Common definitions for RESTful Network APIs

Approved Version 1.0 – 16 Jan 2018

Open Mobile Alliance

OMA-TS-REST_NetAPI_Common-V1_0-20180116-A

Contents

Figures

Tables

1. Scope

The scope of this specification is to provide common definitions and specification material for RESTful Network APIs in OMA.

2. References

2.1 Normative References

[Autho4API_10]	"Authorization Framework for Network APIs", Open Mobile Alliance™, OMA-ER-Autho4API-V1_0, URL: http://www.openmobilealliance.org/
[HTML FORMS]	"HTML Forms", W3C Recommendation, URL:http://www.w3.org/TR/html401/interact/forms.html
[ISO4217]	"ISO 4217 currency names and code elements", URL: http://www.iso.org/
[MMS CONF]	"MMS Conformance Document", Open Mobile Alliance™, OMA-TS-MMS-CONF-V1_3-20110913-A, URL : http://www.openmobilealliance.org
[OMNA_Autho4API]	Open Mobile Naming Authority "Autho4API Scope Values Registry", Open Mobile Alliance™, URL: http://www.openmobilealliance.org/tech/omna NOTE: The hyperlink above will point directly to the OMNA registry page once available.
[ParlayX_Common]	"Open Service Access (OSA); Parlay X web services; Part 1: Common", 3GPP TS 29.199-01, Release 8, Third Generation Partnership Project, URL: http://www.3gpp.org/ftp/Specs/html-info/29-series.htm
[PSA]	"Reference Release Package for Parlay Service Access", Open Mobile Alliance™, OMA-ERP-PSA-V1_0, URL: http://www.openmobilealliance.org/
[REST_NetAPI_ACR]	"RESTful Network API for Anonymous Customer Reference Management", Open Mobile Alliance™, OMA-TS-REST_NetAPI_ACR-V1_0, URL: http://www.openmobilealliance.org/
[REST_NetAPI_CallNotif]	"RESTful Network API for CallNotification", Open Mobile Alliance™, OMA-TS-REST_NetAPI_CallNotification-V1_0, URL: http://www.openmobilealliance.org/
[REST_NetAPI_Notif_Channel]	"RESTful Network API for Notification Channel", Open Mobile Alliance™, OMA-TS-REST_NetAPI_NotificationChannel-V1_0, URL: http://www.openmobilealliance.org/
[RFC2046]	"Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", N. Freed, N. Borenstein, November 1996, URL: http://tools.ietf.org/html/rfc2046
[RFC2119]	"Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997, URL: http://www.ietf.org/rfc/rfc2119.txt
[RFC2183]	"Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field", R. Troost, S. Dorner, K. Moore, August 1997, URL: http://tools.ietf.org/html/rfc2183
[RFC2231]	"MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations", N. Freed, K. Moore, November 1997, URL: http://tools.ietf.org/html/rfc2231
[RFC2387]	"The MIME Multipart/Related Content-type", E. Levinson, August 1998, URL: http://www.ietf.org/rfc/rfc2387.txt
[RFC2388]	"Returning Values from Forms: multipart/form-data", L. Masinter, August, 1998, URL:http://www.ietf.org/rfc/rfc2388.txt
[RFC2818]	"HTTP Over TLS", E. Rescorla, May 2000, URL: http://www.ietf.org/rfc/rfc2818.txt
[RFC3261]	"SIP: Session Initiation Protocol", J. Rosenberg, et. Al, June 2002, URL: http://www.ietf.org/rfc/rfc3261.txt
[RFC3966]	"The tel URI for Telephone Numbers", H. Schulzrinne, December 2004, URL: http://www.ietf.org/rfc/rfc3966.txt
[RFC3986]	"Uniform Resource Identifier (URI): Generic Syntax", T. Berners-Lee, R. Fielding, L. Masinter, January 2005, URL: http://www.ietf.org/rfc/rfc3986.txt
[RFC4122]	"A Universally Unique IDentifier (UUID) URN Namespace", P. Leach, M. Mealling, R. Salz, July 2005, URL: http://www.ietf.org/rfc/rfc4122.txt
[RFC6585]	"Additional HTTP status codes", M. Nottingham, R.Fielding, April 2012, URL: http://www.ietf.org/rfc/rfc6585.txt
[RFC7159]	"The JavaScript Object Notation (JSON) Data Interchange Format", T. Bray, Ed., March 2014, URL:https://tools.ietf.org/html/rfc7159

[RFC7231]	"Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", R. Fielding, J. Reschke, June 2014, URL: http://tools.ietf.org/html/rfc7231
[RFC7232]	"Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", R. Fielding, J. Reschke, June 2014, URL: http://tools.ietf.org/html/rfc7232
[RFC7235]	"Hypertext Transfer Protocol (HTTP/1.1): Authentication", R. Fielding, J. Reschke, June 2014, URL: http://tools.ietf.org/html/rfc7235
[SEC_CF-V1_1]	"Security Common Functions ", Version 1.1, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/
[W3C_URLENC]	HTML 4.01 Specification, Section 17.13.4 Form content types, The World Wide Web Consortium, URL: http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.1
[W3C-XML11]	W3C XML 1.1 Specification, URL: http://www.w3.org/TR/xml11/
[XMLSchema1]	W3C Recommendation, XML Schema Part 1: Structures Second Edition, URL: http://www.w3.org/TR/xmlschema-1/
[XMLSchema2]	W3C Recommendation, XML Schema Part 2: Datatypes Second Edition, URL: http://www.w3.org/TR/xmlschema-2/

2.2 Informative References

[OMA_PUSH]	"Push Access Protocol Specification". Open Mobile Alliance™. OMA-WAP-TS-PAP-V2_3 URL:http://www.openmobilealliance.org/
[OMA_REST_Common]	"Common definitions and specifications for OMA REST interfaces", Open Mobile Alliance™, OMA-TS-REST_Common-V1_0, URL: http://www.openmobilealliance.org/
[OMADICT]	"Dictionary for OMA Specifications", Version 2.8, Open Mobile Alliance™, OMA-ORG-Dictionary-V2_8, URL: http://www.openmobilealliance.org/
[ParlayREST_20]	"RESTful bindings for Parlay X Web Services – Enabler Release Package", Open Mobile Alliance™, OMA-ERP-ParlayREST_Common-V2_0, URL: http://www.openmobilealliance.org/
[ParlayREST_Common]	"RESTful bindings for Parlay X Web Services - Common", Open Mobile Alliance™, OMA-TS-ParlayREST_Common-V1_1, URL: http://www.openmobilealliance.org/
[REST_NetAPI_WP]	"Guidelines for RESTful Network APIs", Open Mobile Alliance™, OMA-WP-Guidelines_for_RESTful_Network_APIs, URL:http://www.openmobilealliance.org/
[XML2JSON]	Open source "UNICA" XML to JSON conversion tool URL: http://forge.morfeo-project.org/projects/unicaxml2json/

3. Terminology and Conventions

3.1 Conventions

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].

All sections and appendixes, except "Scope" and "Introduction", are normative, unless they are explicitly indicated to be informative.

3.2 Definitions

For the purpose of this TS, all definitions from the OMA Dictionary apply [OMADICT].

Client-side Notification URL	An HTTP URL exposed by a client, on which it is capable of receiving notifications and that can be used by the client when subscribing to notifications.
Heavy-weight Resource	A resource which is identified by a resource URL which is then used by HTTP methods to operate on the entire data structure representing the resource.
Instance-based JSON generation	An approach to generate JSON that considers information in the actual instance of a resource representation. For example, such approach considers an XML representation of a resource, without considering the related XML schema information.
Light-weight Resource	A subordinate resource of a Heavy-weight Resource which is identified by its own resource URL which is then used by HTTP methods to operate on a part of the data structure representing the Heavy-weight Resource. The Light-weight Resource URL can be seen as an extension of the Heavy-weight Resource URL. There could be several levels of Light-weight Resources below the ancestor Heavy-weight Resource, depending on the data structure.
Long Polling	A variation of the traditional polling technique, where the server does not reply to a request unless a particular event, status or timeout has occurred. Once the server has sent a response, it closes the connection, and typically the client immediately sends a new request. This allows the emulation of an information push from a server to a client.
Notification Channel	A channel created on the request of the client and used to deliver notifications from a server to a client. The channel is represented as a resource and provides means for the server to post notifications and for the client to receive them via specified delivery mechanisms. For example in the case of Long Polling the channel resource is defined by a pair of URLs. One of the URLs is used by the client as a callback URL when subscribing for notifications. The other URL is used by the client to retrieve notifications from the Notification Server.
Notification Server	A server that is capable of creating and maintaining Notification Channels.
Server-side Notification URL	An HTTP URL exposed by a Notification Server, that identifies a Notification Channel and that can be used by a client when subscribing to notifications.
Structure-aware JSON generation	An approach to generate JSON that considers the information in the actual instance of a resource representation as well as information about the structure of that information. For example, this approach takes into account an XML representation of a resource, as well as the related XML schema information, e.g. cardinality or sequence.

3.3 Abbreviations

ACR	Anonymous Customer Reference
AGPL	Affero General Public License
API	Application Programming Interface
DNS	Domain Name Server
HTTP	Hypertext Transfer Protocol
ID	Identifier

IP	Internet Protocol
JSON	JavaScript Object Notation
OMA	Open Mobile Alliance
PLMN	Public Land Mobile Network
REST	REpresentational State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universal Unique Identifier
XML	Extensible Markup Language
XSD	XML Schema Definition

4. Introduction

To ensure consistency for developers using the various RESTful Network APIs specified in OMA, this “Common” technical specification aims to contain all items that are common across all HTTP protocol bindings using REST architectural style for the various individual interface definitions, such as naming conventions, content type negotiation, representation formats and serialization, and fault definitions. It also provides a repository for common data types.

4.1 Version 1.0

This version of the Common Definitions and Specifications for RESTful Network APIs is a republication of the ParlayREST Common V1.1 [ParlayREST_Common] and OMA REST Common V1.0 [OMA_REST_Common] specifications from the ParlayREST 2.0 release as part of the suite of OMA RESTful Network APIs. The content of these two specifications has been merged and restructured to fit that suite, and to separate general aspects from those aspects that are related to a Parlay X baseline [PSA]. Further, only bug fixes, but no functional changes have been applied.

Version 1.0 of the Common Definitions and Specifications for RESTful Network APIs contains naming conventions, content type negotiation, resource creation, representation formats and serialization, fault definitions and common data types for RESTful Network APIs. It also includes an Annex that provides specifications which are shared by those RESTful Network APIs which are based on Parlay X baselines.

5. Common Specifications for RESTful Network APIs

5.1 Use of REST Guidelines

REpresentational State Transfer (REST) is an architectural style for defining distributed systems. Entities in these systems communicate using the interfaces they expose. Guidelines for defining RESTful Network APIs in OMA, including general key principles, have been collected in [REST_NetAPI_WP].

As for message confidentiality and message authentication, the possible mechanism is available in [RFC2818] and related cipher suites are available in [SEC_CF-V1_1].

5.2 Unsupported Formats

Servers must return a 406 Not Acceptable error if a message body format (e.g. XML or JSON) requested by the application is not supported [RFC7231].

5.3 Authoring Style

5.3.1 Names

Names will be meaningful, and not abbreviated in a way that makes the name hard to understand for users of the REST interfaces that are not literate in computer programming. This does not preclude the use of commonly understood acronyms within names (e.g. ID) or commonly used abbreviations (e.g. max). However, the resulting name must still be meaningful.

5.3.2 Case usage for names

Two general cases are provided for, both using mixed case names; one with a leading capital letter, the other with a leading lowercase letter.

Names will start with a letter and be mixed case, with the leading letter of each but the first word capitalized. The conventions for the leading letter of the first differ depending on the context, as given below. Words will not be separated by white space, underscore, hyphen or other non-letter character.

The following names will have a leading uppercase letter – Type names and value names in an enumeration.

The following names will have a leading lowercase letter – all other names.

For names consisting of concatenated words, all subsequent words start with a capital, for example, “concatenatedWord” or “BothCapitals”. If a lowercase name starts with an abbreviation, all characters of the abbreviation are de-capitalized, e.g. “smsService”.

Path components of resource names are mixed case, with the leading letter lowercase. The leading path component which identifies the RESTful Network API (e.g. thirdpartycall) is all lowercase, and is aligned with the namespace name of the related XML schema.

5.4 Content type negotiation

The Content type of a response used SHALL be established using the following methodology:

As a general rule, content type used in response message body must match content type used in request body. At least XML and JSON content types MUST be supported.

Support for other content types will be specified on a case-by-case basis (e.g. simple name-value pair parameters may be accepted in the URL when using GET and application/x-www-form-urlencoded [W3C_URLENC] may be supported for the request message body when using POST or PUT).

Content type of the request message body SHALL always be determined by Content-Type header of the HTTP message.

Content type of the response body SHALL be determined using the following methodology. When invoking the RESTful Network API, the requesting application SHOULD include the ‘Accept’ request header, and provide the primary content type choice, and OPTIONALLY any supported substitute content types, in this request Accept header.

a. If the server does not support the content type choice listed as priority in the Accept header, it SHALL attempt to return the next preferred choice if one was provided.

b. If the requesting application does not provide an Accept header or any other indication of desired content type of the response (see further below), and the request message body content type is XML or JSON, then the server SHALL provide a response message body with the content type matching that of the request message body. For example, a request with an XML body and no Accept header will trigger an XML response.
c. If the requesting application requires the response message body to be of a different content type than the one determined by the request message body and the Accept header negotiations, it MUST request that content type by inserting in the URL path the query parameter "?resFormat={content type}", where content type SHALL be either XML or JSON. This option overrides the Accept header provided by the application, if present, and the response format SHALL be determined solely by the "resFormat" parameter. Note that this allows an application that does not have sufficient control over the HTTP headers to enforce a response format regardless of the value of the Accept header.
d. If the server cannot return any of the content types based on the negotiation steps described, it SHALL return a 406 response code as per [RFC7231].
e. The default format for notification payloads SHALL be determined as follows: by default, if the subscription was created using an entity body in XML or JSON format, the same format SHALL be used for notifications; if the subscription was created using an entity body in application/x-www-form-urlencoded format, the XML format SHALL be used for notifications. This default behavior can be overridden by using the "notificationFormat" parameter in the subscription.
f. Content type SHALL accompany HTTP response codes 200, 201, 400, 409 in the conditions dictated by the above specified methodology, and MAY be omitted in other cases.

5.5 Resource creation

5.5.1 General procedure of resource creation

Typically, a resource is created either following a POST request (to create a child of an existing resource that is addressed by the request), or following a PUT request (to create a new resource as addressed by the request).

When the server creates a new resource based on a POST request from the client, the prefix of the resource URL identifying the created resource SHALL be taken from the Request-URI of the POST request. To that prefix, the server SHALL append a "/" character followed by a variable part that uniquely identifies the resource. This variable part MUST NOT include any reserved character.

If a resource has been created on the server, the server SHALL return an HTTP response with a "201 Created" header and the Location header containing the location of the created resource, and SHALL include in the response body either a resourceReference element, or a representation of the created resource. Note that this allows the server to control the traffic.

Further note that REST resource representations are designed in such a way that they can include a self reference. (i.e. resourceURL element.). A self reference is always present in any data structure that is a representation of a resource created by POST, and can be included as necessary in other cases. Since a self reference can be defined as a mandatory or optional element to accommodate different situations, the normative aspects on the client and on the server in each optional usage instance in the specification are clarified as follows: 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 also MUST be included in responses to any HTTP method that returns an entity body, and in PUT requests.

Generally resources are used to access entire data structure and those resources are regarded as Heavy-weight Resources. To access a part of the data structure or an individual elements in the data structure, another type of resources called Light-weight Resources are used. Compared to Heavy-weight Resources, Light weight Resources are created following PUT request only (see [REST_NetAPI_WP] for more details about Light-weight Resources).

Elements in data structures with a key properties (keys) are normally not accessable by using Light-weight Resources, however when accessing other elements using Light-weight Resources they may appear in both the Light-weight Resource URL and in the body of the request. In case the server receives PUT request with keys, it SHALL ensure that the key value(s) specified in the URL match those value(s) specified in the body of the request. If not, the server SHALL respond with "409 Conflict" indicating key value(s) conflict.

5.5.2 Error recovery during resource creation

The following mechanism allows recovery from communication failures that can occur during resource creation using POST.

The client MAY (and in some cases SHOULD) include in the parameter set of the resource creation request the "clientCorrelator" field which uniquely maps to the resource to be created.

Note that this allows the client to retry a resource-creating request for which it did not receive an answer due to communication failure, and prevents the duplicate creation of resources on the server side in case of such retry. Note further that depending on the deployment (e.g. Network Address Translation, Proxies), the server might or might not be able to distinguish between different clients.

It is therefore RECOMMENDED that the client generates the value of the “clientCorrelator” in such a way that collisions (i.e. two unrelated requests use the same “clientCorrelator” value) are impossible or at least highly improbable. The way this is achieved is out of scope of this specification, however, it is pointed out that for example UUID [RFC4122] provides a way to implement such a scheme.

In case the server receives a “clientCorrelator” value in a resource-creating POST request, it SHALL do the following:

- in case the request contains a “clientCorrelator” value that has not been used yet to create a resource, the server SHALL create the resource and respond with "201 Created", as above.
- in case the request contains a “clientCorrelator” value that has already been used to create a resource, the server responds as follows:
- in case this is a valid repeated attempt by the same client to create the same resource, the server SHALL respond with "200 OK", and SHALL return a representation of the resource.
- otherwise, it SHALL respond with "409 Conflict", in this case indicating a clientCorrelator conflict, and SHOULD include a payload with a “requestError” structure carrying a “SVC0005 Duplicate correlator” ServiceException. In such case, the client can retry the request using a new “clientCorrelator” value.

5.6 JSON encoding in HTTP Requests/Responses

5.6.1 Serialization rules: Instance-based JSON generation

Specifications of RESTful Network APIs MAY include XML schema files defining the data structures used by that API, for its direct usage in XML format. The following are rules for mapping between XML instances and JSON data formats:

a. XML elements that appear at the same XML hierarchical level (i.e. either root elements or within the same XML parent element), are mapped to a set of *name:value* pairs within a JSON object, as follows:

   (i) Each XML element appearing only once at the same hierarchical level (*“single element”*) is mapped to an individual *name:value* pair. The *name* is formed according to bullet b, while the *value* is formed according to bullet c.

   (ii) XML elements appearing more than once at the same hierarchical level (*“element list”*) are mapped to only one, individual *name:value* pair. The *name* is formed according to bullet b, while the *value* is a JSON array containing one *value* per each occurrence of the XML element. The name is formed according to bullet b whilst values are formed according to bullet c.

   (iii) *Name* and *Value* of JSON objects will go between “”. Additionally, any JSON representation of an element of complex type will go between {}, according to [RFC7159].

b. The *name* of the *name:value* pair is the name of the XML elements (i.e. XML_element_name:value)

c. The *value* is formed as follows:

(i) when the XML element has neither attributes nor child XML elements, the *value* is equal to the value of the XML element. In case the element is nill (i.e it has no value), it will be indicated as having a "null" value within JSON.

(ii) when the XML element has child elements and/or attributes, the *value* is a JSON object containing the following *name:value* pairs:

- one *name:value* pair per each attribute, where *name* is the name of the attribute and *value* is the value of the attribute.
- one *name:value* pair associated to the text value (simple type content) of the XML element, where *name* is the string "$t" and *value* is the value of the XML element.
- *name:value* pairs associated to XML child elements. These *name:value* pairs are formed in accordance with bullet a.

Within JSON, there is no need to reflect:

- the first <?xml version="1.0" encoding="UTF-8" ?> tag
- declaration of namespaces or schemaLocations
- the "xml:space" attribute and its value.

If the XML instance contains an xsi:type attribute, the handling depends on the actual API. The "xsi:type" attribute is used to realize polymorphism in XML instances; it signals the actual type of an instance. Such information may or may not be needed in the JSON representation. Individual API specification will state whether the "xsi:type" attribute is included (default) or excluded. If the attribute is included, it is handled during the conversion the same way as any other attribute (note that the "xsi:" namespace prefix is removed like any other namespace prefix).

If the content of an element is embedded in a CDATA wrapper, that wrapper SHALL be removed before applying the XML-to-JSON conversion to that element.

In order to generate unambiguous JSON from XML instances, based on the rules defined above, the following limitations need to be imposed on the XML data structures:

- it is not allowed that two different elements from different namespaces have the same name, in case they appear at the same level
- within an XML parent element, no attribute is allowed to have the same name as a child element of this parent element.

Note: The instance-based approach to JSON generation defined by the rules above has been used to generate the JSON examples from the XML examples in the Technical Specifications of the OMA RESTful Network APIs.

Note: The instance-based JSON generation represents a pipe-based approach of data format transformation without any side information. As no side information is available, this approach is not able to take structural information into account. The most visible artifact of this is the fact that an array with one element is represented using the same syntax as for representing a scalar value.

5.6.1.1 Utility which implements the instance-based JSON generation rules (Informative)

The general conversion rules are implemented with UNICA XML2JSON utility, an open source tool, distributed, under an AGPL license, within the open source community MORFEO [XML2JSON].

5.6.1.2 Example (Informative)

The following is an example illustrating the guidelines:

Input XML content:

```
<Animals>
```

```
    <dog>
        <name attr="1234">Rufus</name>
        <Breed>labrador</Breed>
    </dog>
    <dog>
        <name>Marty</name>
        <Breed>whippet</Breed>
        <a/>
    </dog>
    <dog/>
    <cat name="Matilda"/>
    <a/>
</Animals>
```

Transformed JSON:

```
{"Animals": {
   "a": null,
   "cat": {"name": "Matilda"},
   "“dog": [
      {
         "Breed": "labrador",
         "name": {
            "$t": "Rufus",
            "attr": "1234"
         }
      },
      {
         "Breed": "whippet",
         "a": null,
         "name": "Marty"
      },
      null
   ]
}}
```

5.6.2 Serialization rules: structure-aware JSON generation

The instance-based approach as defined above relies only on the information in the XML data instance.

In contrast, the structure-aware approach defined in this section considers information in a data instance (e.g. XML) plus further information about the data structure definition (such as the allowed number of element occurrences), as documented in the RESTful Network API specifications and XML Schemas.

This structure-aware approach allows having always the same JSON structure to convey arrays of elements, no matter whether the array contains one element or a plurality of elements.

In this conversion approach, the rules above apply, except for the following modification to the conditions in a (i) and a (ii):If an element is allowed to appear more than once at the same hierarchical level, it SHALL be treated according to a (ii) as element list, otherwise it SHALL be treated according to a (i) as single element.

Note: The structure-aware JSON generation represents a model-based approach of data serialization using side information. Such side information, which typically includes the allowed number of occurrences of an element, allows representing an array with one element using array syntax, rather than using the syntax for representing a scalar value.

5.6.2.1 Example (Informative)

The following example illustrates the structure-aware JSON generation.

In the example, the data instance is represented as XML document:

```
<Animals>
    <dog>
        <name attr="1234">Rufus</name>
        <Breed>labrador</Breed>
    </dog>
    <dog>
        <name>Marty</name>
        <Breed>whippet</Breed>
                                <a/>
    </dog>
    <dog/>
    <cat name="Matilda"/>
    <a/>
</Animals>
```

The information about the data structure is represented as XML schema in this example. Note that the maximum cardinality of the elements is the only piece of information that is used here.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="Animals">
          <xsd:complexType>
               <xsd:sequence>
                    <xsd:element name="dog" maxOccurs="unbounded">
                         <xsd:complexType>
                              <xsd:sequence>
                                   <xsd:element name="name" minOccurs="0">
                                        <xsd:complexType>
                                             <xsd:simpleContent>
                                                  <xsd:extension base="xsd:string">
                                                       <xsd:attribute name="attr" type="xsd:string"/>
                                                  </xsd:extension>
                                             </xsd:simpleContent>
                                        </xsd:complexType>
                                   </xsd:element>
                                   <xsd:element name="Breed" type="xsd:string" minOccurs="0"/>
                                   <xsd:element name="a" minOccurs="0"/>
                              </xsd:sequence>
                         </xsd:complexType>
                    </xsd:element>
                    <xsd:element name="cat" maxOccurs="unbounded">
```

```
                <xsd:complexType>
                    <xsd:simpleContent>
                        <xsd:extension base="xsd:string">
                            <xsd:attribute name="name" type="xsd:string" use="required"/>
                        </xsd:extension>
                    </xsd:simpleContent>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="a"/>
        </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Transformed JSON:

```
{"Animals": {
   "dog": [
      {
         "name": {
            "$t": "Rufus",
            "attr": "1234"
         }
         "Breed": "labrador",
      },
      {
         "name": "Marty"
         "Breed": "whippet",
         "a": null,
      },
      null
   ]
   "cat": [{"name": "Matilda"}],
   "a": null,
}}
```

5.6.3 Rules for JSON-creating and JSON-consuming applications

A JSON-creating application SHALL use either the *structure-aware* or the *instance-based* approach, but not both.

Applications that consume a JSON representation SHALL accept the following two different JSON representations of an array that contains one element:

1. a pair of name and value (e.g. “name”: “one”)
2. a pair of name and array of one value (e.g. “name”: [“one”])

Note: In JSON, according to [RFC7159], the order of objects is not significant, whilst the order of values within an array is.

5.7 Encoding and Serialization Details for MIME format

A MIME multipart message often consists of several parts:

- The root structure, which is a data structure defined in the RESTful Network API specification, expressed in the different possible formats (such as XML or JSON). This part conveys the resource parameters.
- The multimedia contents or attachments as MIME body parts, within the HTTP request or response. They include all contents, both plain text as well as other content types (images, videos, etc).

To represent such MIME multipart messages, there are different options available, namely multipart/related [RFC2387], and multipart/form-data [RFC2388]. The selection of the multipart format to use in a particular API needs to consider multiple factors, such as the conventions in the domain in which the API is defined, how tightly the API is to be coupled to the underlying systems, and how easy the format is to use in the Web community and in browser environments.

In OMA RESTful Network APIs, for simplicity purposes and better suitability to the internet developer community and browsers, multipart/form-data [RFC2388] and [HTML FORMS] can be used instead of multipart/related.

To represent the different categories of message parts in a multipart/form-data message, the following is defined:

1. **Root fields** as described above SHALL 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 content types for the root fields are:

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

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

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

      Content-Disposition: form-data; name=“attachments”

      Content-Type: <Any multipart Content-Type>

      Any multipart Content-Type SHALL be permitted, including multipart/mixed [RFC2046], multipart/related [RFC2387], application/vnd.wap.multipart.mixed, and application/vnd.wap.multipart.related [MMS CONF].

      Then, every one of the possible message contents SHALL be included as subparts according to the enclosing multipart type, with:

      Content-Disposition: <Appropriate disposition for this content item>

      Content-Type: <Corresponding Content-Type>

      The format of Content-Disposition is defined in [RFC2183] and [RFC2231], for example:

      Content-Disposition: attachment; filename=“<Name of the message content>”

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

5.8 Resource URL considerations

5.8.1 Resource URL structure

Each resource URL consists of fixed and variable parts.

For fixed parts, the exact string value is defined by this specification. Implementations SHALL use the exact string of fixed parts.

For variable parts, rules how to build the string value are defined by this specification. Implementations SHALL follow these rules. The variable parts are referred to as “Resource URL variables” in the individual OMA RESTful Network API specifications. Resource URL variables are denoted by a name in curly brackets, such as {apiVersion}.

5.8.1.1 Use of ‘acr:auth’ as a resource URL variable

In the case where a resource URL includes a resource URL variable that identifies a user (e.g. {endUserId}, {senderAddress}, etc), the value of this variable MAY be in the form of an ‘acr’ URI (Appendix H of [REST_NetAPI_ACR]).

The use of ‘acr:auth’ is a special case that SHALL be supported. The reserved keyword ‘auth’ MUST not be assigned as an ACR to any particular user.

When detecting ‘acr:auth’ in the resource URL path the server SHALL:

1. if an authorization token is present
   - o validate the authorization token:
   - o if the token is invalid, return “401 Unauthorized” with a SVC2003 entity body,
   - o else derive the identity of the user from the authorization token and continue processing the request
2. else if the authorization token is not present but the network has other means to authenticate the user (such as using internal procedures of the mobile network to derive the MSISDN from the IP address of the device that sent the request,)
   - o derive the identity of the user using network-internal procedures and continue processing the request
3. else return an HTTP response code “400 Bad Request” with a SVC0002 entity body

As acr: auth represents the user on whose behalf an application is acting, implementations that make use of acr:auth in resource URLs need to be aware of the following: The resource space seen by an application is only conflict-free (i.e. each resource URL corresponds to at most one resource) if the application acts on behalf of one user. An application acting on behalf of multiple users needs to be aware that the same resource URL can address different actual resources, depending on the user represented by the authorization information (such as the access token). Such applications need to disambiguate the resource space by additionally considering the identity of the user when identifying a resource by its resource URL.

For more details on authorization see Appendix D of this specification and [Autho4API_10].

For any specific impact regarding the use of ‘acr:auth’ on a particular OMA RESTful Network API, see the Technical Specification for that particular OMA RESTful Network API.

5.8.2 API version signaling

Each resource URL contains a variable "apiVersion" which signals the version of the API that is used. The value of this variable SHALL be set to "v1" in the initial version of a particular API. In subsequent revisions of the aforementioned API the digit SHALL be incremented by “1” (e.g. increment from “v1” to “v2”).

In each HTTP request sent by the application, the “apiVersion” variable MUST be included in the Request-URI field (which is defined by [RFC7231]). The following applies to the server answering such a request:

- If the server supports the version signaled by the application, it MUST use the same version in the response, as follows: In each HTTP response sent by the server to answer such a request, the “apiVersion” variable MUST be present in the “resourceURL” element(s) in the body of the response. Additionally, if the response contains a “Location” HTTP header (e.g. in case of responses to a resource creation request), the “apiVersion” variable MUST also be present in the URL signaled in that header.
- Otherwise, the server MUST respond as defined in section 5.8.3.

In an HTTP request sent by the server towards the application (i.e. a notification), the server MUST use in the Request-URI field a resource URL with the same API version as in the subscription which has triggered the notification.

Note that the change in the API version can imply a change of the actual data structures used, and functionality offered.

5.8.3 Handling of unsupported versions

In case the server does not support the API version that the application has signaled in a request, but the server supports other versions of the resource in question, the server MUST return a response that lists the available versions and related resource URLs on which the client could repeat the request.

For that, the server MUST return a "300 Multiple Choices" response, with a "versionedResourceList" root element as defined in section 6.2.1.7. In case there is only one version supported by the server, the HTTP header "Location" MUST be populated with the according resource URL. In case there are multiple versions supported by the server, the server MAY populate the "Location" header with the choice deemed most appropriate given the version that was requested. Usually this is the highest version supported by the server which is lower than the requested one.

5.8.3.1 Example 1: Signalling supported versions in case an unsupported version was requested (XML format) (Informative)

5.8.3.1.1 Request

```
GET /exampleAPI/smsmessaging/v2/outbound/tel%3A%2B19585550151/requests HTTP/1.1
Accept: application/xml
Host: example.com
```

5.8.3.1.2 Response

```
HTTP/1.1 300 Multiple Choices
Content-Type: application/xml
Content-Length: nnnn
Location: http://example.com/exampleAPI/smsmessaging/v1/outbound/tel%3A%2B19585550151/requests
Date: Thu, 04 Jun 2009 02:51:59 GMT

<?xml version="1.0" encoding="UTF-8"?>
<common:versionedResourceList xmlns:common="urn:oma:xml:rest:netapi:common:1">
  <resourceReference>
    <apiVersion>v1</apiVersion>
    <resourceURL>http://example.com/exampleAPI/smsmessaging/v1/outbound/tel%3A%2B19585550151/requests</resourceURL>
  </resourceReference>
  <resourceReference>
    <apiVersion>v3</apiVersion>
    <resourceURL>http://example.com/exampleAPI/smsmessaging/v3/outbound/tel%3A%2B19585550151/requests</resourceURL>
  </resourceReference>
</common:versionedResourceList>
```

5.8.3.2 Example 2: Signalling supported versions in case an unsupported version was requested (JSON format) (Informative)

5.8.3.2.1 Request

```
GET /exampleAPI/smsmessaging/v2/outbound/tel%3A%2B19585550151/requests HTTP/1.1
Accept: application/json
Host: example.com
```

5.8.3.2.2 Response

```
HTTP/1.1 300 Multiple Choices
Content-Type: application/json
Content-Length: nnnn
Location: http://example.com/exampleAPI/smsmessaging/v1/outbound/tel%3A%2B19585550151/requests
Date: Thu, 04 Jun 2009 02:51:59 GMT

{"versionedResourceList": {"resourceReference": [
   {
      "apiVersion": "v1",
      "resourceURL": "http://example.com/exampleAPI/smsmessaging/v1/outbound/tel%3A%2B19585550151/requests"
   },
   {
      "apiVersion": "v3",
      "resourceURL": "http://example.com/exampleAPI/smsmessaging/v3/outbound/tel%3A%2B19585550151/requests"
   }
]}}
```

5.9 Backward compatibility

When processing an XML data structure that contains attributes and/or elements not known to a client/server conforming to a certain version of the API, the result of processing that data structure SHALL be the same as the result of processing a data structure where these attributes, or elements including their child elements and attributes, were not present.

When processing a JSON data structure that contains name-value-pairs where the name is not known to a client/server conforming to a certain version of the API, the result of processing that data structure SHALL be the same as the result of processing a data structure where these name-value pairs including their child name-value pairs were not present.

When processing an application/x-www-form-urlencoded [W3C_URLENC] data structure that contains name-value-pairs where the name is not known to a client/server conforming to a certain version of the API, the result of processing that data structure SHALL be the same as the result of processing a data structure where these name-value pairs were not present.

Note: backward compatibility processing of XML, JSON or application/x-www-form-urlencoded data structures, as described above, can be achieved by ignoring the unknown attributes or elements and their child elements/attributes.

6. Data Items

6.1 Address data items

Addresses, unless the specification provides specific additional instruction, MUST conform to the address portion of the URI definition provided in [RFC3966] for 'tel:' URIs, [RFC3261] for 'sip:' URIs, Appendix H of [REST_NetAPI_ACR] for 'acr' URIs or the definition given below for shortcodes or aliased addresses. Optional additions to the address portion of these URI definitions MUST NOT be considered part of the address accepted by the RESTful Network APIs, and an implementation MAY choose to reject an address as invalid if it contains any content other than the address portion.

A tel: URI MUST be defined as a global number (e.g. tel:+19585550100). The use of characters other than digits and the leading "+" sign SHOULD be avoided in order to ensure uniqueness of the resource URL. This applies regardless of whether the user identifier appears in a URL variable or in a parameter in the body of an HTTP message.

When specified in the definition of a service operation, the URI may contain wildcard characters in accordance with the appropriate specification (i.e. [RFC3966] or [RFC3261]).

Shortcodes are short telephone numbers, usually 4 to 6 digits in length reserved for telecom service providers' own functionality. A shortcode is signalled as a string of decimal digits without URI scheme.

Support for aliases in addresses is provided by use of the URI defined in [RFC3986]. One cannot assume that the resource the alias references can be determined without using the URI to access the resource.

An alias is generally a relatively short character string that holds a scrambled address such that only the application identified in the URI can expand it.

6.2 Common data types

This section defines data types which are shared among two or more RESTful Network APIs.

The namespace for the common data types is:

urn:oma:xml:rest:netapi:common:1

The 'xsd' namespace is used in the present document to refer to the XML Schema data types defined in XML Schema [XMLSchema1, XMLSchema2]. The use of the name 'xsd' is not semantically significant.

6.2.1 Structures

6.2.1.1 Type: ChargingInformation

For services that include charging as an inline message part, the charging information is provided in this data structure. See section 6.3 for more information.

Element	Type	Optional	Description
description	xsd:string [1..unbounded]	No	An array of description text to be used for information and billing text.
currency	xsd:string	Yes	Currency identifier as defined in [ISO4217].
amount	xsd:decimal	Yes	Amount to be charged/refunded/reserved. The amount to be charged/refunded/reserved appears either directly in the amount-field or as code in the code-field. If both these two fields are missing or empty a service exception (SVC0007) will be thrown.
code	xsd:string	Yes	Charging code, referencing a contract under which the charge is applied.

Table 1: ChargingInformation Structure

6.2.1.2 Type: CallbackReference

An application can use the CallbackReference data structure to subscribe to notifications.

If a parameter *callbackData* has been passed in a particular subscription, the server MUST copy it into each notification which is related to that particular subscription.

Element	Type	Optional	Description
notifyURL	xsd:anyURI	No	Notify Callback URL
callbackData	xsd:string	Yes	Data the application can register with the server when subscribing to notifications, and that are passed back unchanged in each of the related notifications. These data can be used by the application in the processing of the notification, e.g. for correlation purposes.
notificationFormat	NotificationFormat	Yes	Application can specify format of the resource representation in notifications that are related to this subscription. The choice is between {XML, JSON}. If this parameter is absent, the notification format MUST be the same as the format used in the subscription request (for XML and JSON), or MUST be XML for application/x-www-form-urlencoded subscription requests.

Table 2: CallbackReference Structure

Note: In case the application requires correlating notifications to the related subscription, it can either submit a different *notifyURL* in each subscription, or use the optional *callbackData* parameter as a correlator.

6.2.1.3 Type: ResourceReference

Element	Type	Optional	Description
resourceURL	xsd:anyURI	No	The URL that addresses the resource. 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 also be included in responses to any HTTP method that returns an entity body, and in PUT requests.

Table 3: ResourceReference Structure

The *resourceReference* element of type *ResourceReference* is defined as a root element in the XSD.

6.2.1.4 Type: Link

Attribute	Type	Optional	Description
rel	xsd:string	No	Describes the relationship between the URI and the resource
href	xsd:anyURI	No	URI

Table 4: Link Structure

An element of type *Link* can be provided by the server to point to other resources that are in relationship with the resource. The *rel* attribute is a string. The possible values for the string are defined in each RESTful Network API. *Rel* and *href* are realized as attributes in the XSD.

6.2.1.5 Type: LanguageString

String with an attribute that signals the language of the contained text.

This type is defined as a string of base type xsd:string with an OPTIONAL instance of the built-in XML attribute xml:lang [W3C-XML11].

6.2.1.6 Type: VersionedResource

A resource with associated version string.

Element	Type	Optional	Description
apiVersion	xsd:string	No	The API version provided by the resource.
resourceURL	xsd:anyURI	No	The URL that addresses the resource.

Table 5: VersionedResource structure

6.2.1.7 Type: VersionedResourceList

A list of resources with associated version string.

This data structure and associated root element is intended to signal a list of supported resource versions in case the client has requested an unsupported version, and SHALL NOT be used for any other purpose.

Note that this restriction has been defined because this message may occur in place of any response.

Element	Type	Optional	Description
resourceReference	VersionedResource [1..unbounded]	No	A resource URL with associated version.

Table 6: VersionedResourceList structure

The *versionedResourceList* element of type *VersionedResourceList* is defined as a root element in the XSD.

6.2.2 Enumerations

6.2.2.1 Enumeration: NotificationFormat

List of notification format values.

Enumeration	Description
XML	Notification about new inbound message would use XML format in the POST request
JSON	Notification about new inbound message would use JSON format in the POST request

Table 7: NotificationFormat Values

6.2.2.2 Enumeration: RetrievalStatus

Enumeration	Description
Retrieved	Data retrieved. Current data is provided
NotRetrieved	Data not retrieved, current data is not provided (does not indicate an error, no attempt may have been made). Note that this field is useful in case a list of addresses are requested, some items could be marked as "NotRetrieved" in case retrieval could not be attempted for some reason, e.g. to avoid time outs
Error	Error retrieving data

Table 8: RetrievalStatus

6.3 Charging

This section deals with in-band charging, i.e. passing charging data as part of the RESTful Network API request. To enable this capability to be provided across a variety of services in a consistent manner, the information to be provided in the message for charging information is defined as a common charging data type.

6.3.1 Charging data type

The charging information is provided in an XML data type, using the following schema. See section 6.2.1.1 for the formal definition.

```
<xsd:complexType name="ChargingInformation">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element name="currency" type="xsd:string" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="amount" type="xsd:decimal" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="code" type="xsd:string" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

The application accessing the Service provides this information:

- *"description"* is an array of text. The first entry of a list will often be used to provide billing text. This text does not have specific required content, but would likely include information on the business, the content or service provided, and a transaction identifier. Credit card statements are a good example of description text provided by different companies.

- When more than one entry is provided, the rest should be references to individual operations relevant to the charging. Reference should be set to a value provided in a response message to the operation as a unique identifier to correlate individual operation.

- *"currency"* in which the charge is to be applied. Values for the currency field are defined by [ISO4217].

- *"amount"* defines the amount to be charged.

- *"code"* specifies a charging code which references a contract under which this charge is applied. The code identifier is provided by the Service Provider.

The charging information provided may not be acceptable to the Service Provider. For example, the Service Provider may limit the amount that may be specified for a particular Service or for a particular Service Requester. If the information provided is not acceptable, an appropriate fault message may be returned to the requester (SVC0007 and POL0012 are defined as a generic charging fault, The 'SVC' and 'POL' service exceptions are defined in [ParlayX_Common]).

Especially in case of charging operation such as creating a charge or refund, it is strongly recommended to convey a list of relevant operations related to charging over a description part as described above.

This is useful especially when a charging operation is performed after a certain set of operations.

Some of the services may be meaningful to the user only when a certain set of operations is completed. In that case, service provider may want to charge a user only upon a completion of the entire process, instead of charging per operation. Also, service provider may want to control the actual amount of charging depending on a certain condition, e.g., service usage volume, independent of the volume control provided by the network operators. This is also the case where it is preferable to perform charging operation after a completion of certain set of operations. In these cases where a service provider charges a user for the consumption of a certain service, the service provider is recommended to provide the references to the individual operations performed as evidences. This information can be referenced by the relevant entities to ensure the validity of charging when necessary.

It should be noted that this is for a service provider to provide a list of evidences of their direct use of operations. Any mapping of underlying operations performed internally in the operator must be performed by the operator if necessary. How to maintain the consistency between the information kept at service provider and the operators is out of scope. Also, charging aspects which do not relate to any operations are not covered.

7. Error Handling

7.1 HTTP Response Codes

Following is a list of often used HTTP response codes for RESTful Network APIs. The full set of HTTP response codes can be found in [RFC7231]. The first line of each error code has been copied from [RFC7231]. The second line gives a short informative explanation of the meaning of the error code. For a normative description of the error code see [RFC7231].

200 OK
The operation was successful.

201 Created
The operation was successful, and a new resource has been created by the request.

202 Accepted

The request has been accepted for processing, but the processing has not been completed (yet).

204 No Content
The operation was successful, and the response intentionally contains no data.

300 Multiple Choices
The requested resource corresponds to any one of a set of representations, each with its own specific location. In the OMA RESTful Network APIs, this code is for instance used to signal the supported API versions in case an unsupported version was requested for a particular resource.

303 See Other
The response to the request can be found under a different URI and can be retrieved using a GET method on that resource.

304 Not Modified
[RFC7232] The condition specified in the conditional header(s) was not met for a read operation.

400 Bad Request
In the original HTTP meaning, this error signals invalid parameters in the request. In OMA RESTful Network APIs, this code is also used as the “catch-all” code for error situations triggered by a client request, for which no matching HTTP error code exists.

401 Unauthorized
[RFC7235] Authentication has failed, but the application can retry the request using authorization.

403 Forbidden
The server understood the request, but is refusing to fulfil it (e.g. because application doesn't have permissions to access resource due to the policy constraints)

404 Not Found
The specified resource does not exist.

405 Method Not Allowed
The actual HTTP method (such as GET, PUT, POST, DELETE) is not supported by the resource

406 Not Acceptable
The content type requested is not acceptable for the resource.

408 Request Timeout
The client did not produce a response in the time the server was prepared to wait.

409 Conflict
Occurs in situations when two instances of an application are trying to modify a resource in parallel, in a non-synchronized way.

410 Gone
The requested resource is no longer available at the server.

411 Length Required
The Content-Length header was not specified.

412 Precondition Failed
[RFC7232] The condition specified in the conditional request header(s) was not met for an operation.

413 Payload Too Large
The size of the request body exceeds the maximum size permitted by the server implementation.

414 URI Too Long
The length of the request URI exceeds the maximum size permitted by the server implementation.

415 Unsupported Media Type
The content type of the request body is unsupported by the server.

429 Too Many Requests
The client has sent too many requests in a given amount of time (“rate limiting”) **[RFC6585]**. The server SHOULD include a Retry-After header indicating how long to wait before making a new request. The client SHOULD respect this header.

500 Internal server error
General, catch-all server-side error

503 Service Unavailable
The server is currently unable to receive requests, but the request can be retried at a later time.

7.2 Handling of not allowed HTTP methods

If a method is not allowed by the resource (error code 405), then server SHOULD also include the ‘Allow: {GET|PUT|POST|DELETE} HTTP header in the response as per sections 6.5.5 and 7.4.1 in [RFC7231].

7.3 HTTP Response Codes in Response to Notifications

Handling of HTTP response codes sent by the client application, in response to a notification from the server:

1. in case of HTTP 2xx response codes, server assumes the notification has been sent successfully.
2. in case of HTTP response codes other than 2xx, the handling is left to the server implementation. The server MAY support different actions as dictated by a service provider policy (out-of-scope for this specification).

Appendix A. Change History (Informative)

A.1 Approved Version History

Reference	Date	Description
OMA-TS-REST_NetAPI_Common-V1_0-20180116-A	16 Jan 2018	Status changed to Approved by TP TP Ref # OMA-TP-2018-0001-INP_REST_NetAPI_Common_V1_0_RRP_for_final_Approval

Appendix B. Shared Definitions for Exception Handling in RESTful Network APIs based on Parlay X (Normative)

This appendix defines building blocks for exception handling which are shared among those RESTful Network APIs which have corresponding Parlay X [PSA] specifications as the baseline. These building blocks have been inherited and possibly adapted from [ParlayX_Common].

RESTful Network APIs not having a Parlay X baseline can reference these as well if appropriate.

If an API re-uses the charging mechanism defined in section 6.3, this implies support for handling the RequestError type as well.

B.1 Common data types for exception handling

B.1.1 Type: RequestError

Element	Type	Optional	Description
link	Link[0..unbounded]	Yes	Link to elements external to the resource
serviceException	ServiceException	Choice	Exception Details
policyException	PolicyException	Choice	Exception Details

Table 9: RequestError

A requestError element of type *RequestError* is defined as a root element in the XSD.
XSD modelling uses a "choice" to select either a serviceException or a policyException.

B.1.2 Type: ServiceException

Element	Type	Optional	Description
messageId	xsd:string	No	Message identifier, with prefix SVC
text	xsd:string	No	Message text, with replacement variables marked with %*n*, where *n* is an index into the list of <variables> elements, starting at 1
variables	xsd:string [0..unbounded]	Yes	Variables to substitute into Text string

Table 10: ServiceException

B.1.3 Type: PolicyException

Element	Type	Optional	Description
messageId	xsd:string	No	Message identifier, with prefix POL
text	xsd:string	No	Message text, with replacement variables marked with %*n*, where *n* is an index into the list of <variables> elements, starting at 1
variables	xsd:string [0..unbounded]	Yes	Variables to substitute into Text string

Table 11: PolicyException

B.1.4 Type: ServiceError

In a response to a request, ServiceError is used when an operation involving multiple items fails for only some of the items, whereas ServiceException is used where the entire operation fails.

In notifications, ServiceError is always used to indicate a notification termination or cancellation.

Element	Type	Optional	Description
messageId	xsd:string	No	Message identifier, either with prefix SVC or with prefix POL
text	xsd:string	No	Message text, with replacement variables marked with %n, where n is an index into the list of <variables> elements, starting at 1
variables	xsd:string [0..unbounded]	Yes	Variables to substitute into text string

Table 12: ServiceError

B.2 Handling of Service and Policy exceptions

In case of errors, additional information in the form of Exceptions MAY be included in the HTTP response.

Exceptions are defined with three data elements.

The first data element is a unique identifier for the message. This allows the receiver of the message to recognize the message easily in a language-neutral manner. Thus applications and people seeing the message do not have to understand the message text to be able to identify the message. This is very useful for customer support as well, since it does not depend on the reader to be able to read the language of the message.

The second data element is the message text, including placeholders (marked with %) for additional information. This form is consistent with the form for internationalization of messages used by many technologies (operating systems, programming environments, etc.). Use of this form enables translation of messages to different languages independent of program changes.

The third data element is a list of zero or more strings that represent the content to put in each placeholder defined in the message in the second data element with the first entry mapping to the placeholder %1.

B.2.1 Service exception

The *Service exception* is provided in an XML data type, using the following schema.

```
<xsd:complexType name="ServiceException">
  <xsd:sequence>
    <xsd:element name="messageId" type="xsd:string"/>
    <xsd:element name="text" type="xsd:string"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="variables" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

When a service is not able to process a request, and retrying the request with the same information will also result in a failure, and the issue is not related to a service policy issue, then the service will issue a fault using the ServiceException fault message. A Service Exception uses the letters 'SVC' at the beginning of the message identifier. General 'SVC' service exceptions are defined in Appendix C.

Examples of *Service exceptions* include invalid input, lack of availability of a required resource or a processing error.

B.2.2 Policy exception

The policy exception is provided in an XML data type, using the following schema.

```
<xsd:complexType name="PolicyException">
  <xsd:sequence>
    <xsd:element name="messageId" type="xsd:string"/>
    <xsd:element name="text" type="xsd:string"/>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="variables" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

When a service is not able to complete because the request fails to meet a policy criteria, then the service will issue a fault using the *Policy Exception* fault message. To clarify how a *Policy Exception* differs from a *Service Exception*, consider that all the input to an operation may be valid as meeting the required input for the operation (thus no *Service Exception*), but using that input in the execution of the service may result in conditions that require the service not to complete. A *Policy Exception* uses the letters 'POL' at the beginning of the message identifier. General ‘POL’ service exceptions are defined in Appendix C.

Examples of *Policy exceptions* include privacy violations, requests not permitted under a governing service agreement or input content not acceptable to the service provider.

Appendix C. Common Exception Definitions (Normative)

Note: The exception codes from 0001 to 0999 are inherited from ParlayX [ParlayX_Common] and ParlayREST 2.0 [ParlayREST_20]. New exception codes defined by the individual OMA RESTful Network APIs occupy the range from 1000 to 1999. New common exception codes for the OMA RESTful Network APIs are defined in the range from 2000 to 2099. The range from 2100 – 2999 is reserved for future use. The range from 3000 – 3499 can be used for experimental or private purposes; these values will not be assigned in a specification.

C.1 Common Service Exceptions

Faults related to the operation of the service, not including policy related faults, result in the return of a ServiceException message.

C.1.1 SVC0001: Service error

Name	Description
MessageId	SVC0001
Text	A service error occurred. Error code is %1
Variables	%1 Error code from service
HTTP status code(s)	400 Bad request

Note that an error code is an arbitrary machine-readable string that usually does not include a human-readable text. It should not be confused with the HTTP status code. If a human-readable text is to be included with the error message, SVC2000 is the appropriate exception to be used. If no error code is available, specify “0” here.

C.1.2 SVC0002: Invalid input value

Name	Description
MessageId	SVC0002
Text	Invalid input value for message part %1
Variables	%1 - message part
HTTP status code(s)	400 Bad request

C.1.3 SVC0003: Invalid input value with list of valid values

Name	Description
MessageId	SVC0003
Text	Invalid input value for message part %1, valid values are %2
Variables	%1 - message part %2 – comma-separated list of valid values. Blanks are allowed.
HTTP status code(s)	400 Bad request

If the range of valid values is large or complicated, consider using SVC2004 instead.

C.1.4 SVC0004: No valid address(es)

Name	Description

MessageID	SVC0004
Text	No valid addresses provided in message part %1
Variables	%1 - message part
HTTP status code(s)	404 Not found, 400 Bad request

If the address is part of the resource URL, the status code 404 SHOULD be used; otherwise the status code 400 SHOULD be used.

See also SVC2008, which allows the server to provide more detail.

C.1.5 SVC0005: Duplicate correlator

Name	Description
MessageID	SVC0005
Text	Correlator %1 specified in message part %2 is a duplicate
Variables	%1 - correlator %2 - message part
HTTP status code(s)	409 Conflict

See section 5.5.2 for more information.

C.1.6 SVC0006: Invalid group

Name	Description
MessageID	SVC0006
Text	Group %1 in message part %2 is not a valid group
Variables	%1 - identifier for the invalid group %2 - message part
HTTP status code(s)	400 Bad request

C.1.7 SVC0007: Invalid charging information

Name	Description
MessageID	SVC0007
Text	Invalid charging information
Variables	None
HTTP status code(s)	400 Bad request

C.1.8 SVC0008: Overlapping Criteria

Name	Description
MessageID	SVC0008
Text	Overlapped Criteria %1

Variables	%1 Message part with the overlapped criteria
HTTP status code(s)	400 Bad request

C.1.9 SVC2000: Service Error with description

This is similar to SVC0001, however, it allows a more structured error handling by communicating two variables: a machine-readable error code and an according human-readable description.

Name	Description
MessageID	SVC2000
Text	The following service error occurred: %1. Error code is %2
Variables	%1 Textual description of the error %2 Error code
HTTP status code(s)	400 Bad request, 500 Internal Server Error

C.1.10 SVC2001: No server resources available to process the request

Name	Description
MessageID	SVC2001
Text	No resources
Variables	None
HTTP status code(s)	503 Service unavailable

C.1.11 SVC2002: Requested information not available

Name	Description
MessageID	SVC2002
Text	Requested information not available for address %1
Variables	%1 Address for which the information is not available
HTTP status code(s)	404 Not found

See also SVC2008, which allows the server to provide more detail.

C.1.12 SVC2003: Invalid access token

Name	Description
MessageID	SVC2003
Text	Invalid access token
Variables	None
HTTP status code(s)	401 Unauthorized, 403 Forbidden

C.1.13 SVC2004: Invalid input value with details

Name	Description
MessageId	SVC2004
Text	Invalid input value for %1 %2: %3
Variables	%1 – type of item, e.g., “element” or “attribute” %2 – identifier of invalid item %3 – human-readable description
HTTP status code(s)	400 Bad request

C.1.14 SVC2005: Input item not permitted in request

Name	Description
MessageID	SVC2005
Text	Input %1 %2 not permitted in request
Variables	%1 – type of item, e.g., “element” or “attribute” %2 – identifier of invalid item
HTTP status code(s)	400 Bad Request

This error code indicates that the client has supplied an element, attribute, or other item which is forbidden to appear in this request, e.g., a resourceURL in a POST request.

C.1.15 SVC2006: Mandatory input item missing from request

Name	Description
MessageID	SVC2006
Text	Mandatory input %1 %2 is missing from request
Variables	%1 – type of item, e.g., “element” or “attribute” %2 – identifier of invalid item
HTTP status code(s)	400 Bad Request

This error code indicates that the client has omitted an element, attribute, or other item which is mandatory to appear in this request.

C.1.16 SVC2007: Simultaneous modification not supported

Name	Description
MessageID	SVC2007
Text	Simultaneous modification not supported
Variables	None.
HTTP status code(s)	409 Conflict

This error is given in response to a modification request. It indicates that a modification of this resource is already in progress and the server does not support multiple simultaneous modifications. The client SHOULD retry after a short delay.

C.1.17 SVC2008: Unknown resource

Name	Description
MessageID	SVC2008
Text	Unknown %1 %2
Variables	%1 – type of resource, e.g., "subscriber" or "channel" %2 – identifier supplied
HTTP status code(s)	400 Bad request, 404 Not Found

If the resource identifier is part of the request URI, the status code 404 SHOULD be used; otherwise the status code 400 SHOULD be used.

This error allows the server to report specifically which part of a URL is unrecognised. If the server does not wish to expose this information it can use SVC2002 instead.

C.2 Common Policy Exceptions

Faults related to policies associated with the service result in the return of a PolicyException message.

C.2.1 POL0001: Policy error

Name	Description
MessageID	POL0001
Text	A policy error occurred. Error code is %1
Variables	%1 Error code from service - meaningful to support, and may be documented in product documentation
HTTP status code(s)	403 Forbidden

This exception represents a general, catch-all policy error. It can be used if no more information regarding the error is available, or if it is not intended that the network shares more detailed information with the applicatiuon.

Note that an error code is an arbitrary machine-readable string that usually does not include a human-readable text. It should not be confused with the HTTP status code. If a human-readable text is to be included with the error message, POL2000 is the appropriate exception to be used. If no error code is available, specify "0" here.

C.2.2 POL0002: Privacy error

Name	Description
MessageID	POL0002
Text	Privacy verification failed for address %1, request is refused
Variables	%1 - address privacy verification failed for
HTTP status code(s)	403 Forbidden

C.2.3 POL0003: Too many addresses

Name	Description

MessageID	POL0003
Text	Too many addresses specified in message part %1
Variables	%1 - message part
HTTP status code(s)	403 Forbidden

C.2.4 POL0004: Unlimited notifications not supported

Name	Description
MessageID	POL0004
Text	Unlimited notification request not supported
Variables	None
HTTP status code(s)	403 Forbidden

C.2.5 POL0005: Too many notifications requested

Name	Description
MessageID	POL0005
Text	Too many notifications requested
Variables	None
HTTP status code(s)	403 Forbidden

C.2.6 POL0006: Groups not allowed

Name	Description
MessageID	POL0006
Text	Group specified in message part %1 not allowed
Variables	%1 - message part
HTTP status code(s)	403 Forbidden

C.2.7 POL0007: Nested groups not allowed

Name	Description
MessageID	POL0007
Text	Nested group specified in message part %1 not allowed
Variables	%1 - message part
HTTP status code(s)	403 Forbidden

C.2.8 POL0008: Charging not supported

Name	Description

MessageID	POL0008
Text	Charging is not supported
Variables	None
HTTP status code(s)	403 Forbidden

C.2.9 POL0009: Invalid frequency requested

Name	Description
MessageID	POL0009
Text	Invalid frequency requested
Variables	None
HTTP status code(s)	403 Forbidden

C.2.10 POL0010: Retention time interval expired

Name	Description
MessageID	POL0010
Text	Requested information unavailable as the retention time interval has expired.
Variables	None
HTTP status code(s)	404 Not found, 410 Gone, 403 Forbidden

In case the information that has become unavailable is addressed by a resource URL, the following applies: If the resource URL refers to a resource that has existed in the past and the server is aware of that fact, the status code 410 SHOULD be used; otherwise (if the server is not aware), the status code 404 SHOULD be used.

In all other cases, the status code 403 SHOULD be used.

C.2.11 POL0011: Media Type not supported

Name	Description
MessageID	POL0011
Text	Media type not supported
Variables	None
HTTP status code(s)	406 Not acceptable, 403 Forbidden

If the media type was passed in the HTTP Accept header, the status code MUST be 406. Otherwise, it SHOULD be 403.

Consider using POL2007 instead, so as to provide a human-readable explanation to the client of what types are supported.

C.2.12 POL0012: Too many description entries specified

Name	Description
MessageID	POL0012
Text	Too many description entries specified in message part %1

Variables	%1 – message part
HTTP status code(s)	403 Forbidden

C.2.13 POL0013: Addresses duplication

Name	Description
MessageID	POL0013
Text	Duplicated addresses
Variables	%1 – duplicated addresses
HTTP status code(s)	400 Bad request

C.2.14 POL2000: Policy Error with description

This is similar to POL0001, however, it allows a more structured error handling by communicating two variables: a machine-readable error code and an according human-readable description.

Like POL0001, this exception represents a general, catch-all policy error. It can be used if no more information regarding the error is available, or if it is not intended that the network shares more detailed information with the application.

Name	Description
MessageID	POL2000
Text	The following policy error occurred: %1. Error code is %2
Variables	%1 Textual description of the error %2 Error code
HTTP status code(s)	403 Forbidden

C.2.15 POL2001: User not provisioned for service

Name	Description
MessageID	POL2001
Text	User has not been provisioned for %1
Variables	%1 – the name of the service
HTTP response	403 Forbidden

C.2.16 POL2002: User suspended from service

Name	Description
MessageID	POL2002
Text	User has been suspended from %1
Variables	%1 – the name of the service
HTTP response	403 Forbidden

C.2.17 POL2003: Access denied

Name	Description
MessageID	POL2003
Text	Access denied
Variables	None
HTTP status code(s)	403 Forbidden

C.2.18 POL2004: File size limit exceeded

Name	Description
MessageID	POL2004
Text	File size exceeds the limit %1
Variables	%1 – file size limit
HTTP response	403 Forbidden, 413 Request Entity Too Large

C.2.19 POL2005: Maximum number of requests exceeded

Name	Description
MessageID	POL2005
Text	Maximum number of requests for a given time period is exceeded.
Variables	None
HTTP response	403 Forbidden, 429 Too Many Requests

C.2.20 POL2006: Requested feature not available

Name	Description
MessageID	POL2006
Text	Requested feature %1 not available
Variables	%1 – name of feature
HTTP response	403 Forbidden, 404 Not Found, 405 Method Not Allowed

This exception applies when a (usually OPTIONAL) feature is defined by a specification but is not available in a particular implementation or deployment. The client SHOULD NOT repeat the request.

If the resource is not available, the status code MUST be 404. If the resource is available but the method is not, the status code MUST be 405. Otherwise (i.e., if the resource and method are available but the particular request is not), it SHOULD be 403.

C.2.21 POL2007: Media Type not supported with details

Name	Description
MessageID	POL2007

Text	Media type not supported: %1
Variables	%1 – human-readable description
HTTP status code(s)	406 Not acceptable, 403 Forbidden

If the media type was passed in the HTTP Accept header, the status code MUST be 406. Otherwise, it SHOULD be 403.

C.2.22 POL2008: Too many resources requested

Name	Description
MessageID	POL2008
Text	Too many resources requested: %1
Variables	%1 – Human-readable description of the limit policy
HTTP status code(s)	403 Forbidden, 429 Too Many Requests

This error code is used in response to a resource creation request. It indicates that a policy limit on the number of such resources has been reached.

Appendix D. Authorization aspects (Normative)

This appendix specifies how to use the OMA RESTful Network APIs in combination with some authorization frameworks.

D.1 Use of Autho4API

RESTful Network APIs MAY support the authorization framework defined in [Autho4API_10].

A RESTful Network API supporting [Autho4API_10] SHALL conform to this section D.1.

As for message confidentiality and message authentication, the possible mechanism is available in [RFC2818] and related cipher suites are available in [SEC_CF-V1_1].

D.1.1 Endpoint URLs

The endpoint URL to which [Autho4API_10] compliant clients send authorization requests SHALL be constructed as follows:

```
https://{serverRoot}/autho4api/{version}/authorize
```

The endpoint URL to which [Autho4API_10] compliant clients send token requests SHALL be constructed as follows:

```
https://{serverRoot}/autho4api/{version}/token
```

The endpoint URL to which [Autho4API_10] compliant clients send token revocation requests SHALL be constructed as follows:

```
https://{serverRoot}/autho4api/{version}/revoke
```

Where the request URL variables are:

Name	Description
serverRoot	server base url: hostname+port+base path. Port and base path are OPTIONAL. Example: example.com/exampleAPI
version	version of the [Autho4API_10] framework, SHALL be “v1” without quotes

These endpoints SHALL be able to serve the requests for authorizations and tokens for any OMA RESTful Network API defining support for [Autho4API_10], and for any version of this RESTful Network API.

D.1.2 Scope values

D.1.2.1 Naming and registration

Autho4API scope values defined by OMA RESTful Network API specifications SHALL follow the `{OMAScopeValue}` grammar defined in section 7.3.1.3 of [Autho4API_10], with the additional following constraints:

Name	Description
ApiType	fixed string “rest”
ApiIdentification	identification of the OMA RESTful Network API (e.g. “messaging” without quotes)
Token	identification of a set of operations on a set of resources of this API (e.g. “out” without quotes), to be documented by the OMA RESTful Network API specification

[Autho4API_10] scope values defined by OMA RESTful Network API specifications SHALL be registered with OMNA to the OMNA Autho4API Scope Value Registry [OMNA_Autho4API].

D.1.2.2 Usage

The [Autho4API_10] compliant client SHALL include in the first request of the authorization protocol defined in [Autho4API_10] the scope parameter containing a space-delimited list of scope values belonging to OMNA Autho4API

Scope Value registry. The Authorization Server SHALL return an error response containing the error code appropriate to the protocol flow (e.g. “invalid_scope”) in the following cases:

- the scope parameter is missing;
- the requested scope is invalid, unknown, or malformed;
- the requested scope includes multiple scope values, and one of them is defined by the OMA RESTful Network API for the issuing of one-time access tokens only.

Appendix E. Deployment Considerations (Informative)

Applications using the RESTful Network APIs can be categorized by their execution environment:

- Application is a RESTful client application executing in a server execution environment (e.g. a 3rd party application).
- Application is a RESTful client application executing in a mobile device execution environment.
- Application is a RESTful client application executing in a fixed device execution environment.
- Application is a RESTful client application executing in a browser execution environment.

A RESTful Network API client can execute in any of the above execution environments.

Issues that are dependent on the execution environment and can impact strategic deployment decisions, interoperability, and scalability include (non-exhaustive list):

- Security aspects (e.g. client application authentication). As for message confidentiality and message authentication, the possible mechanism is available in [RFC2818] and related cipher suites are available in [SEC_CF-V1_1].
- Delivery of notifications from server to client application. The mechanism for delivery of notifications may depend on the execution environment of the client application. A non-exhaustive list of notifications delivery mechanisms include:
    - Notifications sent from server to client application, for example:
        i. There must be an active "listener" on the application host (in this case the client device), ready to receive the incoming notification via the HTTP protocol.
        ii. This does not have to be the application itself, but at least some host service/client which can invoke the specific application when needed.
        iii. In a client-server HTTP binding, this requires that the client has the support of an HTTP listener service.
    - Notifications retrieved by the client application using Long Polling at a Server-side Notification URL:
        i. The client must have previously created a Notification Channel to obtain a Server-side Notification URL and a URL on which to perform Long Polling [REST_Notif_Channel].

While solutions to particular issues related to the client application execution environment are out-of-scope for the RESTful Network APIs, other OMA enablers should be re-used (where applicable) to address such particular issues.

E.1 RESTful client application executing in a server execution environment

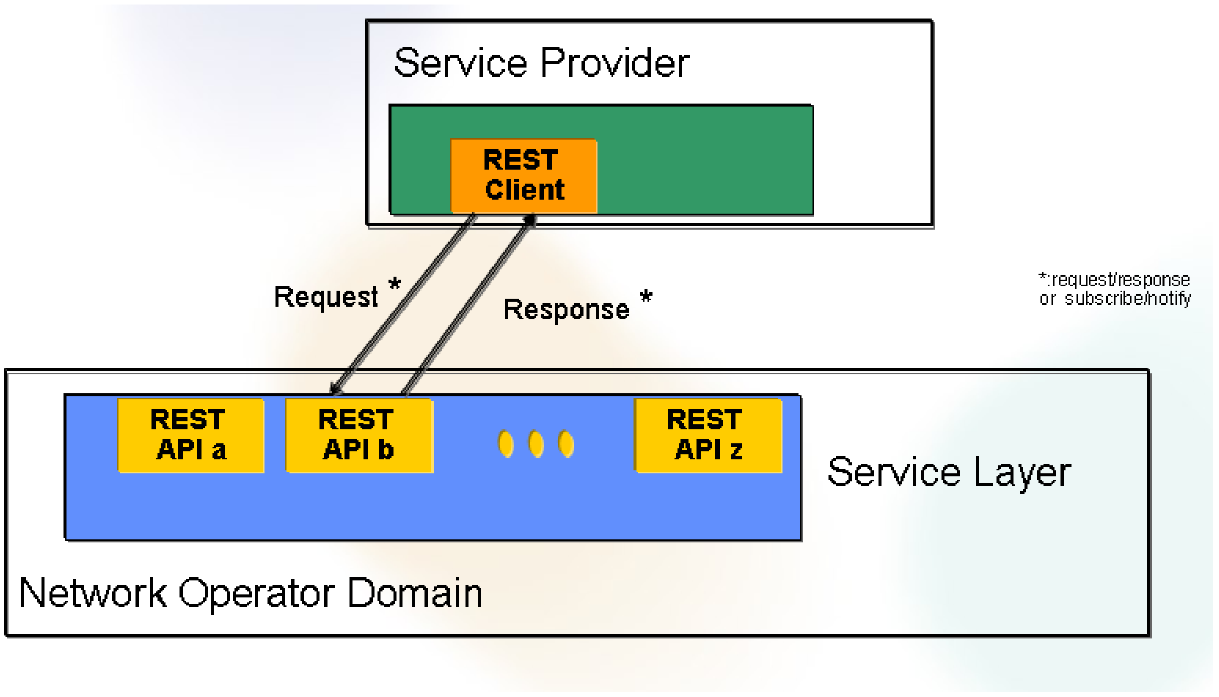

Figure 1 RESTful Network API accessed from a server execution environment (e.g. 3rd party Service Provider application)

The RESTful Network API exposed by the server deployed in the Network Operator service layer domain, may be accessed by a client application executing on a server resident in the Service Provider domain.This deployment can support all resources and operations specified in the RESTful Network APIs. There are no particular issues with support of notifications from a server to aclient application.

E.2 RESTful client application executing in a mobile device execution environment

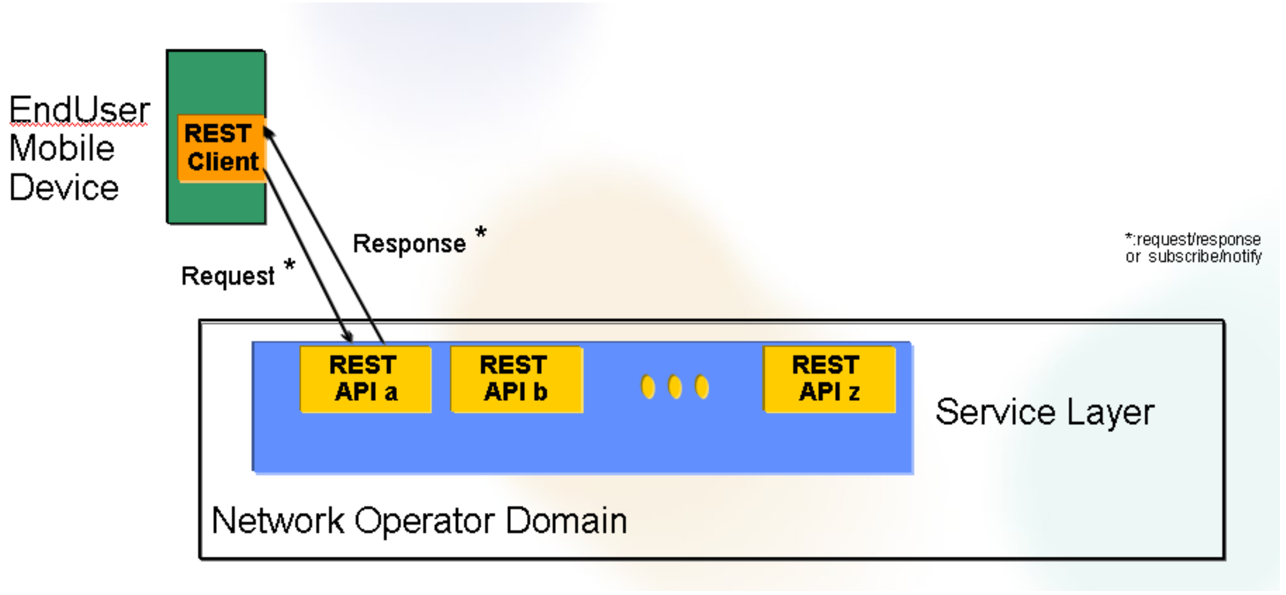

Figure 2 RESTful Network API accessed from a mobile device execution environment

The RESTful Network API exposed by a server deployed in the Network Operator service layer domain, may be accessed by a client application executing on an end user mobile device. This deployment can support most resources and operations specified in the API. There are however particular issues with support of notifications from server to client application:

- Typically in mobile devices, the client does not have the support for an HTTP listener service. The specified client notifications may have to be delivered by alternative means. OMA Push [OMA_PUSH] should be considered to be used to deliver the notifications to the client application.
- It must be possible to actually deliver the notification to the client application, i.e. there must be no boundary across which the protocol is typically blocked. In a client-server HTTP binding, this will typically be an issue as
    - The client is typically within some private network behind a firewall (e.g. PLMN Operator mobile network or home network)
    - The client does not have a fixed IP address or an IP address that is resolvable via DNS.
    - In such cases, a notification service such as OMA Push should be considered to be used to bridge the firewall border and resolve the target address of the notification to an actual client address.

E.3 RESTful client application executing in a fixed device execution environment

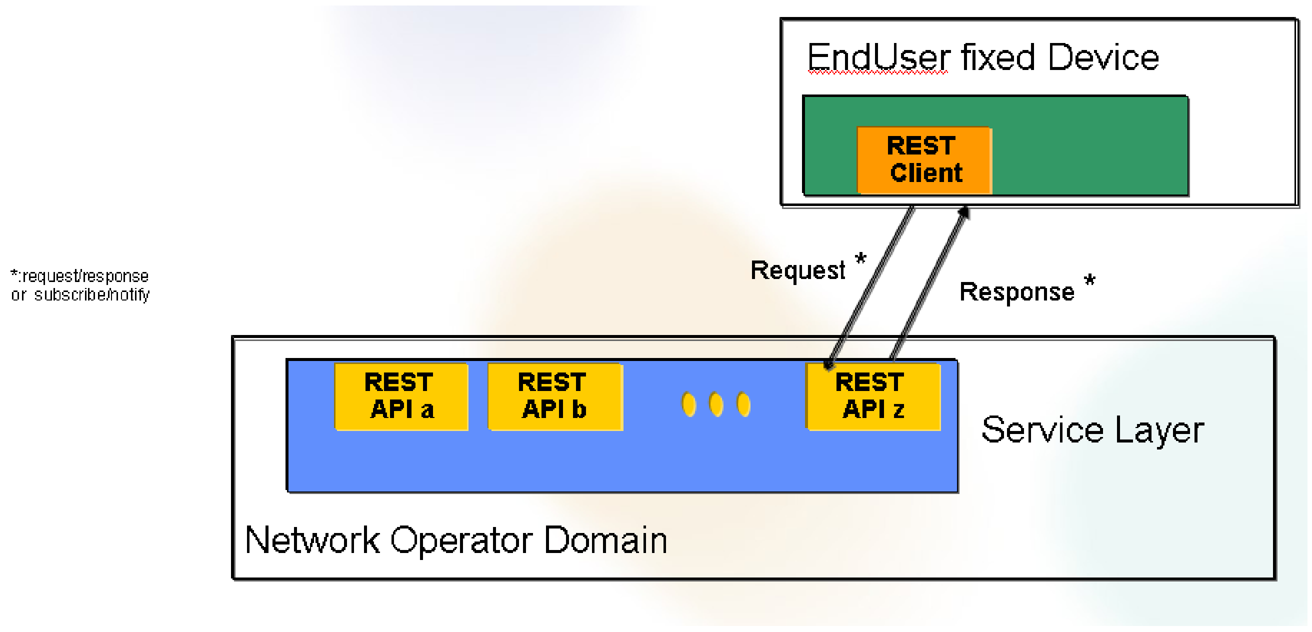

Figure 3 RESTful Network API accessed from a fixed device execution environment

The RESTful Network API exposed by a server deployed on the Network Operator service layer domain, may be accessed by a client application executing on a fixed device connected to the Network Operator.

This deployment can support most resources and operations specified in the API. Some issues with support of notifications from server to client applications may be similar to those mentioned in Appendix E.2. Solutions to those issues may however rely on other mechanisms (e.g. use of COMET).