NETMOD Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 August 2026

K. Watsen, Ed.
Watsen Networks

22 February 2026

XML Encoding of Data Modeled with YANG
draft-yn-netmod-yang-xml-01

Abstract

This document defines encoding rules for representing YANG modeled configuration data, state data, parameters of Remote Procedure Call (RPC) operations or actions, and notifications defined using XML.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Network Modeling Working Group mailing list, which is archived at https://mailarchive.ietf.org/arch/browse/netmod. To subscribe: https://www.ietf.org/mailman/listinfo/netmod

This document is developed on GitHub at https://github.com/netmod-wg/yang-xml). If you wish to contribute, please consider opening a pull request (PR). See the README file for details.

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 https://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 26 August 2026.

Table of Contents

1. Introduction

This document defines encoding rules for representing YANG [I-D.yn-netmod-yang2] modeled configuration data, state data, parameters of Remote Procedure Call (RPC) operations or actions, and notifications defined using the Extensible Markup Language (XML) [XML].

2. Terminology and Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are defined in [I-D.yn-netmod-yang2]:

* 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 XML Encoding

This document defines XML encoding for YANG data trees and their subtrees. It is always assumed that there may be one or more top-level elements in XML-encoded configuration data and state data. RPC operations and notifications contain a single top-level element.

Instances of YANG data nodes (leafs, containers, leaf-lists, lists, anydata nodes, and anyxml nodes) are encoded as XML elements having the name of the YANG data node. Section 4) defines how the name is qualified with a namespace, 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].

With the exception of anyxml and schema-less anydata nodes, it is possible to map an XML-encoded data tree to other encodings, such as the JSON encoding as defined in [RFC7951], and vice versa. However, such conversions require the YANG data model to be available.

4. Names and Namespaces

An XML element name is always identical to the identifier of the corresponding YANG data node.

All XML elements encoding YANG data are namespace qualified. The XML default namespace is never used in YANG encoded data.

The namespace of an XML element is either inherited from its ancestor or set using the "xmlns" attribute in the element.

The "xmlns" attribute may either set the XML default namespace or define a prefix for the namespace. Note that the same XML may be encoded differently by different implementations. For instance, the following two XML documents are the same.

Document 1:

```
<foo xmlns="https://example.com/foo"/>
```

Document 2:

```
<my-prefix:foo xmlns:my-prefix="https://example.com/foo"/>
```

The "namespace" statement 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 of the main module is used.

A namespace MUST be set for all top-level XML elements and then also whenever the namespaces of the data node and its parent node are different.

For example, consider the following YANG module:

```
module example-foomod {

  namespace "https://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 XML-encoded configuration data:

```
<top xmlns="https://example.com/foomod">
  <foo>54</foo>
</top>
```

Note that the top-level element sets the default namespace which "foo" leaf inherits its parent container "top".

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

```
module example-barmod {

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

  prefix "barmod";

  import example-foomod {
    prefix "foomod";
  }

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

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

```
<top xmlns="https://example.com/foomod">
  <foo>54</foo>
  <bar xmlns="https://example.com/barmod">true</bar>
</top>
```

The "bar" leaf's element sets a new default namespace because its parent is defined in a different module.

Explicit namespace prefixes are sometimes needed when encoding values of the "identityref" and "instance-identifier" types. See Section 6.8 and Section 6.11 for details.

To improve readability of XML, a client or server that generates XML or XPath that uses prefixes SHOULD use the prefix defined by the module as the XML namespace prefix, unless there is a conflict.

5. Encoding of YANG Data Node Instances

5.1. The "leaf" Data Node

A leaf node is encoded as an XML element. The element's local name is the leaf's identifier, and its namespace is the module's XML namespace (see Section 4).

The value of the leaf node is encoded to XML according to the type (see Section 6 for type encoding rules) and is sent as character data in the element.

Example: For the leaf node definition

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

the following is a valid XML-encoded instance:

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

5.2. The "container" Data Node

A container node is encoded as an XML element. The element's local name is the container's identifier, and its namespace is the module's XML namespace (see Section 4).

The container's child nodes are encoded as subelements to the container element. If the container defines RPC or action input or output parameters, these subelements are encoded in the same order as they are defined within the "container" statement. Otherwise, the subelements are encoded in any order.

Any whitespace between the subelements to the container is insignificant, i.e., an implementation MAY insert whitespace characters between subelements.

If a non-presence container does not have any child nodes, the container may or may not be present in the XML encoding.

Example: For the container definition

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

the following is valid XML-encoded instance data:

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

5.3. The "leaf-list" Data Node

A leaf-list node is encoded as a series of XML elements. Each element's local name is the leaf-list's identifier, and its namespace is the module's XML namespace (see Section 4). There is no XML element surrounding the leaf-list as a whole.

The value of each leaf-list entry is encoded to XML according to the type and is sent as character data in the element (see Section 6 for type encoding rules).

The XML elements representing leaf-list entries MUST appear in the order specified by the user if the leaf-list is "ordered-by user"; otherwise, the order is implementation dependent. The XML elements representing leaf-list entries MAY be interleaved with elements for siblings of the leaf-list, unless the leaf-list defines RPC or action input or output parameters.

Example: For the leaf-list definition

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

the following is a valid XML-encoded instance:

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

5.4. The "list" Data Node

A list is encoded as a series of XML elements, one for each entry in the list. Each element's local name is the list's identifier, and its namespace is the module's XML namespace (see Section 4). There is no XML element surrounding the list as a whole.

The list's key nodes are encoded as subelements to the list's identifier element, in the same order as they are defined within the "key" statement.

The rest of the list's child nodes are encoded as subelements to the list element, after the keys. If the list defines RPC or action input or output parameters, the subelements are encoded in the same order as they are defined within the "list" statement. Otherwise, the subelements are encoded in any order.

Any whitespace between the subelements to the list entry is insignificant, i.e., an implementation MAY insert whitespace characters between subelements.

The XML elements representing list entries MUST appear in the order specified by the user if the list is "ordered-by user"; otherwise, the order is implementation dependent. The XML elements representing list entries MAY be interleaved with elements for siblings of the list, unless the list defines RPC or action input or output parameters.

Example: For the list definition

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

the following is a valid XML-encoded instance:

```
<bar>
    <foo>123</foo>
    <baz>zig</baz>
</bar>
<bar>
    <foo>456</foo>
    <baz>zag</baz>
</bar>
```

5.5. The "anydata" Data Node

An anydata node is encoded as an XML element. The element's local name is the anydata's identifier, and its namespace is the module's XML namespace (see Section 4). The value of the anydata node is a set of nodes, which are encoded as XML subelements to the anydata element.

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 XML-encoded instances to other encodings, such as JSON [RFC7951] may be impossible.

Note that any XML prefixes used in the encoding are local to each instance encoding. This means that the same XML may be encoded differently by different implementations.

Example: For the anydata definition

```
anydata data;
```

the following is a valid XML-encoded instance:

```
<data>
    <notification xmlns="urn:ietf:params:xml:ns:netmod:notification">
        <eventTime>2014-07-29T13:43:01Z</eventTime>
        <event xmlns="https://example.com/example-event">
            <event-class>fault</event-class>
            <reporting-entity>
                <card>Ethernet0</card>
            </reporting-entity>
            <severity>major</severity>
        </event>
    </notification>
</data>
```

5.6. The "anyxml" Data Node

An anyxml node is encoded the same as an anydata node. Please see Section 5.5 for how the anydata node is encoded.

5.7. Metadata Objects

Apart from instances of YANG data nodes, XML elements MAY contain XML attributes 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 XML Values

The type of the XML 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.

All of the examples in this section use a YANG "leaf-list" solely as means to illustrate multiple variations of the type.

6.1. Numeric Types

All numeric types (int8, int16, int32, uint8, uint16, uint32, int64, uint64, and decimal64) are represented as a text value conforming the to lexical representation for the type described in Section 9.2.1 of [I-D.yn-netmod-yang2] and Section 9.3.1 of [I-D.yn-netmod-yang2] .

Example: For the "int16" type

```
leaf-list foo {
  type int16;
}
```

the following is a valid XML-encoded instance:

```
<foo>4711</foo>    <!-- positive decimal value -->
<foo>-123</foo>    <!-- negative decimal value -->
<foo>0xf00f</foo>  <!-- positive hexadecimal value -->
<foo>-0xf</foo>    <!-- negative hexadecimal value -->
<foo>052</foo>     <!-- positive octal value -->
<foo>-052</foo>    <!-- negative octal value -->
```

6.2. The "string" Type

A "string" value is represented as character data conforming the to lexical representation for the type described in Section 9.4.1 of [I-D.yn-netmod-yang2].

Example: For the "string" type

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

the following is a valid XML-encoded instance:

```
<foo>This string is all on one line.</foo>
<foo>This string is:
        - on more than one line.
        - contains tab characters.
</foo>
```

6.3. The "boolean" Type

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

Example: For the "boolean" type

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

the following is a valid XML-encoded instance:

```
<foo>true</foo>
<foo>false</foo>
```

6.4. The "enumeration" Type

An "enumeration" value is represented as character data conforming the to lexical representation for the type described in Section 9.6.1 of [I-D.yn-netmod-yang2].

Example: For the "enumeration" type

```
leaf-list foo {
  type enumeration {
    enum one;
    enum two;
    enum three;
  }
}
```

the following is a valid XML-encoded instance:

```
<foo>one</foo>
<foo>two</foo>
<foo>three</foo>
```

6.5. The "bits" Type

A "bits" value is represented as character data conforming the to lexical representation for the type described in Section 9.7.2 of [I-D.yn-netmod-yang2].

Example: For the "bits" type

```
leaf-list foo {
  type bits {
     bit zero;
     bit one;
     bit two;
  }
}
```

the following is a valid XML-encoded instance:

```
<foo>zero</foo>
<foo>zero one</foo>
<foo>zero one two</foo>
```

6.6. The "binary" Type

A "binary" value is represented as character data conforming the to lexical representation for the type described in Section 9.8.2 of [I-D.yn-netmod-yang2].

Example: For the "binary" type

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

the following is a valid XML-encoded instance:

```
<foo>SGVsbG8gQm9iCg==</foo>  <!-- Hello Bob -->
<foo>SGVsbG8gQWxpY2UK</foo>  <!-- Hello Alice -->
```

6.7. The "leafref" Type

A "leafref" value is represented as character data conforming the to lexical representation for the type described in Section 9.9.4 of [I-D.yn-netmod-yang2].

Example: For the "leafref" type

```
leaf-list status {
  type leafref {
    path "/my-leaf";      // assume current value is "up"
  }
}

leaf-list ifname {
  type leafref {
    path "/my-list/key";  // assume current key values are
  }                       // "eth0", "eth1", and "eth2"
}

leaf-list color {
  type leafref {
    path "/my-leaf-list";  // assume current values are
  }                         // "red", "green", and "blue"
}
```

the following is valid XML-encoded instance data:

```
<status>up</status>

<ifname>eth0</ifname>
<ifname>eth1</ifname>
<ifname>eth2</ifname>

<color>red</color>
<color>green</color>
<color>blue</color>
```

6.8. The "identityref" Type

A "identityref" value is represented as character data containing the namespace qualified name of the referenced identity. As defined in [XML-NAMES], namespaces are either explicitly qualified using a prefix, or implicitly qualified using the default namespace for the XML element that containing the identityref value.

Example: For the "identityref" type

```
module example-crypto {
  yang-version 1.1;
  namespace "urn:example:crypto";
  prefix ec;

  identity symmetric-key-alg {
    description
      "Base identity used to identify symmetric-key crypto
       algorithms.";
  }

  identity blowfish {
    base symmetric-key-alg;
    description
      "Identity used to identify the 'blowfish' algorithm.";
  }
}

module example-my-crypto {
  yang-version 1.1;
  namespace "urn:example:my-crypto";
  prefix emc;

  import example-crypto {
    prefix ec;
  }

  identity aes {
    base ec:symmetric-key-alg;
    description
      "Identity used to identify the 'aes' algorithm.";
    }

  leaf-list foo {
    type identityref {
      base ec:symmetric-key-alg;
    }
  }
}
```

the following is a valid XML-encoded instance:

```
<foo xmlns:ec="urn:example:crypto">ec:blowfish</foo>
<foo xmlns:x="urn:example:crypto">x:blowfish</foo>
<foo xmlns:emc="urn:example:my-crypto">emc:aes</foo>
<foo>aes</foo>
```

In the above example:

* The first element uses the prefix from the imported module, per best practice.

* The second element uses a local prefix, as is allowed.

* The third element uses the prefix from the local module, per best practice but, as before, a local prefix is allowed.

* The fourth element uses the default namespace, assuming it is "urn:example:my-crypto".

6.9. The "empty" Type

An "empty" value is represented as an empty XML element.

Example: For the "empty" type

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

the following is a valid XML-encoded instance:

```
<foo/>
```

6.10. The "union" Type

A "union" value is represented as character data conforming the to lexical representation for the type described in Section 9.12.2 of [I-D.yn-netmod-yang2].

Example: For the "union" type

```
leaf-list foo {
  type union {
    type int32;
    type enumeration {
      enum "unbounded";
    }
  }
}
```

the following is a valid XML-encoded instance:

```
<foo>16</foo>
<foo>32</foo>
<foo>64</foo>
<foo>unbounded</foo>
```

6.11. The "instance-identifier" Type

A "instance-identifier" value is represented as character data. All node names in an instance-identifier value MUST be qualified with explicit namespace prefixes, and these prefixes MUST be declared in the XML namespace scope in the instance-identifier's XML element.

Any prefixes used in the encoding are local to each instance encoding. This means that the same instance-identifier may be encoded differently by different implementations.

Example: For the "instance-identifier" type

```
leaf-list foo {
  type instance-identifier;
}
```

the following is a valid XML-encoded instance:

```
<foo>/ex:system/ex:services/ex:ssh</foo>
<foo>/ex:system/ex:services/ex:ssh/ex:port</foo>
<foo>/ex:system/ex:user[ex:name='fred']</foo>
<foo>/ex:system/ex:user[ex:name='fred']/ex:type</foo>
<foo>/ex:system/ex:server[ex:ip='192.0.2.1'][ex:port='80']</foo>
<foo>/ex:system/ex:service[ex:name='foo'][ex:enabled='']</foo>
<foo>/ex:system/ex:services/ex:ssh/ex:cipher[.='blowfish-cbc']</foo>
<foo>/ex:stats/ex:port[3]</foo>
```

7. IANA Considerations

8. Security Considerations

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, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[I-D.yn-netmod-yang2] Watsen, K. and M. Bj旦rklund, "The YANG 2.0 Data Modeling Language", Work in Progress, Internet-Draft, draft-yn-netmod-yang2-00, 22 February 2026, <https://datatracker.ietf.org/doc/html/draft-yn-netmod-yang2-00>.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", W3C Recommendation REC-xml-20081126, 26 November 2008, <https://www.w3.org/TR/2008/REC-xml-20081126/>.

[XML-NAMES] 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, 8 December 2009, <http://www.w3.org/TR/2009/REC-xml-names-20091208>.

9.2. Informative References

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

[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, <https://www.rfc-editor.org/info/rfc6241>.

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

[RFC7951] Lhotka, L., "JSON Encoding of Data Modeled with YANG", RFC 7951, DOI 10.17487/RFC7951, August 2016, <https://www.rfc-editor.org/info/rfc7951>.

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

Acknowledgements

Substantial amounts of text in this document was copied from [RFC7950] and [RFC7951]. The authors wish to thank Martin Bj旦rklund and Ladislav Lhotka for authoring RFC 7950 and RFC 7951, respectively.

Author's Address

Kent Watsen (editor)
Watsen Networks
Email: kent+ietf@watsen.net