A Collection of YANG Design Patterns

draft-schoenw-netmod-yang-pattern-00

Abstract

YANG is a data modeling language used to model configuration and state data, remote procedure calls and notifications. This memo documents a number of YANG design patterns. These design patterns aim at providing general reusable solutions to commonly occurring problems in the design of YANG data models.

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 January 24, 2015.

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 .............................................. 2
2. Listening Endpoints ................................. 2
   2.1. Pattern ............................................. 2
   2.2. Example ............................................. 3
   2.3. Design Background ................................. 5
3. Outbound Connections ................................. 6
   3.1. Pattern ............................................. 7
   3.2. Example ............................................. 7
   3.3. Design Background ................................. 9
4. Presence Containers and Enabled Leafs .......................... 9
   4.1. Pattern ............................................. 9
   4.2. Example ............................................. 10
   4.3. Design Background ............................... 10
5. Config vs. Operational State ............................ 11
6. Security Considerations ................................ 11
7. Informative References .................................. 11
Authors’ Addresses ........................................ 11

1. Introduction

YANG is a data modeling language used to model configuration and state data, remote procedure calls and notifications [RFC6020]. A YANG design pattern is a general reusable solution to a commonly occurring problem in the design of YANG data models. The aim of this memo is to document some of the design patterns that have been found useful and to also capture the reasoning behind the design pattern.

This collection of design pattern is intended to complement the general usage guidelines documented in [RFC6087]. Some of the examples may refer to common data types defined in [RFC6991].

2. Listening Endpoints

A common problem is to model configuration of the endpoints a certain application or service is listening on.

2.1. Pattern

The best current practice design pattern looks like this:
list listen {
  key name;
  
  leaf name {
    type string;
    description "An arbitrary name for the listen endpoint.";
  }

  choice transport {
    mandatory true;
    case PROTOCOL {
      container PROTOCOL {
        // protocol specific endpoint information, for example:
        leaf address {
          type inet:ip-host;
          mandatory true;
        }
        leaf port {
          type inet:port-number;
          default DEFPORT;
        }
      }
    }
  }
}

The PROTOCOL cases may designate a transport layer protocol (e.g., tcp or sctp) or they may designate higher layer secure transports (e.g., tls). The DEFPORT indicates a default port number that is used if no explicit value is provided.

2.2. Example

Below is an example for the configuration of an HTTP server.
list listen {
    key name;
    leaf name {
        type string;
        description
            "An arbitrary name for the listen endpoint.";
    }
    choice transport {
        mandatory true;
        case tcp {
            container tcp {
                leaf address {
                    type inet:ip-host;
                    mandatory true;
                }
                leaf port {
                    type inet:port-number;
                    default 80;
                }
            }
        }
        case tls {
            container tls {
                leaf address {
                    type inet:ip-host;
                    mandatory true;
                }
                leaf port {
                    type inet:port-number;
                    default 443;
                }
            }
        }
    }
}

This leads to XML configurations of the following form:
<listen>
  <name>tcp-any-ipv6-default-port</name>
  <tcp>
    <address>::</address>
  </tcp>
</listen>

<listen>
  <name>tcp-any-ipv6-port-8080</name>
  <tcp>
    <address>::</address>
    <port>8080</port>
  </tcp>
</listen>

<listen>
  <name>tls-any-ipv4-port-443</name>
  <tls>
    <address>0.0.0.0</address>
    <port>443</port>
  </tls>
</listen>

2.3. Design Background

Using a name leaf as the list key makes the endpoints extensible since the name does not establish any constraints such as all IP addresses must be unique. Furthermore, using names enables future extensions dealing with multiple routing instances (where identical IP addresses may be assigned to different routing instances) or where link-local addresses are used on different explicitly identified interfaces.

Alternatives that have been considered but found too limiting:

contAINER listen {
  leaf address {
    type inet:ip-host;
    mandatory true;
  }
  leaf port {
    type inet:port-number;
    mandatory true;
  }
}

Using a simple container prevents the configuration of multiple listening endpoints for a given service.
list listen {
    key address;
    leaf address {
        type inet:ip-host;
    }
    leaf port {
        type inet:port-number;
    }
}

This alternative does not allow to configure the service on two different ports on the same IP address. It also does not allow to configure the same IP address with extensions such as support for multiple routing instances. Furthermore, this configuration is not extensible to support additional future transports.

container listen {
    list PROTOCOL {
        key name;
        leaf name {
            type string;
        }
        leaf ip {
            type inet:ip-host;
            mandatory true;
        }
        leaf port {
            type inet:port-number;
            default DEFPORT;
        }
    }
    // ...
}

This solution uses several protocol specific lists instead of using a choice in the common list. This has the disadvantage that it is not possible to generically refer to a listening endpoint, e.g., using a simple leafref. Furthermore, the lists mandate a partial order in which listening endpoints appear on configurations. By using a single list, entries for endpoints using different protocols can be ordered arbitrarily.

3. Outbound Connections

For certain services (like for example a DNS resolver), it is necessary to configure to which endpoint a service should connect to. In many cases, a service can be provided over multiple transport protocols and it is (a) possible that additional transports are added
over time and (b) that transports require additional information to identify an endpoint. Furthermore, it is often desirable to provision multiple endpoints in order to provide fallback options.

3.1. Pattern

The best current practice design pattern looks like this:

```yang
list server {
  key name;
  ordered-by user;

  leaf name {
    type string;
    description "An arbitrary name for the endpoint to connect to.";
  }

  choice transport {
    mandatory true;
    case PROTOCOL {
      container PROTOCOL {
        // protocol specific endpoint information, for example:
        leaf address {
          type inet:ip-host;
          mandatory true;
        }
        leaf port {
          type inet:port-number;
          default DEFPORT;
        }
      }
    }
  }
}

The PROTOCOL cases may designate a transport layer protocol (e.g., tcp or sctp) or they may designate higher layer secure transports (e.g., tls). The DEFPORT indicates a default port number that is used if no explicit value is provided.

3.2. Example
list server {
  key name;
  ordered-by user;

  leaf name {
    type string;
    description
      "An arbitrary name for the endpoint to connect to."
  }

  choice transport {
    mandatory true;
    case tcp {
      container tcp {
        leaf address {
          type inet:ip-host;
          mandatory true;
        }
        leaf port {
          type inet:port-number;
          default 80;
        }
      }
    }
    case tls {
      container tls {
        leaf address {
          type inet:ip-host;
          mandatory true;
        }
        leaf port {
          type inet:port-number;
          default 443;
        }
      }
    }
  }
}

This leads to XML configurations of the following form:
<server>
  <name>foo-over-tls</name>
  <tls>
    <address>foo.example.org</address>
  </tls>
</server>

<server>
  <name>foo-over-tcp-port-1224</name>
  <tcp>
    <address>foo.example.org</address>
    <port>1234</port>
  </tcp>
</server>

3.3. Design Background

Using a name leaf as the list key makes the endpoints extensible
since the name does not establish any constraints such as all IP
addresses must be unique. Furthermore, using names enables future
extensions dealing with multiple routing instances (where identical
IP addresses may be assigned to different routing instances).

The pattern allows to provision fallback endpoints (note that the
list is ordered-by user) and it allows new transports to be augmented
into the data model by adding additional choices.

4. Presence Containers and Enabled Leafs

It is sometimes desirable to have a container representing a certain
functionality that can be enabled or disabled. In some situations,
the functionality is by default enabled and the container should
be present while in other situations the functionality is by default
disabled and the container should not be present.

4.1. Pattern

For a container modeling an optional functionality that should be
enabled by default, the following pattern can be used:

```
container FUNCTION {
  leaf enabled {
    type boolean;
    default "true";
  }
}
```

This container will always exist and the boolean enabled leaf
defaults to true. Hence, the FUNCTION will by default be enabled.
For a container modeling an optional functionality that should be disabled by default, the following pattern can be used:

```yaml
container FUNCTION {
    presence "Enabled FUNCTION unless the 'enabled' leaf (which defaults to 'true') is set to 'false'";
    leaf enabled {
        type boolean;
        default "true";
    }
}
```

This presence container may be absent. If it does not exist, the functionality is disabled. If it exists, then the functionality will be enabled.

### 4.2. Example

The following example can be found in the system data model:

```yaml
container ntp {
    if-feature ntp;
    presence
        "Enables the NTP client unless the 'enabled' leaf (which defaults to 'true') is set to 'false'";
    description
        "Configuration of the NTP client."
    leaf enabled {
        type boolean;
        default true;
        description
            "Indicates that the system should attempt to synchronize the system clock with an NTP server from the 'ntp/server' list."
    }
}
```

### 4.3. Design Background

In both pattern, the ‘enabled’ leaf has been designed to be largely invisible in trim and explicit with-defaults mode [RFC6243]. In the first case, since the non-presence container is always visible and the functionality by default enabled, the ‘enabled’ leaf will not show up in a data tree. In the second case, since the presence container controls whether the ‘enabled’ leaf exists, it is by default not present even though the default is true. Once the
presence container has been created, the ‘enabled’ leaf will still be invisible due to its default value.

5. Config vs. Operational State

[Should we document what we currently have?]

6. Security Considerations

TBD

7. Informative References


Authors’ Addresses

Juergen Schoenwaelder
Jacobs University

Email: j.schoenwaelder@jacobs-university.de

Martin Bjorklund
Tail-f Systems

Email: mbj@tail-f.com