

Instant YANG

Hakan Millroth, Tail-f Systems (email: hakan@tail-f.com)

Introduction

This is a short primer on YANG - the NETCONF data modeling language. To learn more about YANG, take a look at the tutorials and examples at [YANG Central](#). The definitive resource is [RFC 6020](#), which is an excellent and quite readable reference manual. If you wonder why NETCONF needs a data modeling language, check out the "Why YANG?" document at YANG Central.

The Basics

A data model describes how data is represented and accessed. In YANG, data models are represented by definition hierarchies called *schema trees*. Instances of schema trees are called *data trees* and are encoded in XML.

YANG provides two constructs to specify leaf nodes: `leaf` and `leaf-list` statements. They differ in that a leaf node has at most one instance, while a `leaf-list` node may have multiple instances.

This YANG snippet specifies a flag called `enabled` that defaults to `true`:

```
leaf enabled {
    type boolean;
    default true;
}
```

Example XML instance:

```
<enabled>false</enabled>
```

The following YANG example specifies a list of cipher names:

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

Example XML instance:

```
<cipher>blowfish-cbc</cipher>
<cipher>3des-cbc</cipher>
```

The number of valid entries in a leaf-list can be constrained by optional `min-elements` and `max-elements` statements.

There are also two constructs in YANG for specifying non-leaf nodes: `container` and `list` statements. They differ mainly in that a `container` node has at most one instance, while a `list` node may have multiple instances (list entries are identified by keys that distinguish them from each other). The following YANG example uses a container statement to define a timeout mechanism for a request to a server. The timeout has two components: `access-timeout`, which represents the maximum time without server response, and `retries`, which represents the number of request attempts before giving up.

```
container timeout {
  leaf access-timeout {
    type uint32;
  }
  leaf retries {
    type uint8;
  }
}
```

Example XML instance:

```
<timeout>
  <access-timeout>60</access-timeout>
  <retries>2</retries>
</timeout>
```

The next example illustrates YANG lists. The entries in a list are identified by keys that distinguish them from each other. Here computer users are identified with their login name:

```
list user {
  key "login-name";
  leaf login-name {
    type string;
  }
  leaf full-name {
    type string;
  }
}
```

Example XML instances:

```
<user>
  <login-name>hakanm</login-name>
  <full-name>Hakan Millroth</fullname>
</user>
<user>
  <login-name>mbj</login-name>
  <full-name>Martin Bjorklund</fullname>
</user>
```

The examples so far have only used built-in YANG types. The set of built-in types are similar to those of many programming languages, but with some differences due to special requirements from the management domain. YANG also provides a mechanism through which additional types may be defined (these are called *derived types*). For example, the following defines a type `percent` that is a restriction of the built-in type `uint8`:

```
typedef percent {
    type uint8 {
        range "0 .. 100";
    }
}
```

The next example defines the type `ip-address` as the union of two other derived types (`union` is a built-in type):

```
typedef ip-address {
    type union {
        type ipv4-address;
        type ipv6-address;
    }
}
```

More Data Structuring Features

A `choice` node in the schema tree defines a set of alternatives, only one of which may exist at any one time. A choice consists of a number of branches, defined with `case` substatements. The nodes from one of the choice's branches (limited to one) exist in the data tree (the choice node itself does not exist in the data tree). Example:

```
choice route-distinguisher {
    case ip-address-based {
        leaf ip-address {
            type ipv4-address;
        }
        leaf ip-address-number {
            type uint16;
        }
    }
    case asn32-based {
        leaf asn32 {
            type uint32;
        }
        leaf two-byte-number {
            type uint16;
        }
    }
}
```

Example XML instantiation:

```
<asn32>12356789</asn32>
<two-byte-number>2468</two-byte-number>
```

A grouping defines a reusable collection of nodes. Here we define a grouping endpoint comprising two leaf nodes:

```
grouping endpoint {
  leaf address {
    type ip-address;
  }
  leaf port {
    type port-number;
  }
}
```

The `uses` statement is used to reference a grouping definition, copying the nodes defined by the grouping into the current schema tree. Continuing the example, we can reuse the `endpoint` grouping in defining the source and destination of a connection:

```
container connection {
  container source {
    uses endpoint {
      refine port {
        default 161;
      }
    }
  }
  container destination {
    uses endpoint {
      refine port {
        default 161;
      }
    }
  }
}
```

Here we have used the (optional) `refine` statement. Example XML instance:

```
<connection>
  <source>
    <address>192.168.0.3</address>
    <port>8080</port>
  </source>
  <destination>
    <address>192.168.0.4</address>
    <port>8080</port>
  </destination>
</connection>
```

The `presence` statement gives semantics to the existence of a container in the data tree (normally a container is just a data structuring mechanism). It takes as an argument a documentation string that describes what the model designer intends the node's presence to signify:

```
container logging {
  presence "Enables logging";
  leaf directory {
    type string;
    default "/var/log/myapplication";
  }
}
```

An alternative to using `presence` is to define a leaf (called, for example, `enabled` or `disabled` or `on` or `off`) conveying the presence of the container.

References

The `leafref` built-in type is used to reference a particular leaf instance in the data tree, as specified by a *path*. This path is specified using the XML Path Language (XPath), in a notation that is similar to the syntax for directory paths in Unix/Linux. Consider this example:

```
leaf interface-name {
  type leafref {
    path "/interface/name";
  }
}
```

Here `interface-name` is referencing another leaf using an absolute path. The next example instead lets `interface-name` reference a leaf using a relative path:

```
leaf interface-name {
  type leafref {
    path "../interface/name";
  }
}
```

There is an analogous referencing construct for list keys called `keyref`.

Data Constraints

YANG supports several types of constraints that express semantic properties of the data being modeled.

Data nodes can represent either configuration data or state data. The `config` statement specifies whether the definition it occurs in represents configuration data (`config=true`) or status data (`config=false`). If `config` is not specified, the default is the same as the parent schema node's `config` value. If the top node does not specify a `config` statement, the default is `config=true`.

YANG provides a mandatory statement to express that a node must exist in the data tree (it is also possible to have optional nodes). The following example shows the use of `config` and `mandatory`:

```
leaf host-name {
    type string;
    mandatory true;
    config true;
}
```

The `unique` statement ensures unique values within list siblings. Here is an example of its use (note that the constraint specifies that the combination of two leaf values, `ip` and `port`, must be unique):

```
list server {
    key "name";
    unique "ip port";
    leaf name {
        type string;
    }
    leaf ip {
        type ip-address;
    }
    leaf port {
        type port-number;
    }
}
```

The `must` statement expresses constraints that must be satisfied by each data node in the structure where it is defined. The `must` constraints are expressed using XPath expressions. The following example models an IP address that does not belong to the 192.* or 127.* networks:

```
leaf mirrorIp {
    type ip-address;
    default 0.0.0.0;
    must "false() = starts-with(current(), '192')" {
        error-message "The mirrorIp is in the forbidden "+
            "192.* network";
    }
    must "false() = starts-with(current(), '127')" {
        error-message "The mirrorIp is in the occupied "+
            "127.* network";
    }
}
```

Modules

YANG organizes data models into modules and submodules. A module can *import* data from other modules, and *include* data from submodules. Besides schema definitions, a module contains header statements (`yang-version`, `namespace`, `prefix`), linkage statements (`import` and `include`), meta information (`organization`, `contact`), and a revision history.

All nodes defined in a module belong to a specified XML namespace, which is specified by the URI of the namespace. It is good practice to have a one-to-one correspondence between modules and namespaces.

If an `import` or `include` statement includes a `revision-date` substatement, definitions are taken from the specified revision of the imported or included module. By importing specified module revisions one can allow published modules to evolve independently over time. YANG provides very specific rules for how published modules can evolve over time and still be backwards compatible (to oversimplify, it is possible to add new definitions, but not to delete obsolete definitions or change already published definitions).

Here is an example of a module definition:

```
module acme-module {
  namespace "http://acme.example.com/module";
  prefix acme;

  import "yang-types" {
    prefix yang;
  }
  include "acme-system";

  organization "ACME Inc.";
  contact joe@acme.example.com;
  description "The module for entities
              implementing the ACME products";

  revision 2007-06-09 {
    description "Initial revision.";
  }
  . . .
}
```

The module hierarchy can be augmented, allowing one module to add data nodes to the hierarchy defined in another module (without changing the augmented module). Augmentation can be conditional, using `when` statements, with new nodes appearing only if certain conditions are met. In the following example we (unconditionally) augment a generic router module to a data model for a RIP routing stack:

```
module rip_router {
  import router {
    prefix r;
  }
  augment "/r:system/r:router" {
    container rip {
      leaf version { . . . }
      list network-ip { . . . }
      list network-ifname { . . . }
      list neighbor { . . . }
      . . .
    }
  }
}
```

Note that the `import` statement assigns the prefix `r` to the imported module `router`. When we then reference identifiers from that module we must use the prefix, as in `/r:system/r:router`.

This example also illustrates a common YANG design pattern with a top-level container (`rip` in this case).

RPCs and Notifications

YANG allows the definition of NETCONF RPCs, extending the base set of operations provided by the NETCONF standard. Input and output parameters are modeled using YANG data definition statements.

In the following example we model a new RPC for activating software images:

```
rpc activate-software-image {
  input {
    leaf image-name {
      type string;
    }
  }
  output {
    leaf status {
      type string;
    }
  }
}
```

YANG data definition statements can also be used to model the names and content of NETCONF notifications. Here is an example where we reuse the definition of a connection from above:

```
notification link-failure {
  description "A link failure has been detected";
  container link {
    uses connection
  }
}
```

This example also illustrates the use of description statement. All YANG data definition statements (`leaf`, `list`, `leaf-list`, etc.) can include description statements.

Acknowledgements

This paper was inspired by Hetland's *Instant Python* text. Many of the examples and some verbatim text are from the YANG specification (RFC 6020). Some materials are from various IETF tutorials by Martin Björklund and Jürgen Schönwälder. Some examples are from Tail-f Systems' product documentation.