NETMOD
Internet-Draft
Intended status: Standards Track
Expires: October 23, 2014

L. Lhotka
CZ.NIC

April 21, 2014

JSON Encoding of Data Modeled with YANG
draft-ietf-netmod-yang-json-00

Abstract

This document defines rules for representing configuration and state data defined using YANG as JSON text. It does so by specifying a procedure for translating the subset of YANG-compatible XML documents to JSON text, and vice versa. A JSON encoding of XML attributes is also defined so as to allow for including metadata in JSON documents.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 23, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

The aim of this document is define rules for representing configuration and state data defined using the YANG data modeling language [RFC6020] as JavaScript Object Notation (JSON) text [RFC7159]. The result can be potentially applied in two different ways:

1. JSON may be used instead of the standard XML [XML] encoding in the context of the NETCONF protocol [RFC6241] and/or with existing data models expressed in YANG. An example application is the RESTCONF Protocol [RESTCONF].

2. Other documents that choose JSON to represent structured data can use YANG for defining the data model, i.e., both syntactic and semantic constraints that the data have to satisfy.

JSON mapping rules could be specified in a similar way as the XML mapping rules in [RFC6020]. This would however require solving several problems. To begin with, YANG uses XPath [XPath] quite extensively, but XPath is not defined for JSON and such a definition would be far from straightforward.

In order to avoid these technical difficulties, this document employs an alternative approach: it defines a relatively simple procedure which allows for translating the subset of XML that can be modeled using YANG to JSON, and vice versa. Consequently, validation of a JSON text against a data model can done by translating the JSON text to XML, which is then validated according to the rules stated in [RFC6020].

The translation procedure is adapted to YANG specifics and requirements, namely:

1. The translation is driven by a concrete YANG data model and uses information about data types to achieve better results than generic XML-JSON translation procedures.

2. Various document types are supported, namely configuration data, configuration + state data, RPC input and output parameters, and notifications.

3. XML namespaces specified in the data model are mapped to namespaces of JSON objects. However, explicit namespace identifiers are rarely needed in JSON text.

4. Section 4 defines JSON encoding of XML attributes. Although XML attributes cannot be modeled with YANG, they are often used for attaching metadata to elements, and a standard JSON encoding is therefore needed.

5. Translation of XML mixed content, comments and processing instructions is outside the scope of this document.

Item 1 above also means that, depending on the data model, the same XML element can be translated to different JSON objects. For example,

```
<foo>123</foo>
```

is translated to

```
    "foo": 123
```

if the "foo" node is defined as a leaf with the "uint8" datatype, or to

```
    "foo": ["123"]
```

if the "foo" node is defined as a leaf-list with the "string" datatype, and the <foo> element has no siblings of the same name.

2. Terminology and Notation

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].

The following terms are defined in [RFC6020]:

- anyxml
- augment
- container
- data node
- data tree
- datatype
- feature
- identity
- instance identifier
- leaf
- leaf-list
- list
- module
- submodule

The following terms are defined in [XMLNS]:

- local name
- prefixed name
- qualified name

3. Specification of the Translation Procedure

The translation procedure defines a 1-1 correspondence between the subset of YANG-compatible XML documents and JSON text. This means that the translation can be applied in both directions and it is always invertible.

The translation procedure is applicable only to data hierarchies that are modelled by a YANG data model. An input XML document MAY contain enclosing elements representing NETCONF "Operations" and "Messages" layers. However, these enclosing elements do not appear in the resulting JSON document.

Any YANG-compatible XML document can be translated, except documents with mixed content. This is only a minor limitation since mixed content is marginal in YANG - it is allowed only in anyxml data nodes.

The following sections specify rules mainly for translating XML documents to JSON text. Rules for the inverse translation are stated only where necessary, otherwise they can be easily inferred.

REQUIRED parameters of the translation procedure are:

- YANG data model consisting of a set of YANG modules,
- type of the input document,
- optional features (defined via the "feature" statement) that are considered active.

The permissible types of input documents are listed in Table 1 together with the corresponding part of the data model that is used for the translation.

Document Type	Data Model Section
configuration and state data	main data tree
configuration	main data tree ("config true")
RPC input parameters	"input" data nodes under "rpc"
RPC output parameters	"output" data nodes under "rpc"
notification	"notification" data nodes

Table 1: YANG Document Types

When translating XML to JSON, the type of the input document can often be determined form the encapsulating elements belonging to the "Operations" or "Messages" layer as defined by the NETCONF protocol (see Sec. 1.2 in [RFC6241]).

A particular application MAY decide to support only a subset of document types from Table 1.

XML documents can be translated to JSON text only if they are valid instances of the YANG data model and selected document type, also taking into account the active features, if there are any.

The resulting JSON document is always a single object ([RFC7159], Sec. 4) whose members are translated from the original XML document using the rules specified in the following sections.

3.1. Names and Namespaces

The local part of a JSON name is always identical to the local name of the corresponding XML element.

Each JSON name lives in a namespace which is uniquely identified by the name of the YANG module where the corresponding data node is defined. If the data node is defined in a submodule, then the namespace identifier is the name of the main module to which the submodule belongs. The translation procedure MUST correctly map YANG namespace URIs to YANG module names and vice versa.

The namespace SHALL be expressed in JSON text by prefixing the local name in the following way:

```
<module name>:<local name>
```

Figure 1: Encoding a namespace identifier with a local name.

The namespace identifier MUST be used for local names that are ambiguous, i.e., whenever the data model permits a sibling data node with the same local name. Otherwise, the namespace identifier is OPTIONAL.

For example, consider the following YANG module:

```
module foomod {
    namespace "http://example.com/foomod";
    prefix "fm";
    container foo {
        leaf bar {
            type boolean;
        }
    }
}
```

If the data model consists only of this module, then the following is a valid JSON document:

```
{
  "foo": {
    "bar": true
  }
}
```

Now, assume the container "foo" is augmented from another module:

```
module barmod {
    namespace "http://example.com/barmod";
    prefix "bm";
    import foomod {
        prefix fm;
    }
    augment "/fm:foo" {
        leaf bar {
            type uint8;
        }
    }
}
```

In the data model combining "foomod" and "barmod", we have two sibling data nodes with the same local name, namely "bar". In this

case, a valid JSON document has to specify an explicit namespace identifier (module name) for both leaves:

```
{
  "foo": {
    "foomod:bar": true,
    "barmod:bar": 123
  }
}
```

3.2. Mapping XML Elements to JSON Objects

An XML element that is modelled as a YANG data node is translated to a name/value pair where the name is formed from the name of the XML element using the rules in Section 3.1. The value depends on the type of the data node as specified in the following sections.

3.2.1. The "leaf" Data Node

An XML element that is modeled as YANG leaf is translated to a name/value pair and the type of the value is derived from the YANG datatype of the leaf (see Section 3.3 for the datatype mapping rules).

Example: For the leaf node definition

```
leaf foo {
    type uint8;
}
```

the XML element

```
<foo>123</foo>
```

corresponds to the JSON name/value pair

```
"foo": 123
```

3.2.2. The "container" Data Node

An XML element that is modeled as YANG container is translated to a name/object pair.

Example: For the container definition

```
container bar {
    leaf foo {
        type uint8;
    }
}
```

the XML element

```
<bar>
  <foo>123</foo>
</bar>
```

corresponds to the JSON name/value pair

```
"bar": {
  "foo": 123
}
```

3.2.3. The "leaf-list" Data Node

A sequence of one or more sibling XML elements with the same qualified name that is modeled as YANG leaf-list is translated to a name/array pair, and the array elements are primitive values whose type depends on the datatype of the leaf-list (see Section 3.3).

Example: For the leaf-list definition

```
leaf-list foo {
    type uint8;
}
```

the XML elements

```
<foo>123</foo>
<foo>0</foo
```

correspond to the JSON name/value pair

```
"foo": [123, 0]
```

3.2.4. The "list" Data Node

A sequence of one or more sibling XML elements with the same qualified name that is modeled as YANG list is translated to a name/array pair, and the array elements are JSON objects.

Unlike the XML encoding, where the list keys are required to come before any other siblings, and in the order specified by the data

model, the order of members within a JSON list entry is arbitrary, because JSON objects are fundamentally unordered collections of members.

Example: For the list definition

```
list bar {
    key foo;
    leaf foo {
        type uint8;
    }
    leaf baz {
        type string;
    }
}
```

the XML elements

```
<bar>
  <foo>123</foo>
  <baz>zig</baz>
</bar>
<bar>
  <foo>0</foo>
  <baz>zag</baz>
</bar>
```

correspond to the JSON name/value pair

```
"bar": [
  {
    "foo": 123,
    "baz": "zig"
  },
  {
    "foo": 0,
    "baz": "zag"
  }
]
```

3.2.5. The "anyxml" Data Node

An XML element that is modeled as a YANG anyxml data node is translated to a name/object pair. The content of such an element is not modelled by YANG, and there may not be a straightforward mapping to JSON text (e.g., if it is a mixed XML content). Therefore, translation of anyxml contents is necessarily application-specific and outside the scope of this document.

Example: For the anyxml definition

```
anyxml bar;
```

the XML element

```
<bar>
  <p xmlns="http://www.w3.org/1999/xhtml">
    This is <em>very</em> cool.
  </p>
</bar>
```

may be translated to the following JSON name/value pair:

```
{
  "bar": {
    "p": "This is *very* cool."
  }
}
```

3.3. Mapping YANG Datatypes to JSON Values

3.3.1. Numeric Datatypes

A value of one of the YANG numeric datatypes ("int8", "int16", "int32", "int64", "uint8", "uint16", "uint32", "uint64" and "decimal64") is mapped to a JSON number using the same lexical representation.

3.3.2. The "string" Type

A "string" value is mapped to an identical JSON string, subject to JSON encoding rules.

3.3.3. The "boolean" Type

A "boolean" value is mapped to the corresponding JSON value ’true’ or ’false’.

3.3.4. The "enumeration" Type

An "enumeration" value is mapped in the same way as a string except that the permitted values are defined by "enum" statements in YANG.

3.3.5. The "bits" Type

A "bits" value is mapped to a string identical to the lexical representation of this value in XML, i.e., space-separated names representing the individual bit values that are set.

3.3.6. The "binary" Type

A "binary" value is mapped to a JSON string identical to the lexical representation of this value in XML, i.e., base64-encoded binary data.

3.3.7. The "leafref" Type

A "leafref" value is mapped according to the same rules as the type of the leaf being referred to.

3.3.8. The "identityref" Type

An "identityref" value is mapped to a string representing the qualified name of the identity. Its namespace MAY be expressed as shown in Figure 1. If the namespace part is not present, the namespace of the name of the JSON object containing the value is assumed.

3.3.9. The "empty" Type

An "empty" value is mapped to '[null]', i.e., an array with the 'null' value being its only element.

This encoding was chosen instead of using simply 'null' in order to facilitate the use of empty leafs in common programming languages. When used in a boolean context, the '[null]' value, unlike 'null', evaluates to 'true'.

Example: For the leaf definition

```
leaf foo {
    type empty;
}
```

the XML element

```
<foo/>
```

corresponds to the JSON name/value pair

```
"foo": [null]
```

3.3.10. The "union" Type

YANG "union" type represents a choice among multiple alternative types. The actual type of the XML value MUST be determined using the procedure specified in Sec. 9.12 of [RFC6020] and the mapping rules for that type are used.

For example, consider the following YANG definition:

```
leaf-list bar {
    type union {
        type uint16;
        type string;
    }
}
```

The sequence of three XML elements

```
<bar>6378</bar>
<bar>14.5</bar>
<bar>infinity</bar>
```

will then be translated to this name/array pair:

```
"bar": [6378, "14.5", "infinity"]
```

3.3.11. The "instance-identifier" Type

An "instance-identifier" value is a string representing a simplified XPath specification. It is mapped to an analogical JSON string in which all occurrences of XML namespace prefixes are either removed or replaced with the corresponding module name according to the rules of Section 3.1.

When translating such a value from JSON to XML, all components of the instance-identifier MUST be given appropriate XML namespace prefixes. It is RECOMMENDED that these prefixes be those defined via the "prefix" statement in the corresponding YANG modules.

For example, assume "ex" is the prefix defined for the "example" module. Then the XML-encoded instance identifier

```
/ex:system/ex:user[ex:name='fred']
```

corresponds to the following JSON-encoded instance identifier:

```
/example:system/example:user[example:name='fred']
```

or simply

```
/system/user[name='fred']
```

if the local names of the data nodes "system", "user" and "name" are unambiguous.

4. Encoding Metadata in JSON

By design, YANG does not allow for modeling XML attributes. However, attributes are often used in XML instance documents for attaching various types of metadata information to elements. It is therefore desirable to have a standard way for representing attributes in JSON documents as well.

The metadata encoding defined in the rest of this section satisfies the following two important requirements:

1. There has to be a way for adding metadata to instances of all types of YANG data nodes, i.e., leafs, containers, list and leaf-list entries, and anyxml nodes.

2. The encoding of YANG data node instances as defined in the previous sections must not change.

Existing proposals for metadata encoding in JSON, such as [JSON-META], are oriented on rather specific uses of metadata, and fall short with respect to the first requirement.

All attributes assigned to an XML element are mapped in JSON to members (name/value pairs) of a single object, henceforth denoted as the metadata object. The placement of this object depends on the type of the element from YANG viewpoint, as specified in the following paragraphs.

For an XML element that is translated to a JSON object (i.e., a container, anyxml node and list entry), the metadata object is added as a new member of that object with the name "@".

Examples:

- If "cask" is a container or anyxml node, the XML instance with attributes

```
<cask foo="a" bar="b">
  ...
</cask>
```

is mapped to the following JSON object:

```
"cask": {
  "@": {
    "foo": "a",
    "bar": "b"
  }
  ...
}
```

o If "seq" is a list, then the pair of XML elements

```
<seq foo="a">
  <name>one</name>
</seq>
<seq bar="b">
  <name>two</name>
</seq>
```

is mapped to the following JSON array:

```
"seq": [
  {
    "@": {
      "foo": "a"
    },
    "name": "one"
  },
  {
    "@": {
      "bar": "b"
    },
    "name": "two"
  }
]
```

In order to assign attributes to a leaf instance, a sibling name/value pair is added, where the name is the symbol "@" concatenated with the identifier of the leaf.

For example, the element

```
<flag foo="a" bar="b">true</foo>
```

is mapped to the following two name/value pairs:

```
    "flag": true,
    "@flag": {
      "foo": "a",
      "bar": "b"
    }
```

Finally, for a leaf-list instance, which is represented as a JSON array with primitive values, attributes may be assigned to one or more entries by adding a sibling name/value pair, where the name is the symbol "@" concatenated with the identifier of the leaf-list, and the value is a JSON array whose i-th element is the metadata object with attributes assigned to the i-th entry of the leaf-list, or nil if the i-th entry has no attributes.

Trailing nil values in the array, i.e., those following the last non-nil metadata object, MAY be omitted.

For example, a leaf-list instance with four entries

```
    <folio>6</folio>
    <folio foo="a">3</folio>
    <folio bar="b">7</folio>
    <folio>8</folio>
```

is mapped to the following two name/value pairs:

```
    "folio": [6, 3, 7, 8],
    "@folio": [nil, {"foo": "a"}, {"bar": "b"}]
```

The encoding of attributes as specified above has the following two limitations:

- Mapping of namespaces of XML attributes is undefined.
- Attribute values can only be strings, other data types are not supported.

5. IANA Considerations

TBD - register application/yang.data+json media type?

6. Security Considerations

TBD.

7. Acknowledgments

The author wishes to thank Andy Bierman, Martin Bjorklund and Phil Shafer for their helpful comments and suggestions.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for Network Configuration Protocol (NETCONF)", RFC 6020, September 2010.

[RFC6241] Enns, R., Bjorklund, M., Schoenwaelder, J., and A. Bierman, "NETCONF Configuration Protocol", RFC 6241, June 2011.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[XMLNS] Bray, T., Hollander, D., Layman, A., Tobin, R., and H. Thompson, "Namespaces in XML 1.0 (Third Edition)", World Wide Web Consortium Recommendation REC-xml-names-20091208, December 2009, <http://www.w3.org/TR/2009/REC-xml-names-20091208>.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, November 2008, <http://www.w3.org/TR/2006/REC-xml-20060816>.

8.2. Informative References

[IF-CFG] Bjorklund, M., "A YANG Data Model for Interface Management", draft-ietf-netmod-interfaces-cfg-16 (work in progress), January 2014.

[JSON-META] Sakimura, N., "JSON Metadata", draft-sakimura-json-metadata-01 (work in progress), November 2013.

[RESTCONF] Bierman, A., Bjorklund, M., Watsen, K., and R. Fernando, "RESTCONF Protocol", draft-ietf-netconf-restconf-00 (work in progress), March 2014.

[XPath] Clark, J., "XML Path Language (XPath) Version 1.0", World Wide Web Consortium Recommendation REC-xpath-19991116, November 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>.

Appendix A. A Complete Example

The JSON document shown below was translated from a reply to the NETCONF <get> request that can be found in Appendix D of [IF-CFG]. The data model is a combination of two YANG modules: "ietf-interfaces" and "ex-vlan" (the latter is an example module from Appendix C of [IF-CFG]). The "if-mib" feature defined in the "ietf-interfaces" module is considered to be active.

```
{
  "interfaces": {
    "interface": [
      {
        "name": "eth0",
        "type": "iana-if-type:ethernetCsmacd",
        "enabled": false
      },
      {
        "name": "eth1",
        "type": "iana-if-type:ethernetCsmacd",
        "enabled": true,
        "vlan-tagging": true
      },
      {
        "name": "eth1.10",
        "type": "iana-if-type:l2vlan",
        "enabled": true,
        "base-interface": "eth1",
        "vlan-id": 10
      },
      {
        "name": "lo1",
        "type": "iana-if-type:softwareLoopback",
        "enabled": true
      }
    ]
  },
  "interfaces-state": {
```

```
      "interface": [
        {
          "name": "eth0",
          "type": "iana-if-type:ethernetCsmacd",
          "admin-status": "down",
          "oper-status": "down",
          "if-index": 2,
          "phys-address": "00:01:02:03:04:05",
          "statistics": {
            "discontinuity-time": "2013-04-01T03:00:00+00:00"
          }
        },
        {
          "name": "eth1",
          "type": "iana-if-type:ethernetCsmacd",
          "admin-status": "up",
          "oper-status": "up",
          "if-index": 7,
          "phys-address": "00:01:02:03:04:06",
          "higher-layer-if": [
            "eth1.10"
          ],
          "statistics": {
            "discontinuity-time": "2013-04-01T03:00:00+00:00"
          }
        },
        {
          "name": "eth1.10",
          "type": "iana-if-type:l2vlan",
          "admin-status": "up",
          "oper-status": "up",
          "if-index": 9,
          "lower-layer-if": [
            "eth1"
          ],
          "statistics": {
            "discontinuity-time": "2013-04-01T03:00:00+00:00"
          }
        },
        {
          "name": "eth2",
          "type": "iana-if-type:ethernetCsmacd",
          "admin-status": "down",
          "oper-status": "down",
          "if-index": 8,
          "phys-address": "00:01:02:03:04:07",
          "statistics": {
            "discontinuity-time": "2013-04-01T03:00:00+00:00"
```

```
          }
        },
        {
          "name": "lo1",
          "type": "iana-if-type:softwareLoopback",
          "admin-status": "up",
          "oper-status": "up",
          "if-index": 1,
          "statistics": {
            "discontinuity-time": "2013-04-01T03:00:00+00:00"
          }
        }
      ]
    }
  }
```

Author’s Address

Ladislav Lhotka
CZ.NIC

Email: lhotka@nic.cz