/1/smsservice for the 1.0, 1.1, 1.x version and

http://example.com/service/2/smsservice for the 2.0, 2.1, 2.x version of the smsservice.

5. Callback APIs specification and client implementations of the callback APIs have to comply with the remaining set of guidelines in this cookbook. 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. It is strongly advised to include a self reference element in the representation of the resource, in all cases where the resource is defined by the server. See below an example:

SomeExampleType

Element	Type	Optional	Description
...
resourceURL	xsd:anyURI	Yes	Self referring URL. SHALL NOT be included in POST requests, MUST be included in responses to any HTTP method that returns an entity body, and in PUT requests.
...

9. 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.

10. 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.
11. 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.
12. 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 Message 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
link	common:Link[0..unbounded]	Yes	Links to other resources that are in relationship with the resource
id	xsd:string	No	Server generated messageId entifier

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 Status-Code and Reason phrase 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 .../{apiVersion}/smsmessaging/{senderAddress}/outbound/requests/{requestId}
HTTP/1.1
Accept: application/xml
Host: www.example.com:80
```

And the resulting sample REST Response should include:

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

<?xml version="1.0" encoding="UTF-8"?>
<MessageReferences>
  <OutboundMessageReference
    <address>tel:1350000001</address>
    <id>{requestId1}</id>

    <resourceURL>http://{serverRoot}/{apiVersion}/smsmessaging/{senderAddress}/outbound
    /requests/{requestId1}</resourceURL>
    <DeliveryInfos>
      <DeliveryInfo>
        <DeliveryStatus>DeliveredToNetwork</DeliveryStatus>
        <address>tel:+1350000001</address>
      </DeliveryInfo>
    </DeliveryInfos>
  </OutboundMessageReference>
  <OutboundMessageReference>
    <address>tel:+1350000991</address>
    <address>tel:+1350000992</address>
    <id>{requestId2}</id>

    <resourceURL>http://{serverRoot}/{apiVersion}/smsmessaging/{senderAddress}/outbound
    /requests/{requestId2}</resourceURL>
    <DeliveryInfos>
      <DeliveryInfo>
        <DeliveryStatus>DeliveredToNetwork</DeliveryStatus>
        <address>tel:+1350000001</address>
      </DeliveryInfo>
      <DeliveryInfo>
        <DeliveryStatus>DeliveredToTerminal</DeliveryStatus>
        <address>tel:+1350000001</address>
      </DeliveryInfo>
    </DeliveryInfos>
  </OutboundMessageReference>
  ...
</MessageReferences>
```

5.3.1 Common Data Formats

5.3.1.1 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.

5.3.1.2 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.

5.3.1.3 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 *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.

5.3.1.3.1 Serialization guidelines for www-form-urlencoded 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 *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.

If none of the above applies, the use of *www-form-urlencoded* is not recommended.

Within *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.

5.4 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 too as default, as specified in application/json [RFC4627]. Thus, Content-Transfer-Encoding 8bit must be used with this media type.

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

For form-urlencoding 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%Flana&address=621444448
```

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

5.5 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 a same Release (i.e. major revisions).



Figure 1 Clients using older versions of the API

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>
```

```
</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”.

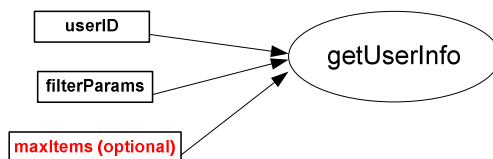


Figure 2 Backwards compatibility for operations

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

5.5.1 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.

5.6 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

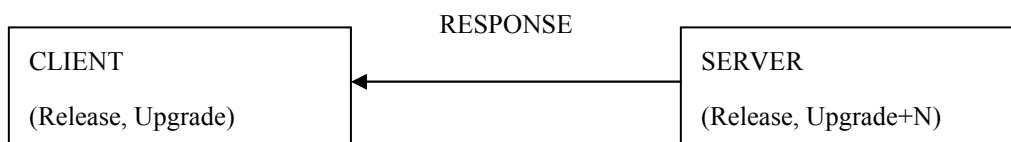


Figure 3 Upgraded servers returning a response to legacy clients



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

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.

5.6.1 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

5.7 Encoding and Serialization Details for MIME format

A MIME multipart message (used for instance in the MMS API) consists of several parts:

- The root structure (e.g. InboundMMSMessage or OutboundMMSMessage), expressed in the different possible formats (xml, json or even in url-encoded format for requests). This part conveys the origin, destination addresses, subject, priority, message identifier, etc.
- The multimedia contents or attachments expressed in the form of links or as MIME body parts, within the HTTP request or response. They include all contents, both plain text as well as other MIME types (images, videos, etc), potentially exchangeable in MMSs.

In REST APIs, for simplicity purposes and better suitability to the internet developer community and browsers, multipart/form-data will be used instead [RFC2388] and [HTML FORMS]. This implies that:

1. **Root fields** as described above will be included as a single form field with a MIME body with:

Content-Disposition: form-data; name="root-fields"

Content-Type: <Corresponding Content type>

Allowed encodings for the root fields are:

- application/xml
- application/json
- application/x-www-form-urlencoded

2. Multimedia contents (text, images, etc.) will be included using one of the following two options:

- a. When the message contains **only one content**: By including a MIME body with:

Content-Disposition: form-data; name="attachments", filename="<Name of the message content>"

Content-Type:<Corresponding Content-Type>

- b. When the message contains more than one content: By including a form-field with a MIME body with:

Content-Disposition: form-data; name="attachments"

Content-Type: multipart/mixed

Then, every one of the possible message contents will be included as subparts, with:

Content-Disposition:attachment; filename="<Name of the message content>" Content-Type:
<Corresponding Content-Type>

3. For every HTTP body part and subparts, it is possible to include other parameters (Content-Description, Content-Transfer-Encoding), etc.

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://{server root}/{api version}/messaging /{sender address}/outbound/requests

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: form-data; name="attachments"

Content-type: multipart/mixed, boundary=BbCo4y

--BbCo4y

Content-disposition: attachment; filename="textBody.txt"

Content-Type: text/plain; charset= "UTF-8"

Content-Transfer-Encoding: 8-bit

... text of the MMS ...

--BbCo4y

Content-disposition: attachment; filename="file2.gif"

Content-type: image/gif

Content-Transfer-Encoding: base64

...contents of file2.gif...

--BbCo4y

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

--BbCo4y--

--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://{server root}/{api version}/messaging /{sender address}/outbound/requests

Other http headers

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

--asdfa487

Content-Disposition: form-data; name="root-fields"

Content-type: application/json

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

--asdfa487

Content-disposition: form-data; name="attachments"; filename="picture.jpeg"

Content-type: image/jpeg

...contents of picture.jpeg...

--asdfa487—

5.8 Light-weight resources

A term called "light-weight resources" is used to describe resources that enable access to a part of a document or individual elements in a data structure. This is in a contrast to other resources that operate on the entire document or entire data structure and are regarded as heavy-weight resources. A document 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 a document.

For light-weight resources the following apply.

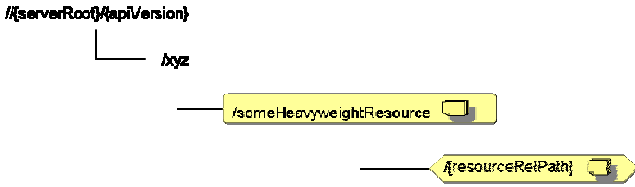
- Only PUT, GET and DELETE operations can be used (PUT will create the resource if it does not exists).
- Precondition for using light-weight resources is that the ancestor heavy-weight resource exists.
- There may be several levels of Light-weight resources below the ancestor heavy-weight resource, 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):

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

Example:



The Resource table should describe the name of the resource as well as applicable operations:

Example:

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

2. Data Structure (Section 5.2.x):

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].

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”)
- A key should be modeled in XML as an attribute to the parent element
- 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/{serviceld}/{version}	The presence attributes related to services. For description of "serviceld" and "version" see 5.2.y. The sub-elements "serviceld" 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
serviceld	xsd:token	No	Not applicable	Identifier of the service. It is a key property of the service. It is defined as an attribute in XML format.
version	xsd:token	No	Not applicable	The version of the specified service. It is a key property of the service. It is defined as an attribute in XML format.
statusIcon	StatusIcon	Yes	service/{serviceld}/{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)

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:80/ParlayREST

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 be created too. The table should describe what types of light-weight resources can be accessed by that particular light-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-ParlayREST-API-specifications-yymmdd-D	06 Nov 2009	All	Initial version of WP as permanent doc
	16 Nov 2009	5.1, 5.2, 5.3	AGREED contributions from OMA ARC conference call: November 10th, 2009 OMA-ARC-REST-2009-0053-INP_merged_Exception_for_cookbook OMA-ARC-REST-2009-0056-INP_merged_Example_for_cookbook OMA-ARC-REST-2009-0057-INP_merged_API_Documentation_for_cookbook
	19 Nov 2009	2, 4	
	01 Dec 2009		Implement agreements during the Vienna Interim: OMA-ARC-REST-2009-0078 OMA-ARC-REST-2009-0101R01
	02 Dec 2009		Implement agreements during the Vienna Interim: OMA-ARC-REST-2009-0099R01
	03 Dec 2009		Implement agreements during the Vienna Interim: OMA-ARC-REST-2009-0116 OMA-ARC-REST-2009-0117
	04 Dec 2009		Implement agreements during the Vienna Interim: OMA-ARC-REST-2009-0123
	02 Feb 2010		Implement OMA-ARC-REST-2010-0012R01
	03 Feb 2010		Implement OMA-ARC-REST-2010-0044
	04 Feb 2010		Implement OMA-ARC-REST-2010-0045 and OMA-ARC-REST-2010-0049R01
	05 Feb 2010		Clerical cleanup
	12 Feb 2010	5.3.1.2	Implement OMA-ARC-REST-2010-0061
	13 Feb 2010		Clerical cleanup, formatting, spelling mistakes
	19 Feb 2010	5.5, 5.6	Implement OMA-ARC-REST-2010-0011R02
	05 Mar 2010	All	Clerical clean-up
11 Mar 2010	All	Implement errors found during the Walk-through of March 11 (see meeting minutes for details – OMA-ARC-REST-2010-0119)	
OMA-WP-Guidelines_for_ParlayREST_API_specifications-20100427-C	27 Apr 2010	All	Status changed to Candidate by TP: OMA-TP-2010-0186-INP_ParlayREST_V1_0_ERP_for_Candidate_Approval
OMA-WP-Guidelines_for_ParlayREST_API_specifications-20100924-D	24 Sep 2010	All	Implemented agreed changes: OMA-ARC-REST-2010-0423R02-CR_WP_update_for_light_weight_resources OMA-ARC-REST-2010-0447R01-CR_Guidelines_regarding_self_reference
OMA-WP-Guidelines_for_ParlayREST_API_specifications-20101123-C	23 Nov 2010	All	Status changed to Candidate by TP: OMA-TP-2010-0463R01-INP_ParlayREST_V1_0_ERP_for_Candidate_reapproval