NETMOD Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 31, 2015

L. Lhotka
CZ.NIC

November 27, 2014

JSON Encoding of Data Modeled with YANG
draft-ietf-netmod-yang-json-02

Abstract

This document defines encoding rules for representing configuration, state data, RPC input and output parameters, and notifications defined using YANG as JavaScript Object Notation (JSON) text.

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 May 31, 2015.

Table of Contents

1. Introduction

The NETCONF protocol [RFC6241] uses XML [W3C.REC-xml-20081126] for encoding data in its Content Layer. Other management protocols might want to use other encodings while still benefiting from using YANG [RFC6020] as the data modeling language.

For example, the RESTCONF protocol [I-D.ietf-netconf-restconf] supports two encodings: XML (media type "application/yang.data+xml") and JSON (media type "application/yang.data+json).

The specification of the YANG data modelling language [RFC6020] defines only XML encoding for data instances, i.e. contents of configuration datastores, state data, RFC input and output parameters, and event notifications. The aim of this document is to define rules for encoding the same data as JavaScript Object Notation (JSON) text [RFC7159].

In order to achieve maximum interoperability while allowing implementations to use a variety of available JSON parsers, the JSON encoding rules follow, as much as possible, the constraints of the I-JSON restricted profile [I-D.ietf-json-i-json]. Section Section 7 discusses I-JSON conformance in more detail.

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
- identity
- instance identifier
- leaf
- leaf-list
- list
- module
- submodule

3. Validation of JSON-encoded Instance Data

Instance data validation as defined in [RFC6020] is only applicable to XML-encoded data. For one, semantic constraints in "must"

statements are expressed using XPath 1.0 [W3C.REC-xpath-19991116], which can be properly interpreted only in the XML context.

This document along with the corresponding "XML Mapping Rules" sections from [RFC6020] also define an implicit schema-driven mapping of JSON-encoded instances to XML-encoded instances (and vice versa). This mapping is mostly straightforward. In cases where doubts could arise, this document gives explicit instructions for mapping JSON-encoded instances to XML.

In order to validate a JSON instance document, it MUST first be mapped, at least conceptually, to the corresponding XML instance document. By definition, the JSON document is then valid if and only if the XML document is valid according to the rules stated in [RFC6020].

4. Names and Namespaces

Instances of YANG data nodes (leafs, containers, leaf-lists, lists and anyxml nodes) are always encoded as members of a JSON object, i.e., as name/value pairs. This section defines how the name part is formed, and the following sections deal with the value part.

Except in the cases specified below, the member name is identical to the identifier of the corresponding YANG data node. Every such name belongs to a namespace which is associated with the YANG module where the corresponding data node is defined. If the data node is defined in a submodule, then the namespace is determined by the main module to which the submodule belongs.

If the namespace of a member name has to be explicitly specified, the module name SHALL be used as a prefix to the (local) member name. Both parts of the member name SHALL be separated with a colon character (":"). In other words, the namespace-qualified name will have the following form:

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

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

Names with namespace identifiers in the form shown in Figure 1 MUST be used for all top-level YANG data nodes, and also for all nodes whose parent node belongs to a different namespace. Otherwise, names with namespace identifiers MUST NOT be used.

For example, consider the following YANG module:

```
module foomod {

  namespace "http://example.com/foomod";

  prefix "foo";

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

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

```
{
  "foomod:top": {
    "foo": 54
  }
}
```

Note that the top-level container instance contains the namespace identifier (module name) but the "foo" leaf doesn't because it is defined in the same module as its parent container.

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

```
module barmod {

  namespace "http://example.com/barmod";

  prefix "bar";

  import foomod {
    prefix "foo";
  }

  augment "/foo:top" {
    leaf bar {
      type boolean;
    }
  }
}
```

A valid JSON-encoded configuration containing both leafs may then look like this:

```
{
  "foomod:top": {
    "foo": 54,
    "barmod:bar": true
  }
}
```

The name of the "bar" leaf must be prefixed with the namespace identifier because its parent is defined in a different module, hence it belongs to another namespace.

Explicit namespace identifiers are sometimes needed when encoding values of the "identityref" and "instances-identifier" types. The same form as shown in Figure 1 is then used as well. See Sections 6.8 and 6.11 for details.

5. Encoding of YANG Data Node Instances

Every complete JSON instance document, such as a configuration datastore content, is an object. Its members are instances of all top-level data nodes defined by the YANG data model.

Character encoding MUST be UTF-8.

Any data node instance is encoded as a name/value pair where the name is formed from the data node identifier using the rules of Section 4. The value depends on the category of the data node as explained in the following subsections.

5.1. The "leaf" Data Node

A leaf instance is encoded as a name/value pair where the value can be a string, number, literal "true" or "false", or the special array "[null]", depending on the type of the leaf (see Section 6 for the type encoding rules).

Example: For the leaf node definition

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

the following is a valid JSON-encoded instance:

```
"foo": 123
```

5.2. The "container" Data Node

An container instance is encoded as a name/object pair. The container's child data nodes are encoded as members of the object.

Example: For the container definition

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

the following is a valid instance:

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

5.3. The "leaf-list" Data Node

A leaf-list is encoded as a name/array pair, and the array elements are values of the same type, which can be a string, number, literal "true" or "false", or the special array "[null]", depending on the type of the leaf-list (see Section 6 for the type encoding rules).

The order of array elements MUST be the same as the order of XML elements representing leaf-list entries in the XML encoding.

Example: For the leaf-list definition

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

the following is a valid instance:

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

5.4. The "list" Data Node

A list instance is encoded as a name/array pair, and the array elements are JSON objects.

The order of array elements MUST be the same as the order of XML elements representing list entries in the XML encoding.

Unlike the XML encoding, where list keys are required to precede any other siblings, and to appear in the order specified by the data model, the order of members within a JSON-encoded 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 following is a valid instance:

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

5.5. The "anyxml" Data Node

An anyxml instance is encoded as a name/value pair. The value can be of any valid JSON type, i.e. an object, array, number, string or one of the literals "true", "false" and "null".

This document defines no mapping between the contents of JSON- and XML-encoded anyxml instances. Note that the mapping is not needed for the purposes of validation (Section 3) because anyxml contents are not subject to YANG-based validation (see sec. 7.10 in [RFC6020]).

Example: For the anyxml definition

```
anyxml bar;
```

the following is a valid instance:

```
"bar": [true, null, true]
```

6. The Mapping of YANG Data Types to JSON Values

The type of the JSON value in an instance of the leaf or leaf-list data node depends on the type of that data node as specified in the following subsections.

6.1. Numeric Types

A value of the "int8", "int16", "int32", "uint8", "uint16" and "uint32" is represented as a JSON number.

A value of the "int64", "uint64" or "decimal64" type is encoded as a JSON string whose contents is the lexical representation of that numeric value as specified in sections 9.2.1 and 9.3.1 of [RFC6020].

For example, if the type of the leaf "foo" in Section 5.1 was "uint64" instead of "uint8", the instance would have to be encoded as

```
"foo": "123"
```

The special handling of 64-bit numbers follows from I-JSON recommendation to encode numbers exceeding the IEEE 754-2000 double precision range as strings, see sec. 2.2 in [I-D.ietf-json-i-json].

6.2. The "string" Type

A "string" value encoded as a JSON string, subject to JSON encoding rules.

6.3. The "boolean" Type

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

6.4. 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. See sec. 9.6 in [RFC6020].

6.5. The "bits" Type

A "bits" value is mapped to a JSON string identical to the lexical representation of this value in XML, i.e., space-separated names representing the individual bit values that are set. See sec. 9.7 in [RFC6020].

6.6. 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. See sec. 9.8 in [RFC6020].

6.7. The "leafref" Type

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

6.8. The "identityref" Type

An "identityref" value is mapped to a string representing the name of an identity. Its namespace MUST be expressed as shown in Figure 1 if it is different from the namespace of the leaf node containing the identityref value, and MAY be expressed otherwise.

For example, consider the following schematic module:

```
module exmod {
  ...
  import ietf-interfaces {
    prefix if;
  }
  import iana-if-type {
    prefix ianaift;
  }
  ...
  leaf type {
    type identityref {
      base "if:interface-type";
    }
  }
}
```

A valid instance of the "type" leaf is then encoded as follows:

```
"type": "iana-if-type:ethernetCsmacd"
```

The namespace identifier "iana-if-type" must be present in this case because the "ethernetCsmacd" identity is not defined in the same module as the "type" leaf.

6.9. The "empty" Type

An "empty" value is mapped to "[null]", i.e., an array with the "null" literal 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;
}
```

a valid instance is

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

6.10. The "union" Type

A value of the "union" type is encoded as the value of any of the member types.

Unlike XML, JSON conveys part of the type information already in the encoding. When validating a value of the "union" type, this information MUST also be taken into account.

For example, consider the following YANG definition:

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

In RESTCONF [I-D.ietf-netconf-restconf], it is fully acceptable to set the value of "bar" in the following way when using the "application/yang.data+xml" media type:

```
<bar>13.5</bar>
```

because the value may be interpreted as a string, i.e., the second member type of the union. When using the "application/yang.data+json" media type, however, this is an error:

```
"bar": 13.5
```

In this case, the JSON encoding indicates the value is supposed to be a number rather than string.

6.11. The "instance-identifier" Type

An "instance-identifier" value is encoded as a string that is analogical to the lexical representation in XML encoding, see sec. 9.13.3 in [RFC6020]. However, the encoding of namespaces in instance-identifier values follows the rules stated in Section 4, namely:

- The namespace identifier is the module name where each data node is defined.
- The encoding of a node name with an explicit namespace is as shown in Figure 1.
- The leftmost (top-level) node name is always prefixed with the namespace identifier.
- Any subsequent node name has the namespace identifier if and only if its parent node has a different namespace. This also holds for node names appearing in predicates.

For example,

```
/ietf-interfaces:interfaces/interface[name='eth0']/ietf-ip:ipv4/ip
```

is a valid instance-identifer value because the data nodes "interfaces", "interface" and "name" are defined in the module "ietf-interfaces", whereas "ipv4" and "ip" are defined in "ietf-ip".

When translating an instance-identifier value from JSON to XML, the namespace identifier (YANG module name) in each component of the instance-identifier MUST be replaced by an XML namespace prefix that is associated with the namespace URI reference of the module in the scope of the element containing the instance-identifier value.

7. I-JSON Compliance

I-JSON [I-D.ietf-json-i-json] is a restricted profile of JSON that guarantees maximum interoperability for protocols that use JSON in their messages, no matter what JSON encoders/decoders are used in protocol implementations. The encoding defined in this document therefore observes the I-JSON requirements and recommendations as closely as possible.

In particular, the following properties are guaranteed:

- Character encoding is UTF-8.
- Member names within the same JSON object are always unique.
- The order of JSON object members is never relied upon.
- Numbers of any type supported by YANG can be exchanged reliably. See Section 6.1 for details.

The only two cases where a JSON instance document encoded according to this document may deviate from I-JSON were dictated by the need to be able to encode the same instance data in both JSON and XML. These two exceptions are:

- Leaf values encoded as strings may contain code points identifying Noncharacters that belong to the XML character set (see sec. 2.2 in [W3C.REC-xml-20081126]). This issue is likely to be solved in YANG 1.1 because noncharacters will not be allowed in string values, see sec. 9.4 in [I-D.ietf-netmod-rfc6020bis].
- Values of the "binary" type are encoded with the base64 encoding scheme (Section 6.6), whereas I-JSON recommends base64url instead. Theoretically, values of the "binary" type might appear in URI references, such as Request-URI in RESTCONF, although in practice the cases where it is really needed should be extremely rare.

8. Security Considerations

This document defines an alternative encoding for data modeled in the YANG data modeling language. As such, it doesn't contribute any new security issues beyond those discussed in sec. 15 of [RFC6020].

JSON is rather different from XML, and JSON parsers may thus suffer from other types of vulnerabilities than their XML counterparts. To minimize these security risks, it is important that client and server software supporting JSON encoding behaves as required in sec. 3 of [I-D.ietf-json-i-json]. That is, any received JSON data that violate any of I-JSON strict constraints MUST NOT be trusted nor acted upon. Violations due to the presence of Unicode Noncharacters in the data (see Section 7) SHOULD be carefully examined.

9. Acknowledgments

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

10. References

10.1. Normative References

[I-D.ietf-json-i-json]
Bray, T., "The I-JSON Message Format", draft-ietf-json-i-json-03 (work in progress), August 2014.

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

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

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

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

[W3C.REC-xml-20081126]
Bray, T., Paoli, J., Sperberg-McQueen, M., 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/2008/REC-xml-20081126>.

10.2. Informative References

[I-D.ietf-netconf-restconf]
Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", draft-ietf-netconf-restconf-03 (work in progress), October 2014.

[I-D.ietf-netmod-rfc6020bis]
Bjorklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", draft-ietf-netmod-rfc6020bis-02 (work in progress), November 2014.

[RFC7223] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 7223, May 2014.

[W3C.REC-xpath-19991116]
Clark, J. and S. DeRose, "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 represents the same data as the reply to the NETCONF <get> request appearing in Appendix D of [RFC7223]. 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 [RFC7223]). The "if-mib" feature defined in the "ietf-interfaces" module is considered to be active.

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

Appendix B. Change Log

RFC Editor: Remove this section upon publication as an RFC.

B.1. Changes Between Revisions -01 and -02

- Encoding of namespaces in instance-identifiers was changed.
- Text specifying the order of array elements in leaf-list and list instances was added.

B.2. Changes Between Revisions -00 and -01

- Metadata encoding was moved to a separate I-D, draft-lhotka-netmod-yang-metadata.
- JSON encoding is now defined directly rather than via XML-JSON mapping.
- The rules for namespace encoding has changed. This affect both node instance names and instance-identifiers.
- I-JSON-related changes. The most significant is the string encoding of 64-bit numbers.
- When validating union type, the partial type info present in JSON encoding is taken into account.
- Added section about I-JSON compliance.
- Updated the example in appendix.
- Wrote Security Considerations.
- Removed IANA Considerations as there are none.

Author's Address

Ladislav Lhotka
CZ.NIC

Email: lhotka@nic.cz