Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: March 13, 2014

A. Bierman
YumaWorks
M. Bjorklund
Tail-f Systems
K. Watsen
Juniper Networks
R. Fernando
Cisco

September 9, 2013

RESTCONF Protocol
draft-bierman-netconf-restconf-01

Abstract

This document describes a RESTful protocol that provides a programmatic interface over HTTP for accessing data defined in YANG, using the datastores defined in NETCONF.

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 March 13, 2014.

Table of Contents

1. Introduction

There is a need for standard mechanisms to allow WEB applications to access the configuration data, operational data, data-model specific protocol operations, and notification events within a networking device, in a modular and extensible manner.

This document describes a RESTful protocol called RESTCONF, running over HTTP [RFC2616], for accessing data defined in YANG [RFC6020], using datastores defined in NETCONF [RFC6241].

The NETCONF protocol defines configuration datastores and a set of Create, Retrieve, Update, Delete (CRUD) operations that can be used to access these datastores. The YANG language defines the syntax and semantics of datastore content, operational data, custom protocol operations, and notification events. RESTful operations are used to access the hierarchical data within a datastore.

A RESTful API can be created that provides CRUD operations on a NETCONF datastore containing YANG-defined data. This can be done in a simplified manner, compatible with HTTP and RESTful design principles. Since NETCONF protocol operations are not relevant, the user should not need any prior knowledge of NETCONF in order to use the RESTful API.

Configuration data and state data are exposed as resources that can be retrieved with the GET method. Resources representing configuration data can be modified with the DELETE, PATCH, POST, and PUT methods. Data-model specific protocol operations defined with the YANG "rpc" statement can be invoked with the POST method. Data-model specific notification events defined with the YANG "notification" statement can be accessed (delivery method TBD).

1.1. Simple Subset of NETCONF Functionality

The framework and meta-model used for a RESTful API does not need to mirror those used by the NETCONF protocol. It just needs to be compatible with NETCONF. A simplified framework and protocol is needed that utilizes the three NETCONF datastores (candidate, running, startup), but hides the complexity of multiple datastores from the client.

A simplified transaction model is needed that allows basic CRUD operations on a hierarchy of conceptual resources. This represents a limited subset of the transaction capabilities of the NETCONF protocol.

Applications that require more complex transaction capabilities might

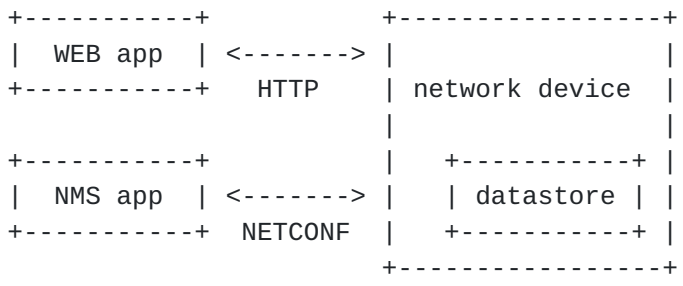
consider NETCONF instead of RESTCONF. The following transaction features are not directly provided in RESTCONF:

- datastore locking (full or partial)
- candidate datastore
- startup datastore
- validate operation
- confirmed-commit procedure

It is possible that a server could expose NETCONF operations as data-model specific operation resources, but that is out of scope within this document.

The RESTful API is not intended to replace NETCONF, but rather provide an additional simplified interface that follows RESTful principles and is compatible with a resource-oriented device abstraction. It is expected that applications that need the full feature set of NETCONF such as notifications will continue to use NETCONF.

The following figure shows the system components:



1.2. Data Model Driven API

RESTCONF combines the simplicity of a RESTful API over HTTP with the predictability and automation potential of a schema-driven API.

A RESTful client using HATEOAS principles would not use any data modelling language to define the application-specific content of the API. The client would discover each new child resource as it traverses the URIs return as Location IDs to discover the server capabilities.

This approach has 3 significant weaknesses wrt/ control of complex networking devices:

- inefficient performance: configuration APIs will be quite complex and may require thousands of protocol messages to discover all the schema information. Typically the data type information has to be passed in the protocol messages, which is also wasteful overhead.

- no data model richness: without a data model, the schema-level semantics and validation constraints are not available to the application.

- no tool automation: API automation tools need some sort of content schema to function. Such tools can automate various programming and documentation tasks related to specific data models.

Data model modules such as YANG modules serve as an "API contract" that will be honored by the server. An application designer can code to the data model, knowing in advance important details about the exact protocol operations and datastore content a conforming server implementation will support.

RESTCONF provides the YANG module capability information supported by the server, in case the client wants to use it. The URIs for custom protocol operations and datastore content are predictable, based on the YANG module definitions. Note that the YANG modules and predictable URIs are optional to use by the client. They can be completely ignored without any loss of protocol functionality.

Operational experience with CLI and SNMP indicates that operators learn the 'location' of specific service or device related data and do not expect such information to be arbitrary and discovered each time the client opens a management session to a server.

1.3. Terminology

The keywords "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].

1.3.1. NETCONF

The following terms are defined in [RFC6241]:

- candidate configuration datastore

- client

- configuration data

- datastore
- configuration datastore
- protocol operation
- running configuration datastore
- server
- startup configuration datastore
- state data
- user

1.3.2. HTTP

The following terms are defined in [RFC2616]:

- entity tag
- fragment
- header line
- message body
- method
- path
- query
- request URI
- response body

1.3.3. YANG

The following terms are defined in [RFC6020]:

- container
- data node
- key leaf

- leaf
- leaf-list
- list
- presence container (or P-container)
- RPC operation (now called protocol operation)
- non-presence container (or NP-container)
- ordered-by system
- ordered-by user

1.3.4. Terms

The following terms are used within this document:

- API resource: a resource with the media type "application/vnd.yang.api+xml" or "application/vnd.yang.api+json". API resources can only be edited by the server.
- data resource: a resource with the media type "application/vnd.yang.data+xml" or "application/vnd.yang.data+json". Data resources can be edited by clients or the server. Only YANG containers and lists can be data resources. Top-level YANG terminals are treated as fields within the datastore resource.
- datastore resource: a resource with the media type "application/vnd.yang.datastore+xml" or "application/vnd.yang.datastore+json". Datastore resources can only be edited by the server.
- edit operation: a RESTCONF operation on a data resource using the POST, PUT, PATCH, or DELETE method.
- event resource: a resource with the media type "application/vnd.yang.event+xml" or "application/vnd.yang.event+json". It represents a conceptual system or data-model specific event that is delivered within a notification message.
- field: a YANG terminal node within a resource.
- operation: the conceptual RESTCONF operation for a message, derived from the HTTP method, request URI, headers, and message body.

- operation resource: a resource with the media type "application/vnd.yang.operation+xml" or "application/vnd.yang.operation+json".

- patch: a generic PATCH operation on the target datastore. The media type of the message body content will identity the patch type in use.

- plain patch: a PATCH operation where the media type is "application/vnd.yang.data+xml" or "application/vnd.yang.data+json".

- query parameter: a parameter (and its value if any), encoded within the query portion of the request URI.

- resource: a conceptual object representing a manageable component within a device. Refers to the resource itself of the resource and all its fields.

- retrieval request: an operation using the GET or HEAD methods.

- target resource: the resource that is associated with a particular message, identified by the "path" component of the request URI.

- unified datastore: A conceptual representation of the device running configuration. The server will hide all NETCONF datastore details for edit operations, such as the ":candidate" and ":startup" capabilities.

- YANG Patch: a PATCH operation where the media type is "application/vnd.yang.patch+xml" or "application/vnd.yang.patch+json".

- YANG terminal node: a YANG node representing a leaf, leaf-list, or anyxml definition.

1.4. Overview

This document defines the RESTCONF protocol, a RESTful API for accessing conceptual datastores containing data defined with YANG language. RESTCONF provides an application framework and meta-model, using HTTP methods.

The RESTCONF resources are accessed via a set of URIs defined in this document. The set of YANG modules supported by the server will determine the additional data model specific operations and top-level data node resources available on the server.

1.4.1. Resource URI Map

The URI hierarchy for the RESTCONF resources consists of an entry point container, 3 top-level resources, and 1 field. Refer to Section 5 for details on each URI.

```
  /restconf
     /datastore
        /<top-level-data-nodes> (config=true or false)
     /modules
        /module
           /name
           /revision
           /namespace
           /feature
           /deviation
     /operations
        /<custom protocol operations>
     /version (field)
```

1.4.2. RESTCONF Message Examples

The examples within this document use the normative YANG module defined in Section 8 and the non-normative example YANG module defined in Section 14.

This section shows some typical RESTCONF message exchanges.

1.4.2.1. Retrieve the Top-level API Resource

By default, when a resource is retrieved, any nested resources are also returned, using the default encoding, which is XML.

The client may start by retrieving the top-level API resource, using the entry point URI "/restconf".

```
   GET /restconf?format=json HTTP/1.1
   Host: example.com
   Accept: application/vnd.yang.api+json
```

The server might respond as follows:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Content-Type: application/vnd.yang.api+json

{
  "restconf": {
    "datastore" : [ null ],
    "modules": {
      "module": [
        {
          "name" : "example-jukebox",
          "revision" : "2013-09-04",
          "namespace" : "example.com"
        }
      ]
    },
    "operations" : {
       "play" : [ null ]
    },
    "version": "1.0"
  }
}
```

To request that the response content to be encoded in XML, the "Accept" header can be used, as in this example request:

```
GET /restconf HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+xml
```

An alternate approach is provided using the "format" query parameter, as in this example request:

```
GET /restconf?format=xml HTTP/1.1
Host: example.com
```

The server will return the same response either way, which might be as follows :

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/vnd.yang.api+xml
```

```
<restconf xmlns="urn:ietf:params:xml:ns:yang:ietf-restconf">
  <datastore />
  <modules>
    <module>
      <name>example-jukebox</name>
      <revision>2013-09-04</revision>
      <namespace>example.com</namespace>
    </module>
  </modules>
  <operations>
    <play xmlns="http://example.com/ns/example-jukebox" />
  </operations>
  <version>1.0</version>
</restconf>
```

Refer to Section 3.3 for details on the GET method.

1.4.2.2. Create New Data Resources

To create a new "jukebox" resource, the client might send:

```
POST /restconf/datastore HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.data+json

{ "example-jukebox:jukebox" : [null] }
```

If the resource is created, the server might respond:

```
HTTP/1.1 201 Created
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Location: http://example.com/restconf/datastore/
  example-jukebox:jukebox
Last-Modified: Mon, 23 Apr 2012 17:01:00 GMT
ETag: b3a3e673be2
```

To create a new "artist" resource within the "library" resource, the client might send the following request.

```
POST /restconf/datastore/example-jukebox:jukebox/library HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.data+json

{ "artist" : {
    "name" : "Foo Fighters"
  }
}
```

If the resource is created, the server might respond as follows. Note that the "Location" header line is wrapped for display purposes only:

```
HTTP/1.1 201 Created
Date: Mon, 23 Apr 2012 17:02:00 GMT
Server: example-server
Location: http://example.com/restconf/datastore/
     example-jukebox:jukebox/library/artist/Foo%20Fighters
Last-Modified: Mon, 23 Apr 2012 17:02:00 GMT
ETag: b3830f23a4c
```

To create a new "album" resource for this artist within the "jukebox" resource, the client might send the following request. Note that the request URI header line is wrapped for display purposes only:

```
POST /restconf/datastore/example-jukebox:jukebox/
   library/artist/Foo%20Fighters  HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.data+json

{
  "album" : {
    "name" : "Wasting Light",
    "genre" : "example-jukebox:alternative",
    "year" : 2012   // note this is the wrong date
  }
}
```

If the resource is created, the server might respond as follows. Note that the "Location" header line is wrapped for display purposes only:

```
HTTP/1.1 201 Created
Date: Mon, 23 Apr 2012 17:03:00 GMT
Server: example-server
Location: http://example.com/restconf/datastore/
  example-jukebox:jukebox/library/artist/Foo%20Fighters/
  album/Wasting%20Light
Last-Modified: Mon, 23 Apr 2012 17:03:00 GMT
ETag: b8389233a4c
```

Refer to Section 3.4 for details on the POST method.

1.4.2.3. Replace an Existing Data Resource

Note: replacing a resource is a fairly drastic operation. The PATCH method is often more appropriate.

The album sub-resource is replaced here for example purposes only. To replace the "album" resource contents, the client might send as follows. Note that the request URI header line is wrapped for display purposes only:

```
PUT /restconf/datastore/example-jukebox:jukebox/
   library/artist/Foo%20Fighters/album/Wasting%20Light   HTTP/1.1
Host: example.com
If-Match: b3830f23a4c
Content-Type: application/vnd.yang.data+json

{
  "album" : {
    "name" : "Wasting Light",
    "genre" : "example-jukebox:alternative",
    "year" : 2011
  }
}
```

If the resource is updated, the server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:04:00 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 17:04:00 GMT
ETag: b27480aeda4c
```

Refer to Section 3.5 for details on the PUT method.

1.4.2.4. Patch an Existing Data Resource

To replace just the "year" field in the "album" resource (instead of replacing the entire resource), the client might send a plain patch as follows. Note that the request URI header line is wrapped for display purposes only:

```
PATCH /restconf/datastore/example-jukebox:jukebox/
   library/artist/Foo%20Fighters/album/Wasting%20Light HTTP/1.1
Host: example.com
If-Match: b8389233a4c
Content-Type: application/vnd.yang.data+json

{ "year" : 2011 }
```

If the field is updated, the server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:49:30 GMT
   Server: example-server
   Last-Modified: Mon, 23 Apr 2012 17:49:30 GMT
   ETag: b2788923da4c
```

The XML encoding for the same request might be:

```
   PATCH /restconf/datastore/example-jukebox:jukebox/
      library/artist/Foo%20Fighters/album/Wasting%20Light HTTP/1.1
   Host: example.com
   If-Match: b8389233a4c
   Content-Type: application/vnd.yang.data+xml


   <year xmlns="http://example.com/ns/example-jukebox">2011</year>
```

Refer to Section 3.6 for details on the PATCH method.

1.4.2.5. Delete an Existing Data Resource

To delete a resource such as the "album" resource, the client might send:

```
   DELETE /restconf/datastore/example-jukebox:jukebox/
      library/artist/Foo%20Fighters/album/Wasting%20Light HTTP/1.1
   Host: example.com
```

If the resource is deleted, the server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:49:40 GMT
   Server: example-server
```

Refer to Section 3.7 for details on the DELETE method.

1.4.2.6. Delete an Optional Field Within a Data Resource

The DELETE method cannot be used to delete an optional field within a resource. This can only be done using the PATCH method with the YANG Patch media type.

Refer to Section 7 for details on the YANG Patch method.

1.4.2.7. Invoke a Data Model Specific Operation

To invoke a data-model specific operation via an operation resource, the POST method is used. A client might send a "backup-datastore"

request as follows:

```
POST /restconf/operations/example-ops:backup-datastore   HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 204 No Content
Date: Mon, 23 Apr 2012 17:50:00 GMT
Server: example-server
```

Refer to Section 3.9 for details on using the POST method with operation resources.

2. Framework

The RESTCONF protocol defines a framework that can be used to implement a common API for configuration management. This section describes the components of the RESTCONF framework.

2.1. Message Model

The RESTCONF protocol uses HTTP entities for messages. A single HTTP message corresponds to a single protocol method. Most messages can perform a single task on a single resource, such as retrieving a resource or editing a resource. The exception is the PATCH method using the YANG Patch format. This allows multiple datastore edits within a single message.

2.2. Notification Model

[TBD]

2.3. Resource Model

The RESTCONF protocol operates on a hierarchy of resources, starting with the top-level API resource itself. Each resource represents a manageable component within the device.

A resource can be considered a collection of conceptual data and the set of allowed methods on that data. It can contain child nodes that are nested resources or fields. The child resource types and methods allowed on them are data-model specific.

A resource has its own media type identifier, represented by the "Content-Type" header in the HTTP response message. A resource can contain zero or more nested resources. A resource can be created and deleted independently of its parent resource, as long as the parent resource exists.

All RESTCONF resources are defined in this document except datastore contents, protocol operations, and notification events. The syntax and semantics for these resource types are defined with YANG statements.

2.3.1. RESTCONF Resource Types

The RESTCONF protocol defines some application specific media types to identify each of the available resource types. The following table summarizes the purpose of each resource.

Resource	Media Type
API	application/vnd.yang.api
Datastore	application/vnd.yang.datastore
Data	application/vnd.yang.data
Event	application/vnd.yang.event
Operation	application/vnd.yang.operation
Patch	application/vnd.yang.patch

RESTCONF Media Types

These resources are described in Section 5.

2.3.2. Resource Discovery

A client SHOULD start by retrieving the top-level API resource, using the entry point URI "/restconf".

The RESTCONF protocol does not include a resource discovery mechanism. Instead, the definitions within the YANG modules advertised by the server are used to construct a predictable operation or data resource identifier.

The "depth" query parameter can be used to control how many descendant levels should be included when retrieving sub-resources. This parameter can be used with the GET method to discover sub-resources within a particular resource.

Refer to Section 3.8.2 for more details on the "depth" parameter.

2.4. Datastore Model

A conceptual "unified datastore" is used to simplify resource management for the client. The RESTCONF datastore is a combination of the running configuration and any non-configuration data supported by the device. By default only configuration data is returned by a GET method on the datastore contents.

The underlying NETCONF datastores can be used to implement the unified datastore, but the server design is not limited to the exact datastore procedures defined in NETCONF.

The "candidate" and "startup" datastores are not visible in the RESTCONF protocol. Transaction management and configuration persistence are handled by the server and not controlled by the client.

2.4.1. Content Model

The RESTCONF protocol operates on a conceptual datastore defined with the YANG data modeling language. The server lists each YANG module it supports in the "/restconf/modules/module" resource in the top-level API resource type, using a structure based on the YANG module capability URI format defined in RFC 6020.

The conceptual datastore contents, data-model-specific operations and notification events are identified by this set of YANG module resources. All RESTCONF content identified as either a data resource, operation resource, or event resource is defined with the YANG language.

The classification of data as configuration or non-configuration is derived from the YANG "config" statement. Data retrieval with the GET method can be filtered in several ways, including the "config" parameter to retrieve configuration or non-configuration data.

Data ordering behavior is derived from the YANG "ordered-by" statement. The YANG Patch operation is provided to allow list or leaf-list fields to be inserted or moved in the same manner as NETCONF.

The server is not required to maintain system ordered data in any particular persistent order. The server SHOULD maintain the same data ordering for system ordered data until the next reboot or termination of the server. The server MUST maintain the same data ordering for user ordered data until the next reboot or termination of the server.

2.4.2. Editing Model

The RESTCONF datastore editing model is simple and direct, similar to the behavior of the ":writable-running" capability in NETCONF.

Each RESTCONF edit of a datastore resource is activated upon successful completion of the transaction. It is an implementation-specific matter how the server accomplishes a RESTCONF edit request. For example, a server which only accepts edits through a candidate datastore may internally edit this datastore and perform the "commit" operation automatically.

Applications which need more control over the editing model might consider using NETCONF instead of RESTCONF.

2.4.2.1. Edit Operation Discovery

Sometimes a server does not implement every operation for every resource. Sometimes data model requirements cause a node to implement a subset of the edit operations. For example, a server may not allow modification of a particular configuration data node after the parent resource has been created.

The OPTIONS method can be used to identify which HTTP methods are supported by the server for a particular resource. For example, if the server will allow a data resource node to be created then the POST method will be returned in the response.

2.4.2.2. Edit Collision Detection

Two "edit collision detection" mechanisms are provided in RESTCONF, for datastore and data resources.

- timestamp: the last change time is maintained and the "Last-Modified" and "Date" headers are returned in the response for a retrieval request. The "If-Unmodified-Since" header can be used in edit operation requests to cause the server to reject the request if the resource has been modified since the specified timestamp.

- entity tag: a unique opaque string is maintained and the "ETag" header is returned in the response for a retrieval request. The "If-Match" header can be used in edit operation requests to cause the server to reject the request if the resource entity tag does not match the specified value.

Note that the server is only required to maintain these properties for a datastore resource, not for individual data resources.

Example:

In this example, the server just supports the mandatory datastore last-changed timestamp. The client has previously retrieved the "Last-Modified" header and has some value cached to provide in the following request to replace a list entry with key value "11":

```
PATCH /restconf/datastore/example-jukebox:jukebox/
  library/artist/Foo%20Fighters/album/Wasting%20Light/year HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
If-Unmodified-Since: Mon, 23 Apr 2012 17:01:00 GMT
Content-Type: application/vnd.yang.data+json
```

```
  { "year" : "2011" }
```

In this example the datastore resource has changed since the time specified in the "If-Unmodified-Since" header. The server might respond:

```
   HTTP/1.1 304 Not Modified
   Date: Mon, 23 Apr 2012 19:01:00 GMT
   Server: example-server
   Last-Modified: Mon, 23 Apr 2012 17:45:00 GMT
   ETag: b34aed893a4c
```

2.4.3. Locking Model

Datastore locking is not provided by RESTCONF. An application that needs to make several changes to the running configuration datastore contents in sequence, without disturbance from other clients might consider using the NETCONF protocol instead of RESTCONF.

2.4.4. Persistence Model

Each RESTCONF edit of a datastore resource is saved to non-volatile storage in an implementation-specific matter by the server. There is no guarantee that configuration changes are saved immediately, or that the saved configuration is always a mirror of the running configuration.

Applications which need more control over the persistence model might consider using NETCONF instead of RESTCONF.

2.4.5. Defaults Model

NETCONF has a rather complex defaults handling model for leafs. RESTCONF attempts to avoid this complexity by restricting the operations that can be applied to a resource.

If the target of a GET method (plus "select" value) is a data node that represents a leaf that has a default value, and the leaf has not been given a value yet, the server MUST return the default value that is in use by the server.

The GET method returns only descendant nodes that exist, which will be determined by the server. There is no mechanism for the client to ask the server for the default values of nested resources that would be used for any nodes not present, but some default value is in use by the server. (There is no retrieval mode like "with-defaults=report-all" in NETCONF.)

Applications which need more control over the defaults model might consider using NETCONF instead of RESTCONF.

2.5. Transaction Model

The RESTCONF protocol provides an extensible transaction framework that allows a simplified transaction model that uses plain REST operations to edit one resource (and sub-resources) at a time. It also provides YANG Patch, which is a standard variant of the PATCH method. This allows a richer set of edit operations that can be applied to multiple resources at once.

RESTCONF does not provide a more complex transaction model that allows for multiple edits to be stored in a temporary scratchpad and committed all at once.

Applications which need more control over the transaction model might consider using NETCONF instead of RESTCONF.

2.6. Extensibility Model

The RESTCONF protocol is designed to be extensible for datastore content and data-model specific protocol operations. New protocol operations can be added without changing the entry point if they are optional and do not alter any existing operations.

Separate namespaces for each YANG module are used. Content encoded in XML will indicate the module using the "namespace" URI value in the YANG module. Content encoded in JSON will indicate the module using the module name specified in the YANG module, but this is not required unless multiple sibling nodes have the same YANG identifier name. JSON encoding rules for module names are specified in [I-D.lhotka-netmod-json].

2.7. Versioning Model

The version of a resource instance is identified with an entity tag, as defined by HTTP. The version identifiers in this section apply to the version of the schema definition of a resource. There are two types of schema versioning information used in the RESTCONF protocol:

- the RESTCONF protocol version
- data and operation resource definition versions

The protocol version is identified by the string used for the well-known URI entry point "/restconf". This would be changed (e.g., "/restconf2") if non-backward compatible changes are ever needed.

Minor version changes that do not break backward-compatibility will not cause the entry point to change.

The API "restconf/version" resource can be used by the client to identify the exact version of the RESTCONF protocol implemented by the server. This value will include the complete RESTCONF protocol version. The "/restconf/version" resource MUST be updated every time the protocol specification is republished.

The resource definition version for a data or operation resource is a date string, which is the revision date of the YANG module that defines the resource. The resource version for all other resource types is a numeric string, defined by the "/restconf/version" field.

2.8. Retrieval Filtering Model

There are three types of filtering for retrieval of data resources in the RESTCONF protocol.

- conditional all-or-nothing: use some conditional test mechanism in the request headers and retrieve either a complete "200 OK" response if the condition is met, or a "304 Not Modified" Status-Line if the condition is not met.

- data classification: request configuration or non-configuration data.

- filter: request a subset of all possible descendant nodes within the target resource. The "select" query parameter can be used for this purpose.

Refer to Section 5.3.2 for details on data retrieval filtering.

2.9. Access Control Model

The RESTCONF protocol provides no granular access control for any content except for operation and data resources. The NETCONF Access Control Model (NACM) is defined in [RFC6536]. There is a specific mapping between RESTCONF operations and NETCONF edit operations, defined in Table 1. The resource path also needs to be converted internally by the server to the corresponding YANG instance-identifier. Using this information, the server can apply the NACM access control rules to RESTCONF messages.

The server MUST NOT allow any operation to any resources that the client is not authorized to access.

3. Operations

The RESTCONF protocol uses HTTP methods to identify the CRUD operation requested for a particular resource. The following table shows how the RESTCONF operations relate to NETCONF protocol operations:

RESTCONF	NETCONF
OPTIONS	none
HEAD	none
GET	<get-config>, <get>
POST	<edit-config> (operation="create")
PUT	<edit-config> (operation="replace")
PATCH	<edit-config> (operation="merge")
DELETE	<edit-config> (operation="delete")

Table 1: CRUD Methods in RESTCONF

The NETCONF "remove" operation attribute is not supported by the HTTP DELETE method. The resource must exist or the DELETE method will fail.

This section defines the RESTCONF protocol usage for each HTTP method.

3.1. OPTIONS

The OPTIONS method is sent by the client to discover which methods are supported by the server for a specific resource. It is supported for all media types. Note that implementation of this method is part of HTTP, and this section does not introduce any additional requirements.

The request MUST contain a request URI that contains at least the entry point component.

The server will return a "Status-Line" header containing "204 No Content". and include the "Allow" header in the response. This header will be filled in, based on the target resource media type. Other headers MAY also be included in the response.

Example 1:

A client might request the methods supported for a data resource called "library":

```
   OPTIONS /restconf/datastore/example-jukebox:jukebox/
     library/artist   HTTP/1.1
   Host: example.com
```

The server might respond (for a config=true list):

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:01:00 GMT
   Server: example-server
   Allow: OPTIONS,HEAD,GET,POST,PUT,PATCH,DELETE
```

Example 2:

A client might request the methods supported for a non-configuration "counters" resource within a "system" resource:

```
   OPTIONS /restconf/datastore/example-system:system/
      counters HTTP/1.1
   Host: example.com
```

The server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:02:00 GMT
   Server: example-server
   Allow: OPTIONS,HEAD,GET
```

Example 3:

A client might request the methods supported for an operation resource called "play":

```
   OPTIONS /restconf/operations/example-jukebox:play HTTP/1.1
   Host: example.com
```

The server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 23 Apr 2012 17:02:00 GMT
   Server: example-server
   Allow: POST
```

3.2. HEAD

The HEAD method is sent by the client to retrieve just the headers that would be returned for the comparable GET method, without the response body. It is supported for all resource types, except operation resources.

The request MUST contain a request URI that contains at least the entry point component.

The same query parameters supported by the GET method are supported by the HEAD method. For example, the "select" query parameter can be used to specify a nested resource within the target resource.

The access control behavior is enforced as if the method was GET instead of HEAD. The server MUST respond the same as if the method was GET instead of HEAD, except that no response body is included.

Example:

The client might request the response headers for JSON representation of the "library" resource:

```
HEAD /restconf/datastore/example-jukebox:jukebox/
    library HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:02:40 GMT
Server: example-server
Content-Type: application/vnd.yang.data+json
Cache-Control: no-cache
Pragma: no-cache
ETag: a74eefc993a2b
Last-Modified: Mon, 23 Apr 2012 11:02:14 GMT
```

3.3. GET

The GET method is sent by the client to retrieve data and meta-data for a resource. It is supported for all resource types, except operation resources. The request MUST contain a request URI that contains at least the entry point component.

The following query parameters are supported by the GET method:

Name	Section	Description
config	3.8.1	Request either configuration or non-configuration data
depth	3.8.2	Control the depth of a retrieval request

format	3.8.3	Request either JSON or XML content in the response
select	3.8.6	Specify a nested resource within the target resource

GET Query Parameters

The server MUST NOT return any data resources for which the user does not have read privileges.

If the user is not authorized to read any portion of the target resource, an error response containing a "403 Forbidden" Status-Line is returned to the client.

If the user is authorized to read some but not all of the target resource, the unauthorized content is omitted from the response message body, and the authorized content is returned to the client.

Example:

The client might request the response headers for a JSON representation of the "library" resource:

```
GET /restconf/datastore/example-jukebox:jukebox/
  library/artist/Foo%20Fighters/album?format=json HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+json
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:02:40 GMT
Server: example-server
Content-Type: application/vnd.yang.data+json
Cache-Control: no-cache
Pragma: no-cache
ETag: a74eefc993a2b
Last-Modified: Mon, 23 Apr 2012 11:02:14 GMT

{
  "album" : {
    "name" : "Wasting Light",
    "genre" : "example-jukebox:alternative",
    "year" : 2011
  }
}
```

3.4. POST

The POST method is sent by the client for various reasons. The server uses the target resource media type to determine how to process the request.

The request MUST contain a request URI that contains a target resource which identifies one of the following resource types:

Type	Description
Datastore	Create a top-level configuration data resource
Data	Create a configuration data sub-resource
Operation	Invoke protocol operation

Resource Types that Support POST

3.4.1. Create Resource Mode

If the target resource type is a Datastore or Data resource, then the POST is treated as a request to create a resource or sub-resource.

The following query parameters are supported by the POST method for Datastore and Data resource types. They can only be used for YANG list data nodes which are ordered by the user.

Name	Section	Description
insert	3.8.4	Specify where to insert a resource
point	3.8.5	Specify the insert point for a resource

POST Query Parameters

If the POST method succeeds, a "204 No Content" Status-Line is returned and there is no response message body.

If the user is not authorized to create the target resource, an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.4.2. Invoke Operation Mode

If the target resource type is an Operation resource, then the POST method is treated as a request to invoke that operation. The message body (if any) is processed as the operation input parameters. Refer to Section 5.4 for details on Operation resources.

If the POST method succeeds, a "200 OK" Status-Line is returned if there is a response message body, and a "204 No Content" Status-Line is returned if there is no response message body.

If the user is not authorized to invoke the target operation, an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.5. PUT

The PUT method is sent by the client to replace the target resource.

The request MUST contain a request URI that contains a target resource that identifies the data resource to replace.

If the PUT method succeeds, a "200 OK" Status-Line is returned, and there is no response message body.

If the user is not authorized to replace the target resource an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.6. PATCH

The PATCH method uses the HTTP PATCH method defined in [RFC5789] to provide an extensible framework for resource patching mechanisms. Each patch type needs a unique media type. Any number of patch types can be supported by the server. There are two mandatory patch types that MUST be implemented by the server:

- plain patch type: If the specified media type is "application/vnd.yang.data", then the PATCH method is a simple merge operation on the target resource. The message body contains the XML or JSON encoded resource content that will be merged with the target resource.

- YANG Patch type: If the specified media type is "application/vnd.yang.patch", then the PATCH method is a YANG Patch formatted list of edits (see Section 7). The message body contains the XML

or JSON encoded instance of the 'patch' container specified in the 'ietf-restconf' YANG module (see Section 8).

The PATCH method MUST be used to create or delete an optional field within an existing resource or sub-resource.

If the PATCH method succeeds, a "200 OK" Status-Line is returned, and there is no response message body.

If the user is not authorized to alter the target resource an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.7. DELETE

The DELETE method is used to delete the target resource.

If the DELETE method succeeds, a "200 OK" Status-Line is returned, and there is no response message body.

If the user is not authorized to delete the target resource then an error response containing a "403 Forbidden" Status-Line is returned to the client. All other error responses are handled according to the procedures defined in Section 6.

3.8. Query Parameters

Each RESTCONF operation allows zero or more query parameters to be present in the request URI. Refer to Section 3 for details on the query parameters used in the definition of each operation.

Query parameters can be given in any order. Each parameter can appear zero or one time. A default value may apply if the parameter is missing.

This section defines all the RESTCONF query parameters.

3.8.1. "config" Parameter

The "config" parameter is used to specify whether configuration or non-configuration data is requested.

This parameter is only supported for the GET and HEAD methods. It is also only supported if the target resource is a data resource.

```
syntax: config= true | false
default: true
```

Example:

This example request by the client would retrieve only the non-configuration data nodes that exist within the "library" resource.

```
GET /restconf/datastore/example-jukebox:jukebox/
  library?config=false   HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
```

The server might respond:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:30 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/vnd.yang.data+json

{
  "library" : {
     "artist-count" : 42,
     "album-count" : 59,
     "song-count" : 374
  }
}
```

3.8.2. "depth" Parameter

The "depth" parameter is used to specify the number of nest levels returned in a response for a GET method. A nest-level consists of the target resource and any child nodes which are contained within the target resource node.

The start level is determined by the target resource for the operation.

```
syntax: depth=<range: 1..max> | unbounded
default: unbounded
```

Example:

This example operation would retrieve 2 levels of configuration data nodes that exist within the top-level "jukebox" resource.

```
GET /restconf/datastore/example-jukebox:jukebox
    ?depth=2 HTTP/1.1
Host: example.com
```

```
   Accept: application/vnd.yang.data+json
```

The server might respond:

```
   HTTP/1.1 200 OK
   Date: Mon, 23 Apr 2012 17:11:30 GMT
   Server: example-server
   Cache-Control: no-cache
   Pragma: no-cache
   Content-Type: application/vnd.yang.data+json

   {
     "jukebox" : {
       "library" : {
         "artist" : {
           "name" : "Foo Fighters"
         }
       },
       "player" : {
         "gap" : 0.5
       }
     }
   }
```

By default, the server will include all sub-trees within a retrieved resource, which is the same resource type. Only only level of sub-resources with a different media type than the target resource will be returned.

For example, if the client retrieves the "application/vnd.yang.api" resource type, then the node for the datastore resource is returned as an empty node, because all its child nodes are data resources. The entire contents of the datastore are not returned in this case. The operation resources also are returned as empty nodes (e.g. "play" operation).

Request URL:

```
   GET /restconf HTTP/1.1
```

Response:

```
{
  "restconf": {
    "datastore" : [ null ],
    "modules": {
      "module": [
        {
          "name" : "example-jukebox",
          "revision" : "2013-09-04",
          "namespace" : "example.com"
        }
      ]
    },
    "operations" : {
       "play" : [ null ]
    },
    "version": "1.0"
  }
}
```

3.8.3. "format" Parameter

The "format" parameter is used to specify the format of any content returned in the response. Note that this parameter MAY be used instead of the "Accept" header to identify the format desired in the response.

The "format" parameter is only supported for the GET and HEAD methods. It is supported for all RESTCONF media types.

```
syntax: format= xml | json
default: Accept header, then xml
```

If the "format" parameter is present, then it overrides the Accept header, if present. If neither the Accept header or the "format" parameter are present, then the default is XML.

Examples:

```
GET /restconf/datastore/example-routing:routing  HTTP/1.1
Host: example.com
Accept: application/vnd.yang.data+json
```

This example request would retrieve only the configuration data nodes that exist within the top-level "routing" resource, and retrieve them in JSON encoding.

```
GET /restconf/datastore/example-routing:routing
    ?format=json HTTP/1.1
```

```
   Host: example.com
```

This example request would retrieve only the configuration data nodes that exist within the top-level "routing" resource, and retrieve them in JSON encoding.

3.8.4. "insert" Parameter

The "insert" parameter is used to specify how a resource should be inserted within a user-ordered list.

This parameter is only supported for the POST method. It is also only supported if the target resource is a data resource, and that data represents a YANG list that is ordered by the user, not the system.

If the values "before" or "after" are used, then a "point" parameter for the insertion parameter MUST also be present.

```
     syntax: insert= first | last | before | after
     default: last
```

Example:

Request from client:

```
  PATCH /restconf/datastore/example-jukebox:jukebox/
    playlist/Foo-One?insert=first HTTP/1.1
  Host: example.com
  Content-Type: application/vnd.yang.data+json

  {
    "song" : {
       "index" : 1,
       "id" : "/example-jukebox:jukebox/library/artist/
           Foo%20Fighters/album/Wasting%20Light/song/Rope"
     }
  }
```

Response from server:

```
  HTTP/1.1 201 Created
  Date: Mon, 23 Apr 2012 13:01:20 GMT
  Server: example-server
  Last-Modified: Mon, 23 Apr 2012 13:01:20 GMT
  Location: http://example.com/restconf/datastore/
     example-jukebox:jukebox/playlist/Foo-One/song/1
  ETag: eeeada438af
```

3.8.5. "point" Parameter

The "point" parameter is used to specify the insertion point for a data resource that is being created or moved within a user ordered list. It is ignored unless the "insert" query parameter is also present, and has the value "before" or "after".

This parameter contains the instance identifier of the resource to be used as the insertion point for a POST method. It is encoded according to the rules defined in Section 5.3.1. There is no default for this parameter.

```
   syntax: point= <instance-identifier of insertion point node>
```

Example:

In this example, the client is inserting a new "song" resource within an "album" resource after another song. The request URI is split for display purposes only.

Request from client:

```
  POST /restconf/datastore/example-jukebox:jukebox/
     library/artist/Foo%20Fighters/album/Wasting%20Light?insert=after
     &point=/restconf/datastore/example-jukebox:jukebox/
     library/artist/Foo%20Fighters/album/Wasting%20Light/song/
     Bridge%20Burning   HTTP/1.1
  Host: example.com
  Content-Type: application/vnd.yang.data+json

  {
    "song" : {
       "name" : "Rope",
       "location" : "/media/rope.mp3",
       "format" : "MP3",
       "length" : 259
    }
  }
```

Response from server:

```
  HTTP/1.1 204 No Content
  Date: Mon, 23 Apr 2012 13:01:20 GMT
  Server: example-server
  Last-Modified: Mon, 23 Apr 2012 13:01:20 GMT
  ETag: abcada438af
```

3.8.6. "select" Parameter

The "select" query parameter is used to specify an expression which can represent a subset of all data nodes within the target resource. It contains a relative path expression, using the target resource as the context node.

It is supported for all resource types except operation resources. The contents are encoded according to the "api-select" rule defined in Section 5.3.1. This parameter is only allowed for GET and HEAD methods.

[FIXME: the syntax of the select string is still TBD; XPath, schema-identifier, regular expressions, something else; Perhaps add parameter "xselect" for XPath and this param is limited to a path-expr.]

In this example the client is retrieving the API version field from the server in JSON format:

```
GET /restconf?select=version&format=json HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+json
```

The server might respond as follows.

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Last-Modified: Sun, 22 Apr 2012 01:00:14 GMT
Content-Type: application/vnd.yang.api+json

{ "version": "1.0" }
```

3.9. Protocol Operations

The RESTCONF protocol allows data-model specific protocol operations to be invoked using the POST method. The media type "application/vnd.yang.operation+xml" or "application/vnd.yang.operation+json" MUST be used in the "Content-Type" line in the message header.

Data model specific operations are supported. The syntax and semantics of these operations exactly correspond to the YANG "rpc" statement definition for the operation.

Refer to Section 5.4 for details on operation resources.

4. Messages

This section describes the messages that are used in the RESTCONF protocol.

4.1. Request URI Structure

Resources are represented with URIs following the structure for generic URIs in [RFC3986].

A RESTCONF operation is derived from the HTTP method and the request URI, using the following conceptual fields:

```
     <OP> /restconf/<path>?<query>#<fragment>


       ^       ^        ^       ^         ^
       |       |        |       |         |
    method  entry  resource  query    fragment

       M       M        O        O         I

    M=mandatory, O=optional, I=ignored


    <text> replaced by client with real values
```

- method: the HTTP method identifying the RESTCONF operation requested by the client, to act upon the target resource specified in the request URI. RESTCONF operation details are described in Section 3.

- entry: the well-known RESTCONF entry point ("/restconf").

- resource: the path expression identifying the resource that is being accessed by the operation. If this field is not present, then the target resource is the API itself, represented by the media type "application/vnd.yang.api".

- query: the set of parameters associated with the RESTCONF message. These have the familiar form of "name=value" pairs. There is a specific set of parameters defined, although the server MAY choose to support additional parameters not defined in this document.

- fragment: This field is not used by the RESTCONF protocol.

The client SHOULD NOT assume the final structure of a URI path for a resource. Instead, existing resources can be discovered with the GET

method. When new resources are created by the client, a "Location" header is returned, which identifies the path of the newly created resource. The client MUST use this exact path identifier to access the resource once it has been created.

The "target" of an operation is a resource. The "path" field in the request URI represents the target resource for the operation.

4.2. Message Headers

There are several HTTP header lines utilized in RESTCONF messages. Messages are not limited to the HTTP headers listed in this section.

HTTP defines which header lines are required for particular circumstances. Refer to each operation definition section in Section 3 for examples on how particular headers are used.

There are some request headers that are used within RESTCONF, usually applied to data resources. The following tables summarize the headers most relevant in RESTCONF message requests:

Name	Description
Accept	Response Content-Types that are acceptable
Content-Type	The media type of the request body
Host	The host address of the server
If-Match	Only perform the action if the entity matches ETag
If-Modified-Since	Only perform the action if modified since time
If-Unmodified-Since	Only perform the action if un-modified since time

RESTCONF Request Headers

The following tables summarize the headers most relevant in RESTCONF message responses:

Name	Description
Allow	Valid actions when 405 error returned
Content-Type	The media type of the response body
Date	The date and time the message was sent
ETag	An identifier for a specific version of a resource
Last-Modified	The last modified date and time of a resource
Location	The resource identifier for a newly created resource

RESTCONF Response Headers

4.3. Message Encoding

RESTCONF messages are encoded in HTTP according to RFC 2616. The "utf-8" character set is used for all messages. RESTCONF message content is sent in the HTTP message body.

Content is encoded in either JSON or XML format.

XML encoding rules for data nodes are defined in [RFC6020]. The same encoding rules are used for all XML content.

JSON encoding rules are defined in [I-D.lhotka-netmod-json]. Plain JSON cannot be used because special encoding rules are needed to handle multiple module namespaces and provide consistent data type processing.

Request input content encoding format is identified with the Content-Type header. This field MUST be present if a message body is sent by the client.

Response output content encoding format is identified with the Accept header, the "format" query parameter, or if neither is specified, the request input encoding format is used. If there was no request input, then the default output encoding is XML. File extensions encoded in the request are not used to identify format encoding.

4.4. RESTCONF Meta-Data

The RESTCONF protocol needs to retrieve the same meta-data that is used in the NETCONF protocol. Information about default leafs, last-modified timestamps, etc. are commonly used to annotate representations of the datastore contents. This meta-data is not defined in the YANG schema because it applies to the datastore, and

is common across all data nodes.

This information is encoded as attributes in XML, but JSON does not have a standard way of attaching non-schema defined meta-data to a resource or field.

4.4.1. JSON Encoding of RESTCONF Meta-Data

The YANG to JSON mapping [I-D.lhotka-netmod-json] does not support attributes because YANG does not support meta-data in data node definitions. This section specifies how RESTCONF meta-data is encoded in JSON.

Only simple meta-data is supported:

- A meta-data instance can appear 0 or 1 times for a particular data node

- A meta-data instance associated with a resource is encoded as if it were a YANG leaf of type "string", according to the encoding rules in [I-D.lhotka-netmod-json], except the identifier is prepended with a "@" (%40) character.

- A meta-data instance associated with a field within a resource is encoded as if it were a container for the meta-data values and the field value in its native encoding. It is encoded according to the rules in [I-D.lhotka-netmod-json], except the meta-data identifiers are prepended with a "@" (%40) character. The field name/value pair is repeated inside this container, which contains the actual value of the field.

Examples:

Meta-data:

```
enabled=<boolean>
owner=<owner-name>
```

YANG example:

```
container top {
  leaf A {
    type int32;
  }
  leaf B {
    type boolean;
  }
}
```

The client is retrieving the "top" data resource, and the server is including datastore meta-data. Note that a query parameter to request or suppress specific meta-data is not provided in RESTCONF.

```
GET /restconf/datastore/example:top HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+json
```

The server might respond as follows:

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Content-Type: application/vnd.yang.api+json

{
  "top": {
    "@enabled" : "true",
    "@owner" : "fred",
    "A" : {
      "@enabled" : "true",
      "A" : 42
    },
    "B" : {
      "@enabled" : "false",
      "B" : true
    }
  }
}
```

4.5. Return Status

Each message represents some sort of resource access. An HTTP "Status-Line" header line is returned for each request. If a 4xx or 5xx range status code is returned in the Status-Line, then the error information will be returned in the response, according to the format defined in Section 6.1.

4.6. Message Caching

Since the datastore contents change at unpredictable times, responses from a RESTCONF server generally SHOULD NOT be cached.

The server SHOULD include a "Cache-Control" header in every response that specifies whether the response should be cached. A "Pragma" header specifying "no-cache" MAY also be sent in case the "Cache-Control" header is not supported.

Instead of using HTTP caching, the client SHOULD track the "ETag" and/or "Last-Modified" headers returned by the server for the datastore resource (or data resource if the server supports it). A retrieval request for a resource can include headers such as "If-None-Match" or "If-Modified-Since" which will cause the server to return a "304 Not Modified" Status-Line if the resource has not changed. The client MAY use the HEAD method to retrieve just the message headers, which SHOULD include the "ETag" and "Last-Modified" headers, if this meta-data is maintained for the target resource.

5. Resources

The resources used in the RESTCONF protocol are identified by the "path" component in the request URI. Each operation is performed on a target resource.

5.1. API Resource (/restconf)

The API resource contains the state and access points for the RESTCONF features. It is the top-level resource and has the media type "application/vnd.yang.api+xml" or "application/vnd.yang.api+json". It is accessible through the well-known relative URI "/restconf".

The "restconf" container definition in the "ietf-restconf" module defined in Section 8 is used to specify the structure and syntax of the conceptual fields and sub-resources within the API resource.

The "restconf" entry point container, and all fields and sub-resources with the same resource type, are defined in the namespace of the "ietf-restconf" module.

There is one mandatory field "version" that identifies the specific version of the RESTCONF protocol implemented by the server:

- The same server-wide response MUST be returned each time this field is retrieved.
- It is assigned by the server when the server is started.
- The server MUST return the value "1.0" for this version of the RESTCONF protocol.
- This field is encoded with the rules for an "enumeration" data type, using the "version" leaf definition in Section 8.

This resource has the following child resources:

Child Resource	Description
datastore	Link to "datastore" resource
modules	YANG module capability URIs
operations	Data-model specific operations

RESTCONF Resources

5.1.1. /restconf/datastore

This mandatory resource represents the running configuration datastore and any non-configuration data available. It may be retrieved and edited directly. It cannot be created or deleted by the client. This resource type is defined in Section 5.2.

5.1.2. /restconf/modules

This mandatory resource contains the identifiers for the YANG data model modules supported by the server.

The server MUST maintain a last-modified timestamp for this resource, and return the "Last-Modified" header when this resource is retrieved with the GET or HEAD methods.

The server SHOULD maintain an entity-tag for this resource, and return the "ETag" header when this resource is retrieved with the GET or HEAD methods.

5.1.2.1. /restconf/modules/module

This mandatory resource contains one URI string for each YANG data model module supported by the server. There MUST be an instance of this resource for every YANG module that is accessible via an operation resource or a data resource.

The contents of the "module" resource are defined in the "module" YANG list statement in Section 8.

The server MAY maintain a last-modified timestamp for each instance of this resource, and return the "Last-Modified" header when this resource is retrieved with the GET or HEAD methods. If not supported then the timestamp for the parent "modules" resource MAY be used instead.

The server MAY maintain an entity-tag for each instance of this resource, and return the "ETag" header when this resource is retrieved with the GET or HEAD methods. If not supported then the timestamp for the parent "modules" resource MAY be used instead.

There are additional encoding requirements for this resource. The URI MUST follow the YANG module capability URI formatting defined in section 5.6.4 of [RFC6020].

5.1.2.2. Retrieval Example

In this example the client is retrieving the modules resource from the server in JSON format:

```
GET /restconf/modules&format=json HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+json
```

The server might respond as follows.

```
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Last-Modified: Sun, 22 Apr 2012 01:00:14 GMT
Content-Type: application/vnd.yang.api+json

{
  "modules": {
    "module": [
      {
        "name" : "foo",
        "revision" : "2012-01-02",
        "namespace" : "http://example.com/ns/foo",
        "feature" : [ "feature1", "feature2" ]
      },
      {
        "name" : "foo-types",
        "revision" : "2012-01-05",
        "namespace" : "http://example.com/ns/foo-types"
      },
      {
        "name" : "bar",
        "revision" : "2012-11-05",
        "namespace" : "http://example.com/ns/bar",
        "feature" : [ "bar-ext" ]
      }
    ]
  }
}
```

5.1.3. /restconf/operations

This optional resource provides access to the data-model specific protocol operations supported by the server. The server MAY omit this resource if no data-model specific operations are advertised.

Any data-model specific operations defined in the YANG modules advertised by the server MAY be available as child nodes of this resource.

5.2. Datastore Resource

A datastore resource represents the conceptual root of a tree of data resources.

The server MUST maintain a last-modified timestamp for this resource, and return the "Last-Modified" header when this resource is retrieved with the GET or HEAD methods. Only changes to configuration data resources within the datastore affect this timestamp.

The server SHOULD maintain a resource entity tag for this resource, and return the "ETag" header when this resource is retrieved with the GET or HEAD methods. The resource entity tag SHOULD be changed to a new previously unused value if changes to any configuration data resources within the datastore are made.

A datastore resource can be retrieved with the GET method, to retrieve either configuration data resources or non-configuration data resources within the datastore. The "config" query parameter is used to choose between them. Refer to Section 3.8.1 for more details.

The depth of the subtrees returned in retrieval operations can be controlled with the "depth" query parameter. The number of nest levels, starting at the target resource, can be specified, or an unlimited number can be returned. Refer to Section 3.8.2 for more details.

[FIXME: not clear if top-level YANG data nodes MUST be containers or lists.

A datastore resource can only be written directly with the PATCH method. Only the configuration data resources within the datastore resource can be edited directly with all methods.]

5.3. Data Resource

A data resource represents a YANG data node that is a descendant node of a datastore resource. Only YANG container and list data node types are considered to represent data resources. Other YANG data nodes are considered to be fields within their parent resource.

For configuration data resources, the server MAY maintain a last-modified timestamp for the resource, and return the "Last-Modified"

header when it is retrieved with the GET or HEAD methods.

For configuration data resources, the server MAY maintain a resource entity tag for the resource, and return the "ETag" header when it is retrieved as the target resource with the GET or HEAD methods. The resource entity tag SHOULD be changed to a new previously unused value if changes to the resource or any configuration resource within the resource is altered.

A data resource can be retrieved with the GET method, to retrieve either configuration data resources or non-configuration data resources within the target resource. The "config" query parameter is used to choose between them. Refer to Section 3.8.1 for more details.

The depth of the subtrees returned in retrieval operations can be controlled with the "depth" query parameter. The number of nest levels, starting at the target resource, can be specified, or an unlimited number can be returned. Refer to Section 3.8.2 for more details.

A configuration data resource can be altered by the client with some of all of the edit operations, depending on the target resource and the specific operation. Refer to Section 3 for more details on edit operations.

5.3.1. Encoding YANG Instance Identifiers in the Request URI

In YANG, data nodes are named with an absolute XPath expression, from the document root to the target resource. In RESTCONF, URL friendly path expressions are used instead.

The YANG "instance-identifier" (i-i) data type is represented in RESTCONF with the path expression format defined in this section.

Name	Comments
point	Insertion point is always a full i-i
path	Request URI path is a full or partial i-i

RESTCONF instance-identifier Type Conversion

The "path" component of the request URI contains the absolute path expression that identifies the target resource. The "select" query parameter is used to optionally identify the requested data nodes within the target resource to be retrieved in a GET method.

A predictable location for a data resource is important, since applications will code to the YANG data model module, which uses static naming and defines an absolute path location for all data nodes.

A RESTCONF data resource identifier is not an XPath expression. It is encoded from left to right, starting with the top-level data node, according to the "api-path" rule in Section 5.3.1.1. The node name of each ancestor of the target resource node is encoded in order, ending with the node name for the target resource.

If the "select" is present, it is encoded, starting with a child node of the target resource, according to the "api-select" rule defined in Section 5.3.1.1.

If a data node in the path expression is a YANG list node, then the key values for the list (if any) are encoded according to the "key-value" rule. If the list node is the target resource, then the key values MAY be omitted, according to the operation. For example, the POST method to create a new data resource for a list node does not allow the key values to be present in the request URI.

The key leaf values for a data resource representing a YANG list MUST be encoded as follows:

- The value of each leaf identified in the "key" statement is encoded in order.

- All the components in the "key" statement MUST be encoded. Partial instance identifiers are not supported.

- Each value is encoded using the "key-value" rule in Section 5.3.1.1, according to the encoding rules for the data type of the key leaf.

- An empty string can be a valid key value (e.g., "/top/list/key1//key3").

- The "/" character MUST be URL-encoded (i.e., "%2F").

- All whitespace MUST be URL-encoded.

- A "null" value is not allowed since the "empty" data type is not allowed for key leafs.

- The XML encoding is defined in [RFC6020].

- The JSON encoding is defined in [I-D.lhotka-netmod-json].

- The entire "key-value" MUST be properly URL-encoded, according to the rules defined in [RFC3986].

- resource URI values returned in Location headers for data resources SHOULD identify the module name, even if there are no conflicting local names when the resource is created. This insures the correct resource will be identified even if the server loads a new module that the old client does not know about.

Examples:

[lines wrapped for display purposes only]

```
/restconf/datastore/example-jukebox:jukebox/library/
   artist/Beatles&select=album

/restconf/datastore/example-list:newlist/
   17&select=nextlist/22/44/acme-list-ext:ext-leaf

/restconf/datastore/example-list:somelist/
   fred%20and%20wilma

/restconf/datastore/example-list:somelist/
   fred%20and%20wilma/address
```

5.3.1.1. ABNF For Data Resource Identifiers

The following ABNF syntax is used to construct RESTCONF path identifiers:

```
api-path = "/" api-identifier
                0*("/" (api-identifier | key-value ))

[FIXME: the syntax for the select string is still TBD]
api-select = api-identifier
                0*("/" (api-identifier | key-value ))

api-identifier = [module-name ":"] identifier

module-name = identifier

key-value = string

;; An identifier MUST NOT start with
;; (('X'|'x') ('M'|'m') ('L'|'l'))
identifier  = (ALPHA / "_")
              *(ALPHA / DIGIT / "_" / "-" / ".")

string = <an unquoted string>
```

5.3.2. Data Resource Retrieval

There are three types of filtering for retrieval of data resources. This section defines each mode.

5.3.2.1. Conditional Retrieval

The HTTP headers (such as "If-Modified-Since" and "If-Match") can by used in for a request message for a GET method to check a condition within the server state, such as the last time the datastore resource was modified, or the resource entity tag of the target resource.

If the condition is met according to the header definition, a "200 OK" Status-Line and the data requested is returned in the response message. If the condition is not met, a "304 Not Modified" Status-Line is returned in response message instead.

5.3.2.2. Data Classification Retrieval

The "config" query parameter can be used with the GET method to specify whether configuration or non-configuration data is requested. Refer to Section 3.8.1 for more details on the "config" query parameter.

5.3.2.3. Filtered Retrieval

The "select" query parameter is used to specify a filter that should be applied to the target resource to request a subset of all possible

descendant nodes within the target resource.

The format of the "select" parameter string is defined in Section 3.8.6. The set of nodes selected by the filter expression is applied to each context node identified by the target resource.

5.4. Operation Resource

An operation resource represents an protocol operation defined with the YANG "rpc" statement.

All operation resources share the same module namespace as any top-level data resources, so the name of an operation resource cannot conflict with the name of a top-level data resource defined within the same module.

If 2 different YANG modules define the same "rpc" identifier, then the module name MUST be used in the request URI. For example, if "module-A" and "module-B" both defined a "reset" operation, then invoking the operation from "module-A" would be requested as follows:

```
POST /restconf/operations/module-A:reset HTTP/1.1
Server example.com
```

Any usage of an operation resource from the same module, with the same name, refers to the same "rpc" statement definition. This behavior can be used to design protocol operations that perform the same general function on different resource types.

If the "rpc" statement has an "input" section, then a message body MAY be sent by the client in the request, otherwise the request message MUST NOT include a message body. If the "rpc" statement has an "output" section, then a message body MAY be sent by the server in the response. Otherwise the server MUST NOT include a message body in the response message, and MUST send a "204 No Content" Status-Line instead.

5.4.1. Encoding Operation Input Parameters

If the "rpc" statement has an "input" section, then the "input" node is provided in the message body, corresponding to the YANG data definition statements within the "input" section.

Example:

The following YANG definition is used for the examples in this section.

```
      rpc reboot {
        input {
          leaf delay {
            units seconds;
            type uint32;
            default 0;
          }
          leaf message { type string; }
          leaf language { type string; }
        }
      }
```

The client might send the following POST request message:

```
   POST /restconf/datastore/operations/
      example-ops:reboot HTTP/1.1
   Host: example.com
   Content-Type: application/vnd.yang.data+json

   {
     "input" : {
       "delay" : 600,
       "message" : "Going down for system maintenance",
       "language" : "en-US"
     }
   }
```

The server might respond:

```
   HTTP/1.1 204 No Content
   Date: Mon, 25 Apr 2012 11:01:00 GMT
   Server: example-server
```

5.4.2. Encoding Operation Output Parameters

If the "rpc" statement has an "output" section, then the "output" node is provided in the message body, corresponding to the YANG data definition statements within the "output" section.

Example:

The following YANG definition is used for the examples in this section.

```
      rpc get-reboot-info {
        output {
          leaf reboot-time {
            units seconds;
            type uint32;
          }
          leaf message { type string; }
          leaf language { type string; }
        }
      }
```

The client might send the following POST request message:

```
   POST /restconf/datastore/operations/example-ops:get-reboot-info
      HTTP/1.1
   Host: example.com
   Content-Type: application/vnd.yang.data+json
```

The server might respond:

```
   HTTP/1.1 200 OK
   Date: Mon, 25 Apr 2012 11:10:30 GMT
   Server: example-server

   {
     "output" : {
       "reboot-time" : 30,
       "message" : "Going down for system maintenance",
       "language" : "en-US"
     }
   }
```

5.5. Event Resource

[TBD]

6. Error Reporting

HTTP Status-Lines are used to report success or failure for RESTCONF operations. The <rpc-error> element returned in NETCONF error responses contains some useful information. This error information is adapted for use in RESTCONF, and error information is returned for "4xx" class of status codes.

The following table summarizes the return status codes used specifically by RESTCONF operations:

Status-Line	Description
100 Continue	POST accepted, 201 should follow
200 OK	Success with response body
201 Created	POST to create a resource success
202 Accepted	POST to create a resource accepted
204 No Content	Success without response body
304 Not Modified	Conditional operation not done
400 Bad Request	Invalid request message
403 Forbidden	Access to resource denied
404 Not Found	Resource target or resource node not found
405 Method Not Allowed	Method not allowed for target resource
409 Conflict	Resource or lock in use
413 Request Entity Too Large	too-big error
414 Request-URI Too Large	too-big error
415 Unsupported Media Type	non RESTCONF media type
500 Internal Server Error	operation-failed
501 Not Implemented	unknown-operation
503 Service Unavailable	Recoverable server error

HTTP Status Codes used in RESTCONF

Since an operation resource is defined with a YANG "rpc" statement, a mapping between the NETCONF <error-tag> value and the HTTP status code is needed. The specific error condition and response code to use are data-model specific and might be contained in the YANG "description" statement for the "rpc" statement.

<error-tag>	status code
in-use	409
invalid-value	400
too-big	413
missing-attribute	400
bad-attribute	400
unknown-attribute	400
bad-element	400
unknown-element	400
unknown-namespace	400
access-denied	403
lock-denied	409
resource-denied	409
rollback-failed	500
data-exists	409
data-missing	409
operation-not-supported	501
operation-failed	500
partial-operation	500
malformed-message	400

Mapping from error-tag to status code

6.1. Error Response Message

When an error occurs for a request message on a data resource or an operation resource, and a "4xx" class of status codes (except for status code "403"), then the server SHOULD send a response body containing the information described by the "errors" container definition within the YANG module Section 8.

Example:

The following example shows an error returned for an "lock-denied" error on a datastore resource.

```
POST /restconf/operations/example-ops:lock-datastore HTTP/1.1
Host: example.com
```

The server might respond:

```
HTTP/1.1 409 Conflict
Date: Mon, 23 Apr 2012 17:11:00 GMT
Server: example-server
Content-Type: application/vnd.yang.api+json

{
  "errors": {
    "error": {
      "error-type": "protocol",
      "error-tag": "lock-denied",
      "error-message": "Lock failed, lock is already held",
    }
  }
}
```

7. YANG Patch

The YANG Patch operation is provided so complex editing operations can be performed within RESTCONF. The "plain patch" operation only provides a simple merge edit operation on the target datastore.

A "YANG Patch" is an ordered list of edits that are applied to the target datastore by the server. The specific fields are defined with the 'yang-patch' container definition in the YANG module Section 8.

Each patch is identified by a client provided string, called the "patch-id".

The client can control the type of error handling that should be applied to the list of supplied edits.

7.1. Why not use JSON Patch?

The RESTCONF PATCH method requires that the media type of the patch content be specified, so it should be possible to use any patch mechanism, including JSON Patch [RFC6902].

The RESTCONF protocol is designed to utilize the YANG data modelling language to specify content schemas. The JSON Patch mechanism is incompatible with RESTCONF for the following reasons:

- A patch mechanism that works with either XML or JSON encoding is needed.

- YANG configuration nodes can be named with complex keys, using one or more key leafs. JSON arrays are packed and all the YANG keys would be collapsed down to a single integer index.

- YANG configuration nodes are named with stable, persistent identifiers, using key leafs. JSON arrays are packed, and if entry I is added or deleted, then all entries I+1 .. Imax are renumbered.

- The edit operation set needs to align with the NETCONF protocol, and JSON Patch does not provide an aligned set of edit operations.

- The datastore validation procedures need to be specific and aligned with YANG validation procedures.

- The error reporting needs to align with the NETCONF protocol, and JSON Patch does not provide an aligned error reporting mechanism.

7.2. YANG Patch Target Data Node

The target data node for each edit operation is determined by the value of the target resource in the request and the "target" leaf within each "edit" entry.

If the target resource specified in the request URI identifies a datastore resource, then the path string in the "target" leaf is an absolute path expression. The first node specified in the "target" leaf is a top-level data node defined within a YANG module.

If the target resource specified in the request URI identifies a data resource, then the path string in the "target" leaf is a relative path expression. The first node specified in the "target" leaf is a child node of the data node associated with the target resource.

7.3. YANG Patch Edit Operations

Each YANG patch edit specifies one edit operation on the target data node. The set of operations is aligned with the NETCONF edit operations, but also includes some new operations.

Operation	Description
create	create a new data resource if it does not already exist or error
delete	delete a data resource if it already exists or error
insert	insert a new user-ordered data resource
merge	merge the edit value with the target data resource; create if it does not already exist
move	re-order the target data resource
replace	replace the target data resource with the edit value
remove	remove a data resource if it already exists or no error

YANG Patch Edit Operations

7.4. YANG Patch Error Handling

There are three error handling modes available that the server MUST support. These modes specify how the server will behave when errors occur in the processing of each edit operation. Note that the server MUST ensure that a well-formed message is received and that the supplied message body conforms to the YANG schema definition for the "patch" container, defined in the YANG module Section 8.

If a well-formed, schema-valid YANG Patch message is received, then then the server will process the supplied edits in ascending order. The following error modes apply to the processing of this edit list:

- all-or-none: All the specified edits MUST be applied or the target datastore contents SHOULD be returned to its original state before the PATCH method started. The server MAY fail to restore the contents of the target datastore completely and with certainty. It is possible for a rollback to fail or and "undo" operation to fail.

- stop-on-error: Each edit will be attempted in order and if an error occurs, the the server will stop processing the edit list and return an error report identifying the edit list entry that caused the error.

- continue-on-error: Each edit will be attempted in order and if an error occurs, the the server will record an error identifying the edit list entry that caused the error, and continue to the next edit entry.

The server will save the running datastore to non-volatile storage if it has changed, after the edits have been attempted.

7.5. YANG Patch Response

A special response is returned for YANG Patch operations, in order to report status information for each individual edit. It is possible to report general errors as well. The YANG conceptual container definition "yang-patch-status" defined in Section 8 is used to define the syntax.

7.6. YANG Patch Examples

7.6.1. Continue-on-error Example

The following example shows several songs being added to an existing album.

- Each edit contains one song.

- The first song already exists, so an error will be reported for that edit.

- The error-action is continue-on-error, so the rest of the songs will be added without error.

Request from client:

```
PATCH /restconf/datastore/example-jukebox:jukebox/
   library/artist/Foo%20Fighters/album/Wasting%20Light HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.patch+json

{
  "yang-patch" : {
    "patch-id" : "add-songs-patch",
    "error-action" : "continue-on-error",
    "edit" : [
      {
        "edit-id" : 1,
        "operation" : "create",
        "target" : "/song",
        "value" : {
          "song" : {
            "name" : "Bridge Burning",
            "location" : "/media/bridge_burning.mp3",
            "format" : "MP3",
            "length" : 288
          }
        }
      },
      {
        "edit-id" : 2,
        "operation" : "create",
        "target" : "/song",
        "value" : {
          "song" : {
            "name" : "Rope",
            "location" : "/media/rope.mp3",
            "format" : "MP3",
            "length" : 259
          }
        }
      },
      {
        "edit-id" : 3,
        "operation" : "create",
        "target" : "/song",
        "value" : {
          "song" : {
            "name" : "Dear Rosemary",
            "location" : "/media/dear_rosemary.mp3",
            "format" : "MP3",
            "length" : 269
```

```
              }
            }
          }
        ]
      }
    }
```

Response from server:

```
  HTTP/1.1 409 Conflict
  Date: Mon, 23 Apr 2012 13:01:20 GMT
  Server: example-server
  Last-Modified: Mon, 23 Apr 2012 13:01:20 GMT
  Content-Type: application/vnd.yang.api+json

  {
    "yang-patch-status" : {
      "patch-id" : "add-songs-patch",
      "edit-status" : {
        "edit" : [
          {
            "edit-id" : 1,
            "errors" : {
              "error" : {
              "error-type": "application",
              "error-tag": "data-exists",
              "error-path": "/example-jukebox:jukebox/library/artist/
                 Foo%20Fighters/album/Wasting%20Light/song/
                 Burning%20Light",
              "error-message": "Data already exists, cannot be
               created"
            }
          },
          {
            "edit-id" : 2,
            "location" : "http://example.com/restconf/
               datastore/example-jukebox:jukebox/library/artist/
               Foo%20Fighters/album/Wasting%20Light/song/Rope"
          },
          {
            "edit-id" : 3,
            "location" : "http://example.com/restconf/
               datastore/example-jukebox:jukebox/library/artist/
               Foo%20Fighters/album/Wasting%20Light/song/
               Dear%20Rosemary"
          }
        }
      ]
```

```
      }
    }
  }
```

7.6.2. Move list entry example

The following example shows a song being moved within an existing playlist. Song "1" in playlist "Foo-One" is being moved after song "3" in the playlist. The operation succeeds, so a non-error reply example can be shown.

Request from client:

```
PATCH /restconf/datastore/example-jukebox:jukebox/
  playlist/Foo-One   HTTP/1.1
Host: example.com
Content-Type: application/vnd.yang.patch+json

{
  "yang-patch" : {
    "patch-id" : "move-song-patch",
    "error-action" : "all-or-none",
    "edit" : [
      {
        "edit-id" : 1,
        "operation" : "move",
        "target" : "/song/1",
        "point" : "/song3",
        "where" : "after"
      }
    ]
  }
}
```

Response from server:

```
HTTP/1.1 400 OK
Date: Mon, 23 Apr 2012 13:01:20 GMT
Server: example-server
Last-Modified: Mon, 23 Apr 2012 13:01:20 GMT
Content-Type: application/vnd.yang.api+json

{
  "yang-patch-status" : {
    "patch-id" : "move-song-patch",
    "edit-status" : {
      "edit" : [
        {
          "edit-id" : 1,
          "ok" : [ null ]
        }
      }
    ]
  }
}
```

8. RESTCONF module

The "ietf-restconf" module defines conceptual definitions within groupings, which are not meant to be implemented as datastore contents by a server.

The "ietf-yang-types" and "ietf-inet_types" modules from [RFC6991] are used by this module for some type definitions.

RFC Ed.: update the date below with the date of RFC publication and remove this note.

<CODE BEGINS> file "ietf-restconf@2013-09-09.yang"

```
module ietf-restconf {
  // RFC Ed.: replace XXXX with 'ietf' and remove this note
  namespace "urn:XXXX:params:xml:ns:yang:ietf-restconf";
  prefix "restconf";

  import ietf-yang-types { prefix yang; }
  import ietf-inet-types { prefix inet; }

  organization
    "IETF NETCONF (Network Configuration) Working Group";

  contact
    "Editor:   Andy Bierman
               <mailto:andy@yumaworks.com>

     Editor:   Martin Bjorklund
               <mailto:mbj@tail-f.com>

     Editor:   Kent Watsen
               <mailto:kwatsen@juniper.net>

     Editor:   Rex Fernando
               <mailto:rex@cisco.com>";

  description
    "This module contains conceptual YANG specifications
     for the YANG Patch and error content that is used in
     RESTCONF protocol messages. A conceptual container
     representing the RESTCONF API nodes (type vnd.yang.api).

     Note that the YANG definitions within this module do not
     represent configuration data of any kind.
     The YANG grouping statements provide a normative syntax
     for XML and JSON message encoding purposes.
```

```
      Copyright (c) 2013 IETF Trust and the persons identified as
      authors of the code.  All rights reserved.

      Redistribution and use in source and binary forms, with or
      without modification, is permitted pursuant to, and subject
      to the license terms contained in, the Simplified BSD License
      set forth in Section 4.c of the IETF Trust's Legal Provisions
      Relating to IETF Documents
      (http://trustee.ietf.org/license-info).

      This version of this YANG module is part of RFC XXXX; see
      the RFC itself for full legal notices.";

    // RFC Ed.: replace XXXX with actual RFC number and remove this
    // note.

    // RFC Ed.: remove this note
    // Note: extracted from draft-bierman-netconf-restconf-01.txt

    // RFC Ed.: update the date below with the date of RFC publication
    // and remove this note.
    revision 2013-09-09 {
      description
        "Initial revision.";
      reference
        "RFC XXXX: RESTCONF Protocol.";
    }

    typedef data-resource-identifier {
      type string {
        length "1 .. max";
      }
      description
        "Contains a Data Resource Identifier formatted string
         to identify a specific data node.";
      reference
        "RFC XXXX: [sec. 5.3.1.1 ABNF For Data Resource Identifiers]";
    }

    // this typedef is TBD; not currently used
    typedef datastore-identifier {
      type union {
        type enumeration {
          enum candidate {
            description
              "Identifies the NETCONF shared candidate datastore.";
            reference
              "RFC 6241, section 8.3";
```

```
          }
          enum running {
            description
              "Identifies the NETCONF running datastore.";
            reference
              "RFC 6241, section 5.1";
          }
          enum startup {
            description
              "Identifies the NETCONF startup datastore.";
            reference
              "RFC 6241, section 8.7";
          }
        }
        type string;
      }
      description
        "Contains a string to identify a specific datastore.
         The enumerated datastore identifier values are
         reserved for standard datastore names.";
    }

    typedef revision-identifier {
      type string {
        pattern '\d{4}-\d{2}-\d{2}';
      }
      description
        "Represents a specific date in YYYY-MM-DD format.
         TBD: make pattern more precise to exclude leading zeros.";
    }

    grouping yang-patch {

      description
        "A grouping that contains a YANG container
         representing the syntax and semantics of a
         YANG Patch edit request message.";

      container yang-patch {
        description
          "Represents a conceptual sequence of datastore edits,
           called a patch. Each patch is given a client-assigned
           patch identifier. A patch is applied with client-specified
           error handling to control how the ordered list of edits
           is applied if an error is encountered.

           A patch MUST be validated by the server to be a
           well-formed message before any of the patch edits
```

```
            are validated or attempted.

            The validation model for patches closely follows
            the constraint enforcement model in YANG, except it
            is conceptually enforced on an ordered list of edits.

            The server MUST conceptually perform field validation
            for each edit in ascending order, as defined in RFC 6020,
            section 8.3.1 and 8.3.2.  This is most relevant if the edit
            error-action is 'stop-on-error', since the identification
            of the first error determines where edit processing is
            terminated.

            If YANG datastore validation (defined in RFC 6020, section
            8.3.3) is required, then it performed after all edits have
            been individually validated.

            It is possible for a datastore constraint violation to occur
            due to any node in the datastore, including nodes not
            included in the edit list. Any validation errors SHOULD
            be reported in the reply message.

            If datastore validation is required and fails, the server
            SHOULD NOT allow the datastore to remain invalid. It is an
            implementation-specific matter how the server fixes the
            invalid datastore.  For example, the server might prune
            invalid nodes causing the datastore validation error,
            or undo the entire patch.";

          reference
            "RFC 6020, section 8.3.";

          leaf patch-id {
            type string;
            description
              "An arbitrary string provided by the client to identify
               the entire patch.  This value SHOULD be present in any
               audit logging records generated by the server for the
               patch. Error messages returned by the server pertaining
               to this patch will be identified by this patch-id value.";
          }

          leaf error-action {
            type enumeration {
              enum all-or-none {
                description
                  "The server will apply all edits in the patch only
                   if no errors occur.  If any errors occur then
```

```
              none of the edits will be applied and the
              contents of the target datastore MUST be unchanged.";
          }
          enum stop-on-error {
            description
              "The server will apply edits in the specified order
               and will stop processing edits if any error occurs.
               Any previous edits which were successfully applied
               will remain applied.  No further edits will be
               attempted after the first error is encountered.";
          }
          enum continue-on-error {
            description
              "The server will apply edits in the specified order
               and will continue processing edits if any error
               occurs.";
          }
        }
        default all-or-none;
        description
          "The error handling behavior for the ordered list of
           edits.";
      }

      list edit {
        key edit-id;

        description
          "Represents one edit within the YANG Patch
           request message.";

        leaf edit-id {
          type uint32;
          description
            "Arbitrary integer index for the edit.
             The server MUST process edits in ascending order.
             Error messages returned by the server pertaining
             to a specific edit will be identified by this
             identifier value.";
        }

        leaf operation {
          type enumeration {
            enum create {
              description
                "The target data node is created using the
                 supplied value, only if it does not already
                 exist.";
```

```
            }
            enum delete {
              description
                "Delete the target node, only if the data resource
                 currently exists, otherwise return an error.";
            }
            enum insert {
              description
                "Insert the supplied value into a user-ordered
                 list or leaf-list entry. The target node must
                 represent a new data resource.";
            }
            enum merge {
              description
                "The supplied value is merged with the target data
                 node.";
            }
            enum move {
              description
                "Move the target node. Reorder a user-ordered
                 list or leaf-list. The target node must represent
                 an existing data resource.";
            }
            enum replace {
              description
                "The supplied value is used to replace the target
                 data node.";
            }
            enum remove {
              description
                "Delete the target node if it currently exists.";
            }
          }
          mandatory true;
          description
            "The datastore operation requested for the associated
             edit entry";
        }

        leaf target {
          type data-resource-identifier;
          mandatory true;
          description
            "Identifies the target data resource for the edit
             operation.";
        }

        leaf point {
```

```
            when "(../operation = 'insert' or " +
              "../operation = 'move') and " +
              "(../where = 'before' or ../where = 'after')" {
              description
                "Point leaf only applies for insert or move
                 operations, before or after an existing entry.";
            }
            type data-resource-identifier;
            description
              "The absolute URL path for the data node that is being
               used as the insertion point or move point for the
               target of this edit entry.";
          }

          leaf where {
            when "../operation = 'insert' or ../operation = 'move'" {
              description
                "Where leaf only applies for insert or move
                 operations.";
            }
            type enumeration {
              enum before {
                description
                  "Insert or move a data node before the data resource
                   identified by the 'point' parameter.";
              }
              enum after {
                description
                  "Insert or move a data node after the data resource
                   identified by the 'point' parameter.";
              }
              enum first {
                description
                  "Insert or move a data node so it becomes ordered
                   as the first entry.";
              }
              enum last {
                description
                  "Insert or move a data node so it becomes ordered
                   as the last entry.";
              }

            }
            default last;
            description
              "Identifies where a data resource will be inserted or
               moved. YANG only allows these operations for
               list and leaf-list data nodes that are ordered-by
```

```
              user.";
          }

          anyxml value {
            when "(../operation = 'create' or " +
              "../operation = 'merge' " +
              "or ../operation = 'replace' or " +
              "../operation = 'insert')" {
              description
                "Value node only used for create, merge,
                 replace, and insert operations";
            }
            description
              "Value used for this edit operation.";
          }
        }
      }

    } // grouping yang-patch


    grouping yang-patch-status {

      description
        "A grouping that contains a YANG container
         representing the syntax and semantics of
         YANG Patch status response message.";

      container yang-patch-status {
        description
          "A container representing the response message
           sent by the server after a YANG Patch edit
           request message has been processed.";

        leaf patch-id {
          type string;
          description
            "The patch-id value used in the request";
        }

        container global-errors {
          uses errors;
          description
            "This container will be present if global
             errors unrelated to a specific edit occurred.";
        }

        container edit-status {
```

```
            description
              "This container will be present if there are
               edit-specific status responses to report.";

            list edit {
              key edit-id;

              description
                "Represents a list of status responses,
                 corresponding to edits in the YANG Patch
                 request message.";

              leaf edit-id {
                type uint32;
                 description
                   "Response status is for the edit list entry
                    with this edit-id value.";
              }
              choice edit-status-choice {
                description
                  "A choice between different types of status
                   responses for each edit entry.";

                leaf ok {
                  type empty;
                  description
                    "This edit entry was invoked without any
                     errors detected by the server associated
                     with this edit.";
                }
                leaf location {
                  type inet:uri;
                  description
                    "Contains the Location header value that would be
                     returned if this edit causes a new resource to be
                     created. If the edit identified by the same edit-id
                     value was successfully invoked and a new resource
                     was created, then this field will be returned
                     instead of 'ok'.";
                }
                leaf skipped {
                  type empty;
                  description
                    "This edit entry was skipped or not reached
                     by the server.";
                }
                case errors {
                  uses errors;
```

```
                description
                  "The server detected errors associated with the
                    edit identified by the same edit-id value.";
              }
            }
          }
        }
      }
    }  // grouping yang-patch-status

    grouping errors {

      description
        "A grouping that contains a YANG container
         representing the syntax and semantics of a
         YANG Patch errors report within a response message.";

      container errors {
        config false;  // needed so list error does not need a key
        description
          "Represents an error report returned by the server if
           a request results in an error.";

        list error {
          description
            "An entry containing information about one
             specific error that occurred while processing
             a RESTCONF request.";
          reference "RFC 6241, Section 4.3";

          leaf error-type {
            type enumeration {
              enum transport {
                description "The transport layer";
              }
              enum rpc {
                description "The rpc or notification layer";
              }
              enum protocol {
                description "The protocol operation layer";
              }
              enum application {
                description "The server application layer";
              }
            }
            mandatory true;
            description
              "The protocol layer where the error occurred.";
```

```
          }

          leaf error-tag {
            type string;
            mandatory true;
            description
              "The enumerated error tag.";
          }

          leaf error-app-tag {
            type string;
            description
              "The application-specific error tag.";
          }

          leaf error-path {
            type data-resource-identifier;
            description
              "The target data resource identifier associated
               with the error, if any.";
          }

          leaf error-message {
            type string;
            description
              "A message describing the error.";
          }

          container error-info {
             description
               "A container allowing additional information
                to be included in the error report.";
             // arbitrary anyxml content here
          }
        }
      }
    } // grouping errors


    grouping restconf {

      description
        "A grouping that contains a YANG container
         representing the syntax and semantics of
         the RESTCONF API resource.";

      container restconf {
        description
```

```
      "Conceptual container representing the vnd.yang.api
       resource type.";

    container datastore {
      description
        "Container representing the vnd.yang.datastore resource
         type. Represents the conceptual root of the unified
         datastore containing YANG data nodes.  The child nodes
         of this container can be data resources (vnd.yang.data)
         defined as top-level YANG data nodes from the modules
         advertised by the server in /restconf/modules.";
    }
    container modules {
      description
        "Contains a list of module description entries.
         These modules are currently loaded into the server.";

      list module {
        key "name revision";
        description
          "Each entry represents one module currently
           supported by the server.";

        leaf name {
          type yang:yang-identifier;
          description "The YANG module name.";
        }
        leaf revision {
          type union {
            type revision-identifier;
            type string { length 0; }
          }
          description
            "The YANG module revision date. An empty string is
             used if no revision statement is present in the
             YANG module.";
        }
        leaf namespace {
          type inet:uri;
          mandatory true;
          description
            "The XML namespace identifier for this module.";
        }
        leaf-list feature {
          type yang:yang-identifier;
          description
            "List of YANG feature names from this module that are
             supported by the server.";
```

```
            }
            leaf-list deviation {
              type yang:yang-identifier;
              description
                "List of YANG deviation module names used by this
                 server to modify the conformance of the module
                 associated with this entry.";
            }
          }
        }
        container operations {
          description
            "Container for all operation resources
             (vnd.yang.operation),

             Each resource is represented as an empty leaf with the
             name of the RPC operation from the YANG rpc statement.

             E.g.;

                POST /restconf/operations/show-log-errors

                leaf show-log-errors {
                  type empty;
                }
            ";
        }
        leaf version {
          type enumeration {
            enum "1.0" {
              description
                "Version 1.0 of the RESTCONF protocol.";
            }
          }
          config false;
          description
            "Contains the RESTCONF protocol version.";
        }
      }
    }  // grouping restconf

}



<CODE ENDS>
```

9. IANA Considerations

9.1. YANG Module Registry

This document registers one URI in the IETF XML registry [RFC3688]. Following the format in RFC 3688, the following registration is requested to be made.

```
// RFC Ed.: replace XXXX with 'ietf' and remove this note
URI: urn:XXXX:params:xml:ns:yang:ietf-restconf
Registrant Contact: The NETMOD WG of the IETF.
XML: N/A, the requested URI is an XML namespace.
```

This document registers one YANG module in the YANG Module Names registry [RFC6020].

```
name:         ietf-restconf
namespace:    urn:ietf:params:xml:ns:yang:ietf-restconf
prefix:       restconf
// RFC Ed.: replace XXXX with RFC number and remove this note
reference:    RFC XXXX
```

10. Security Considerations

TBD

11. Change Log

-- RFC Ed.: remove this section before publication.

11.1. 00 to 01

- Removed incorrect /.well-known URI prefix.
- remove incorrect IANA request for well-known URI.
- Clarified that API resource type nodes are defined in the ietf-restconf namespace.
- changed CamelCase names in example-jukebox.yang to lowercase, and updated examples.
- updated and corrected YANG types in ietf-restconf module.

11.2. YANG-API-01 to RESTCONF-00

- Protocol renamed from YANG-API to RESTCONF
- Fields are clarified. Containers and lists are sub-resources. All other YANG data node types are fields within a parent resource.
- The 'optional-key' YANG extension has been removed.
- The default value is returned by the server if the target resource represents a missing data node but the server is using a default value for the leaf.
- The default for the 'depth' parameter has been changed from '1' to 'unbounded'. The depth is only limited if an integer value for this parameter is specified by the client.
- The default for the 'format' parameter has been changed from 'json' to 'xml'.
- expanded introduction
- removed transactions
- removed capabilities
- removed usage of Range and IfRange headers

- simplified editing model
- removed global protocol operations from ietf-restconf.yang
- changed RPC operation terminology to protocol operation
- updated JSON draft reference
- updated IANA section
- added YANG Patch
- added YANG definitions to ietf-restconf.yang
- added Kent Watsen and Rex Fernando as co-authors
- updated YANG modules so they pass pyang --ietf checking
- changed examples so resource URIs use the module name variant to identify data resources
- changed depth behavior so the entire server contents are not returned for "GET /restconf"; Server will stop at new resource type; e.g. yang.api --> yang.datastore returns the datastore as an empty node; yang.api --> yang.operation returns the operation name as an empty node;

12. Closed Issues

o Which WG should do this work? NETCONF? NETMOD? It is not clear since RESTCONF builds on concepts and standards from documents owned by both working groups.

A: The NETCONF WG would do this work.

o Should sessions be used or not? Should "reusable sessions" be used? Better for auditing? How does locking of the /restconf/ datastore resource work for multiple edits if a session is 1 operation? When does the server release the lock and decide it has been abandoned or client was disconnected?

A: RESTCONF is a session-less protocol. It could be implemented to utilize persistent HTTP connections, but this is not required or designed into the protocol.

o Should the "/restconf/modules" resource within the API resource be a separate resource, with its own timestamp? Currently the API timestamp is coupled to any changes to the list of loaded modules. Should the API resource be static and cacheable?

A: all child containers are considered sub-resources. The server MAY support timestamps and entity IDs for data nodes.

o What to do about no REMOVE operation, just DELETE? The effect is local to the request; in a NETCONF edit-config it is worse, since the netconf request might create/delete/modify many nodes

A: The YANG Patch operation allows remove or delete semantics.

o Should every YANG data node be a data resource and every YANG RPC statement an operation resource? Is a YANG extension needed to allow data modeler control of resource boundaries?

A: Nested containers and lists are considered sub-resources. Terminal nodes (leaf, leaf-list, anyxml) are considered properties of the parent resource.

o Resource creation order and other dependencies between resources are not well identified in YANG. YANG has leafrefs and instance-identifiers, which can be used to identify some order dependencies. Are any new mechanisms needed in RESTCONF needed to identify resource creation order and other dependency requirements?

A: YANG Patch allows the client to control creation order when

multiple resources need to be edited at once. The edit operations allow the server to order all the descendant resources provided by the client, for a single datastore edit target node.

- Encoding of leafrefs? Is there some additional meta-data needed? Do leafref nodes need to be identified in responses (RFC 5988) or is the YANG module definition sufficient to provide this meta-data?

A: no special message encoding of leaf-refs is needed. The server must understand the YANG schema no matter what protocol or encoding is used.

- What should the default algorithm be for defining data resources? Should the default for an augment from another namespace be to start a new resource? Top-level data node defaults as a resource OK?

A: Augmented nodes do not follow different rules than other nested YANG structures. Containers and lists start new sub-resources.

13. Open Issues

- There is no "message-id" field in a RESTCONF message. Is a message identifier needed? If so, should either the "Message-ID" or "Content-ID" header from RFC 2392 be used for this purpose?

- What syntax should be used for the "select" query parameter? The current choices are "xpath" and "path-expr". Perhaps an additional parameter to identify the select string format is needed to allow extensibility?

- Are all header lines used by RESTCONF supported by common application frameworks, such as FastCGI and WSGI? If not, then should query parameters be used instead, since the QUERY_STRING is widely available to WEB applications?

- Should the <errors> element returned in error responses be a separate media type?

- How should additional datastores be supported, which may be added to the NETCONF/NETMOD framework in the future?

- How does a client know which PATCH media types are supported by the server in addition to application/vnd.yang.data and application/vnd.yang.patch?

- Is the /restconf/version field considered meta-data? Should it be returned as XRD (Extensible Resource Descriptor)? In addition or instead of the version field? Should this be the ietf-restconf YANG module revision date, instead of the string 1.0?

- Notification message delivery is TBD

- Alignment between NETCONF and RESTCONF notification is expected to be very close to RFC 5277 design. Additional Sub/pub features still TBD.

- Some sections may need to be rewritten to support notifications and event resources

- Since data resources can only be YANG containers or lists, what should be done about top-level YANG data nodes that are not containers or lists? Are they allowed in RESTCONF?

- Can a choice be a resource? YANG choices are invisible to RESTCONF at this time.

- Does RESTCONF need to Use a .well-known link relation to to re-map API entry point?

The client first discovers the server's root for the RESTCONF API. In this example, it is "/api/restconf":

```
Request
-------
GET /.well-known/host-meta users HTTP/1.1
Host: example.com
Accept: application/xrd+xml

Response
--------
HTTP/1.1 200 OK
Content-Type: application/xrd+xml
Content-Length: nnn


<XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>
    <Link rel='restconf' href='/api/restconf'/>
</XRD>
```

Once discovering the RESTCONF API root, the client MUST prepend it to any access to a RESTCONF resource:

```
Request
-------
GET /api/restconf?select=version&format=json HTTP/1.1
Host: example.com
Accept: application/vnd.yang.api+json

Response
--------
HTTP/1.1 200 OK
Date: Mon, 23 Apr 2012 17:01:00 GMT
Server: example-server
Cache-Control: no-cache
Pragma: no-cache
Last-Modified: Sun, 22 Apr 2012 01:00:14 GMT
Content-Type: application/vnd.yang.api+json

{ "version": "1.0" }
```

14. Example YANG Module

```
module example-jukebox {

   namespace "http://example.com/ns/example-jukebox";
   prefix "jbox";

   organization "Example, Inc.";
   contact "support at example.com";
   description "Example Jukebox Data Model Module";
   revision "2013-09-09" {
     description "Initial version.";
     reference "example.com document 1-4673";
   }

   identity genre {
     description "Base for all genre types";
   }

   // abbreviated list of genre classifications
   identity alternative {
     base genre;
     description "Alternative music";
   }
   identity blues {
     base genre;
     description "Blues music";
   }
   identity country {
     base genre;
     description "Country music";
   }
   identity jazz {
     base genre;
     description "Jazz music";
   }
   identity pop {
     base genre;
     description "Pop music";
   }
   identity rock {
     base genre;
     description "Rock music";
   }

   container jukebox {
     presence
       "An empty container indicates that the jukebox
```

```
           service is available";

        description
          "Represents a jukebox resource, with a library, playlists,
           and a play operation.";

        container library {

          description "Represents the jukebox library resource.";

          list artist {
            key name;

            description
              "Represents one artist resource within the
               jukebox library resource.";

            leaf name {
              type string {
                length "1 .. max";
              }
              description "The name of the artist.";
            }

            list album {
              key name;

              description
                "Represents one album resource within one
                 artist resource, within the jukebox library.";

              leaf name {
                type string {
                  length "1 .. max";
                }
                description "The name of the album.";
              }

              leaf genre {
                type identityref { base genre; }
                description
                  "The genre identifying the type of music on
                   the album.";
              }

              leaf year {
                type uint16 {
                  range "1900 .. max";
```

```
            }
            description "The year the album was released";
          }

          list song {
            key name;

            description
              "Represents one song resource within one
               album resource, within the jukebox library.";

            leaf name {
              type string {
                 length "1 .. max";
              }
              description "The name of the song";
            }
            leaf location {
              type string;
              mandatory true;
              description
               "The file location string of the
                media file for the song";
            }
            leaf format {
              type string;
              description
                "An identifier string for the media type
                 for the file associated with the
                 'location' leaf for this entry.";
            }
            leaf length {
              type uint32;
              units "seconds";
              description
                "The duration of this song in seconds.";
            }
          }   // end list 'song'
        }   // end list 'album'
      }  // end list 'artist'

      leaf artist-count {
         type uint32;
         units "songs";
         config false;
         description "Number of artists in the library";
      }
      leaf album-count {
```

```
            type uint32;
            units "albums";
            config false;
            description "Number of albums in the library";
          }
          leaf song-count {
            type uint32;
            units "songs";
            description "Number of songs in the library";
          }
        }  // end library

        list playlist {
          key name;

          description
            "Example configuration data resource";

          leaf name {
            type string;
            description
              "The name of the playlist.";
          }
          leaf description {
            type string;
            description
              "A comment describing the playlist.";
          }
          list song {
            key index;
            ordered-by user;

            description
              "Example nested configuration data resource";

            leaf index {    // not really needed
              type uint32;
              description
                "An arbitrary integer index for this
                 playlist song.";
            }
            leaf id {
              type instance-identifier;
              mandatory true;
              description
                "Song identifier. Must identify an instance of
                 /jukebox/library/artist/album/song/name.";
            }
```

```
        }
      }

      container player {
        description
          "Represents the jukebox player resource.";

        leaf gap {
          type decimal64 {
            fraction-digits 1;
            range "0.0 .. 2.0";
          }
          units "tenths of seconds";
          description "Time gap between each song";
        }
      }
    }

    rpc play {
      description "Control function for the jukebox player";
      input {
        leaf playlist {
          type string;
          mandatory true;
          description "playlist name";
        }
        leaf song-number {
          type uint32;
          mandatory true;
          description "Song number in playlist to play";
        }
      }
    }
  }
```

15. References

15.1. Normative References

[I-D.lhotka-netmod-json]
Lhotka, L., "Modeling JSON Text with YANG", draft-lhotka-netmod-yang-json-01 (work in progress), April 2013.

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, March 2010.

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

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

[RFC6536] Bierman, A. and M. Bjorklund, "Network Configuration Protocol (NETCONF) Access Control Model", RFC 6536, March 2012.

[RFC6991] Schoenwaelder, J., "Common YANG Data Types", RFC 6991, July 2013.

15.2. Informative References

[RFC6902] Bryan, P. and M. Nottingham, "JavaScript Object Notation (JSON) Patch", RFC 6902, April 2013.

Authors' Addresses

Andy Bierman
YumaWorks

Email: andy@yumaworks.com

Martin Bjorklund
Tail-f Systems

Email: mbj@tail-f.com

Kent Watsen
Juniper Networks

Email: kwatsen@juniper.net

Rex Fernando
Cisco

Email: rex@cisco.com