Internet Engineering Task Force (IETF)
Request for Comments: 7951
Category: Standards Track
ISSN: 2070-1721

L. Lhotka
CZ.NIC

August 2016

# JSON Encoding of Data Modeled with YANG

## Abstract

This document defines encoding rules for representing configuration data, state data, parameters of Remote Procedure Call (RPC) operations or actions, and notifications defined using YANG as JavaScript Object Notation (JSON) text.

## Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7951.

Table of Contents

## 1. Introduction

The Network Configuration Protocol (NETCONF) [RFC6241] uses XML [XML] for encoding data in its Content Layer. Other management protocols might want to use other encodings while still benefiting from using YANG [RFC7950] as the data modeling language.

For example, the RESTCONF protocol [RESTCONF] supports two encodings: XML (media type "application/yang.data+xml") and JavaScript Object Notation (JSON) (media type "application/yang.data+json").

The specification of the YANG 1.1 data modeling language [RFC7950] defines only XML encoding of data trees, i.e., configuration data, state data, input/output parameters of Remote Procedure Call (RPC) operations or actions, and notifications. The aim of this document is to define rules for encoding the same data as JSON text [RFC7159].

## 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 [RFC7950]:

- action
- anydata
- anyxml
- augment
- container
- data node
- data tree
- identity
- instance identifier
- leaf
- leaf-list
- list

- module
- RPC operation
- submodule

The following terms are defined in [RFC6241]:

- configuration data
- notification
- state data

## 3. Properties of the JSON Encoding

This document defines JSON encoding for YANG data trees and their subtrees. It is always assumed that the top-level structure in JSON-encoded data is an object.

Instances of YANG data nodes (leafs, containers, leaf-lists, lists, anydata nodes, and anyxml nodes) are encoded as members of a JSON object, i.e., name/value pairs. Section 4 defines how the name part is formed, and the following sections deal with the value part. The encoding rules are identical for all types of data trees, i.e., configuration data, state data, parameters of RPC operations, actions, and notifications.

With the exception of "anydata" encoding (Section 5.5), all rules in this document are also applicable to YANG 1.0 [RFC6020].

Unlike XML element content, JSON values carry partial type information (number, string, boolean). The JSON encoding is defined so that this information is never in conflict with the data type of the corresponding YANG leaf or leaf-list.

With the exception of anyxml and schema-less anydata nodes, it is possible to map a JSON-encoded data tree to XML encoding as defined in [RFC7950], and vice versa. However, such conversions require the YANG data model to be available.

In order to achieve maximum interoperability while allowing implementations to use a variety of existing JSON parsers, the JSON encoding rules follow, as much as possible, the constraints of the I-JSON (Internet JSON) restricted profile [RFC7493]. Section 7 discusses I-JSON conformance in more detail.

## 4. Names and Namespaces

A JSON object member name MUST be in one of the following forms:

- simple - identical to the identifier of the corresponding YANG data node.

- namespace-qualified - the data node identifier is prefixed with the name of the module in which the data node is defined, separated from the data node identifier by the colon character (":").

The name of a module determines the namespace of all data node names defined in that module. If a data node is defined in a submodule, then the namespace-qualified member name uses the name of the main module to which the submodule belongs.

ABNF syntax [RFC5234] of a member name is shown in Figure 1, where the production for "identifier" is defined in Section 14 of [RFC7950].

```
member-name = [identifier ":"] identifier
```

Figure 1: ABNF Production for a JSON Member Name

A namespace-qualified member name MUST be used for all members of a top-level JSON object and then also whenever the namespaces of the data node and its parent node are different. In all other cases, the simple form of the member name MUST be used.

For example, consider the following YANG module:

```
module example-foomod {

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

  prefix "foomod";

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

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

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

Note that the member of the top-level object uses the namespace-qualified name but the "foo" leaf doesn't because it is defined in the same module as its parent container "top".

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

```
module example-barmod {

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

  prefix "barmod";

  import example-foomod {
    prefix "foomod";
  }

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

Valid JSON-encoded configuration data containing both leafs may then look like this:

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

The name of the "bar" leaf is prefixed with the namespace identifier because its parent is defined in a different module.

Explicit namespace identifiers are sometimes needed when encoding values of the "identityref" and "instance-identifier" types. The same form of namespace-qualified name as defined above is then used. See Sections 6.8 and 6.11 for details.

## 5. Encoding of YANG Data Node Instances

Every 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.

Character encoding MUST be UTF-8.

### 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

A 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 JSON-encoded 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 some scalar 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 ordering of array elements follows the same rules as the ordering of XML elements representing leaf-list entries in the XML encoding. Specifically, the "ordered-by" properties (Section 7.7.7 in [RFC7950]) MUST be observed.

Example: For the leaf-list definition

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

the following is a valid JSON-encoded 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 ordering of array elements follows the same rules as the ordering of XML elements representing list entries in the XML encoding. Specifically, the "ordered-by" properties (Section 7.7.7 in [RFC7950]) MUST be observed.

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

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

## 5.5. The "anydata" Data Node

The anydata data node serves as a container for an arbitrary set of nodes that otherwise appear as normal YANG-modeled data. A data model for anydata content may or may not be known at runtime. In the latter case, converting JSON-encoded instances to the XML encoding defined in [RFC7950] may be impossible.

An anydata instance is encoded in the same way as a container, i.e., as a name/object pair. The requirement that anydata content can be modeled by YANG implies the following rules for the JSON text inside the object:

- It is valid I-JSON [RFC7493].
- All object member names satisfy the ABNF production in Figure 1.
- Any JSON array contains either only unique scalar values (as a leaf-list; see Section 5.3) or only objects (as a list; see Section 5.4).
- The "null" value is only allowed in the single-element array "[null]" corresponding to the encoding of the "empty" type; see Section 6.9.

Example: For the anydata definition

```
anydata data;
```

the following is a valid JSON-encoded instance:

```
"data": {
  "ietf-notification:notification": {
    "eventTime": "2014-07-29T13:43:01Z",
    "example-event:event": {
      "event-class": "fault",
      "reporting-entity": {
        "card": "Ethernet0"
      },
      "severity": "major"
    }
  }
}
```

## 5.6. The "anyxml" Data Node

An anyxml instance is encoded as a JSON name/value pair. The value MUST satisfy I-JSON constraints.

Example: For the anyxml definition

```
anyxml bar;
```

the following is a valid JSON-encoded instance:

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

### 5.7. Metadata Objects

Apart from instances of YANG data nodes, a JSON document MAY contain special object members whose name starts with the "@" character (COMMERCIAL AT). Such members are used for special purposes, such as encoding metadata [RFC7952]. The exact syntax and semantics of such members are outside the scope of this document.

## 6. Representing YANG Data Types in 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", or "uint32" type is represented as a JSON number.

A value of the "int64", "uint64", or "decimal64" type is represented as a JSON string whose content is the lexical representation of the corresponding YANG type as specified in Sections 9.2.1 and 9.3.1 of [RFC7950].

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 the I-JSON recommendation to encode numbers exceeding the IEEE 754-2008 double-precision range [IEEE754-2008] as strings; see Section 2.2 in [RFC7493].

### 6.2. The "string" Type

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

### 6.3. The "boolean" Type

A "boolean" value is represented as the corresponding JSON literal name "true" or "false".

### 6.4. The "enumeration" Type

An "enumeration" value is represented as a JSON string -- one of the names assigned by "enum" statements in YANG.

The representation is identical to the lexical representation of the "enumeration" type in XML; see Section 9.6 in [RFC7950].

### 6.5. The "bits" Type

A "bits" value is represented as a JSON string -- a space-separated sequence of names of bits that are set. The permitted bit names are assigned by "bit" statements in YANG.

The representation is identical to the lexical representation of the "bits" type; see Section 9.7 in [RFC7950].

### 6.6. The "binary" Type

A "binary" value is represented as a JSON string -- base64 encoding of arbitrary binary data.

The representation is identical to the lexical representation of the "binary" type in XML; see Section 9.8 in [RFC7950].

### 6.7. The "leafref" Type

A "leafref" value is represented using the same rules as the type of the leaf to which the leafref value refers.

### 6.8. The "identityref" Type

An "identityref" value is represented as a string -- the name of an identity. If the identity is defined in a module other than the leaf node containing the identityref value, the namespace-qualified form (Section 4) MUST be used. Otherwise, both the simple and namespace-qualified forms are permitted.

For example, consider the following schematic module:

```
module example-mod {
  ...
  import ietf-interfaces {
    prefix if;
  }
  ...
  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 represented as "[null]", i.e., an array with the "null" literal being its only element. For the purposes of this document, "[null]" is considered an atomic scalar value.

This encoding of the "empty" type was chosen instead of using simply "null" in order to facilitate the use of empty leafs in common programming languages where the "null" value of a member is treated as if the member is not present.

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.

When validating a value of the "union" type, the type information conveyed by the JSON encoding MUST also be taken into account. JSON syntax thus provides additional means for resolving the member type of the union that are not available in XML encoding.

For example, consider the following YANG definition:

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

In RESTCONF [RESTCONF], it is possible 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 that the value is supposed to be a number rather than a string, and it is not a valid "uint16" value.

Conversely, the value of

```
"bar": "1"
```

is to be interpreted as a 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 Section 9.13.2 in [RFC7950]. However, the encoding of namespaces in instance-identifier values follows the rules stated in Section 4, namely:

- The leftmost (top-level) data node name is always in the namespace-qualified form.
- Any subsequent data node name is in the namespace-qualified form if the node is defined in a module other than its parent node, and the simple form is used otherwise. This rule also holds for node names appearing in predicates.

For example,

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

is a valid instance-identifier 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".

## 7. I-JSON Compliance

I-JSON [RFC7493] 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 JSON encoding defined in this document deviates from I-JSON only in the representation of the "binary" type. In order to remain compatible with XML encoding, the base64 encoding scheme is used (Section 6.6), whilst I-JSON recommends base64url instead.

## 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 Section 17 of [RFC7950].

This document defines no mechanisms for signing and encrypting data modeled with YANG. Under normal circumstances, data security and integrity are guaranteed by the management protocol in use, such as NETCONF [RFC6241] or RESTCONF [RESTCONF]. If this is not the case, external mechanisms, such as Public-Key Cryptography Standards (PKCS) #7 [RFC2315] or JSON Object Signing and Encryption (JOSE) [RFC7515] [RFC7516], need to be considered.

JSON processing is rather different from XML, and JSON parsers may thus suffer from different types of vulnerabilities than their XML counterparts. To minimize these new security risks, software on the receiving side SHOULD reject all messages that do not comply with the rules of this document and reply with an appropriate error message to the sender.

## 9. References

### 9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <http://www.rfc-editor.org/info/rfc5234>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <http://www.rfc-editor.org/info/rfc6241>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <http://www.rfc-editor.org/info/rfc7493>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <http://www.rfc-editor.org/info/rfc7950>.

## 9.2. Informative References

[IEEE754-2008] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2008, DOI 10.1109/IEEESTD.2008.4610935, 2008, <http://standards.ieee.org/findstds/standard/754-2008.html>.

[RESTCONF] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", Work in Progress, draft-ietf-netconf-restconf-16, August 2016.

[RFC2315] Kaliski, B., "PKCS #7: Cryptographic Message Syntax Version 1.5", RFC 2315, DOI 10.17487/RFC2315, March 1998, <http://www.rfc-editor.org/info/rfc2315>.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <http://www.rfc-editor.org/info/rfc6020>.

[RFC7223] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 7223, DOI 10.17487/RFC7223, May 2014, <http://www.rfc-editor.org/info/rfc7223>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <http://www.rfc-editor.org/info/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <http://www.rfc-editor.org/info/rfc7516>.

[RFC7952] Lhotka, L., "Defining and Using Metadata with YANG", RFC 7952, DOI 10.17487/RFC7952, August 2016, <http://www.rfc-editor.org/info/rfc7952>.

[XML] 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>.

## 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 supported.

```
{
  "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"
```

```
          }
        }
      ]
    }
  }
```

## Acknowledgements

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

## Author's Address

Ladislav Lhotka
CZ.NIC

Email: lhotka@nic.cz