JSON-RPC

Release master

OpenDaylight Project

Sep 22, 2020

CONTENTS

CHAPTER

ONE

JSON-RPC DEVELOPER GUIDE

1.1 Overview

JSON-RPC ODL extension provides the ability to interface OpenDaylight to other systems by using the JSON-RPC 2.0 protocol.

JSON-RPC 2.0 is a simple and lightweight transport-agnostic protocol that allows remote procedure calls and notifications between two systems. There is a number of implementations for HTTP, Zero MQ <http://zeromq.org>, as well as for other transports. The implementation in OpenDaylight Oxygen provides full support for the Zero MQ transport. Other transports will be added in future revisions.

1.2 Concepts

1. Naming, Schema and Endpoint Configuration
2. RPCs
3. Notifications
4. Data

1.2.1 Naming, Schema, and Endpoint Configuration

JSON-RPC 2.0 is a stateless, lightweight protocol. The specification was designed to allow two entities to easily interoperate with a minimal development effort. JSON-RPC 2.0 is not an enterprise bus specification. It leaves a number of issues out of scope, such as discovery, naming, and so on. It is up to the developer (in this case the JSON-RPC 2.0 ODL team) to implement them for their particular use case.

The OpenDaylight JSON-RPC 2.0 extension implements a set of naming, schema, and endpoint discovery services that have been designed to cater specifically for the needs of interfacing external entities to ODL. These services are built around the concept of data modeling in YANG (same as most of ODL).

In order for ODL to interface to an external system, it must have an appropriate YANG model for what it is talking to. It must also know which endpoints to contact for a particular operation. The information on models and endpoints is obtained by ODL from both configuration and an external service called governance, which is accessed over JSON-RPC 2.0.

Governance is a very simple JSON-RPC 2.0 service that provides two RPCs: one to fetch YANG models and one to reply to requests for mapping a particular function (for example, RPC or Data) to an endpoint.

Both models and endpoints can also be specified in the JSON-RPC 2.0 extension configuration inside ODL. If a configuration is specified, it always takes precedence. In such a case, governance will not be queried for the configured

endpoint. If governance and configuration return null to a request for an endpoint mapping, that endpoint (for example an RPC) will not be configured because there is no information on how to reach it.

Here is an example configuration for the JSON-RPC 2.0 extension that is configured to communicate with a simple echo service:

```
{
    "config":{
        "governance-root":"zmq://127.0.0.1:4570",
        "configured-endpoints":[
            {
                "name":"echo-1",
                "modules":["echo"],
                "rpc-endpoints": [{
                    "path":"{}",
                    "endpoint-uri":"zmq://192.168.97.133:4569"
                }]
            }
        ]
    }
}
```

Notice the `"path"` statement in the above configuration snippet. The JSON-RPC 2.0 extension uses a YANG Instance Identifier like `"path"` to describe which part of the device YANG model an endpoint is responsible for. The syntax is different from RFC7951, because all YANG RFCs limit YIId scope to data only. As a result, the syntax described in RFC7951 cannot be used to for RPC and Notification paths. For this reason, JSON-RPC 2.0 has chosen a different notation. The path in YANG-modelled JSON RPC needs to cater for a number of non-data use cases like RPCs and Notifications. The relevant use cases are covered in detail in the YANG-modelled JSON RPC 2.0 draft at https://tools.ietf.org/html/draft-yang-json-rpc

The configuration example for the echo service uses `"{}"` as a path specifier, which equates to "everything". All RPCs described in the model are mapped at this endpoint and distinguished on the basis of the RPC method that is supplied in each JSON-RPC 2.0 call. For more details, please see the JSON RPC 2.0 draft at https://tools.ietf.org/html/draft-yang-json-rpc

The model for the echo service is as follows:

```
module echo {
    yang-version "1.1";
    namespace "urn:com.inocybe:echo";
    prefix "echo";
    organization "Inocybe Technologies";
    revision "2017-11-06" {
        description "Initial version";
    }
    rpc echo {
        input {
            leaf test {
                type string;
                description "test";
            }
        }
        output {
            leaf test {
                type string;
                description "test";
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

When the previous configuration is added to ODL, this model is requested from the JSON-RPC 2.0 schema service. This service first attempts to locate the model from the ones already loaded in ODL; if that fails, the service asks the governance service. The governance service is configured to be at "zmq://127.0.0.1:4570" in the previous configuration.

A minimal governance service can be extremely simple. If all of the endpoint-to-URI mapping is performed solely through ODL configuration statements, it can be limited to model fetch-over-RPC.

Following is an example of such a service (using Perl, JSON RPC2.0, and ZMQ stacks):

```
#!/usr/bin/perl
#
# Example perl governance script: ./governance.pl URL PATH
# Note - little or no error or security checking is performed,
# this is example only

use strict;

use ZMQ::FFI qw(ZMQ_REQ ZMQ_REP);
use JSON::RPC2::Server;
use JSON::MaybeXS;

if ($socket->bind($ARGV[0])) {
    print STDERR "Errno $!\n";
};

# JSON RPC callback - maps responce to actual transport's send
sub send_response {
    my ($json_response) = @_;
    $socket->send($json_response);
}

# Governance RPC Call stub - always says "no idea" to any requests
sub governance {
    my (@remote_params) = @_;
    return undef; # always return null to any uri-to-path mapping
}

# Source RPC Call - tries to find a model source file and sends it back
# warning - do not use "as is" - this is vulnerable to fs traversal
sub source {
    my (@remote_params) = @_;
    foreach my $modeldir (split(/:/, $MODELS)) {
        $modeldir = $modeldir . "/";
        open(my $fh, "<", $modeldir . $remote_params[0] . ".yang") || next;
        my @model = <$fh>;
        close($fh);
        return "@model";
    }
}

# create the JSON RPC Server
my $srv = JSON::RPC2::Server->new();
# register methods
```

(continues on next page)

(continued from previous page)

```
$srv->register('source', \&source);
$srv->register('governance', \&governance);

# run the json rpc loop
while (my $rpc_in = $socket->recv()) {
    print STDERR "RPC IN $rpc_in \n";
    $srv->execute($rpc_in, \&send_response);
}
```

1.2.2 RPCs

Regarding RPCs, for starters, we can look again at the simplistic governance example in the previous section. This code is also a good simplistic example for an RPC. We have two methods: "source" and "governance" that are registered with the JSON-RPC 2.0 stack. In this case, the implementation is in Perl. There is a multitude of stacks available, such as Java, C, C++, Javasript, Python, and so. Any one of them can be used as long as it can be integrated to use the same transports for which ODL can use (ZMQ in this release, ZMQ and HTTP in future releases).

As long as the service is limited to RPCs and Notifications, the implementation can be very simplistic (such as the examples shown in the previous section). There is no need to make the external code aware of YANG, provided that it can understand the arguments supplied in the remote procedure calls and reply with a correctly formatted JSON result.

An example would be the implementation of the echo service that is described earlier:

```
#!/usr/bin/perl

use strict;

use ZMQ::FFI qw(ZMQ_REQ ZMQ_REP);
use JSON::RPC2::Server;
use JSON::MaybeXS;

my $Ctx = ZMQ::FFI->new();
my $socket = $Ctx->socket(ZMQ_REP);
$socket->bind($ARGV[0]);

sub send_response {
    my ($json_response) = @_;
    print STDERR "RPC OUT:$json_response\n";
    $socket->send($json_response);
}

sub echo {
    my (%remote_params) = @_;
    return (\%remote_params, undef, undef, undef);
}

my $srv = JSON::RPC2::Server->new();
$srv->register_named('echo', \&echo);

while (my $rpc_in = $socket->recv()) {
    print STDERR "RPC IN $rpc_in \n";
    $srv->execute($rpc_in, \&send_response);
}
```

1.2.3 Notifications

Notifications in JSON-RPC 2.0 are virtually identical to RPCs. The difference is that they do not expect a response and do not have the additional fields in the on-the-wire payload needed for that.

A notification service is not very different from a RPC client. It produces a stream of notifications to which the listeners need to subscribe at a specific URL.

1.2.4 Data Endpoints

YANG-modelled data is significantly more complex than RPCs and Notifications. A YANG data endpoint must be capable of understanding a path and mapping it onto its own data structures in order to return correct results. Additionally, a YANG-modelled data endpoint must be transaction aware. It should be able to group the basic data operations that are used by OpenDaylight into a sequence and commit them to the backend only after OpenDaylight has issued a commit request. It should also be able to roll back any accumulated changes.

ODL requires from a data implementation the following RPCs:

1. read(entity, store, path). Perform a data read. If multiple yang modelled entities (devices) are supported by this endpoint, entity can be used to differentiate between them. store is YANG store - configuration or operational. Path is path to the data element to be read in draft-yang-json-rpc form. Returns data in json form.
2. exists(entity, store, path) Same as read, but returns true if the required data element exists; false otherwise.
3. txid() Allocate a new transaction on the JSON-RPC 2.0 server side which ODL can refer to for any future data operations. Returns the string representation of UUID4.
4. put(txid, entity, store, path, data) Create a data element if it does not exist or overwrite an existing element at the location specified by path in the datastore specified by store and associated with entity/managed device specified by entity. Use the data supplied in data. Enqueue the changes to transaction txid. There is no return value from this call as the actual transaction is not verified or executed at this stage.
5. merge(txid, entiry, store, path, data) Change an existing data element at the location specified by path in the datastore specified by store and associated with entity/managed device specified by entity. Use the data supplied in data. Enqueue the changes to transaction txid. There is no return value from this call as the actual transaction is not verified or executed at this stage.
6. delete(txid, entity, store, path) Delete an existing data element at the location specified by path in the the datastore specified by store and associated with entity/managed device specified by entity. Enqueue the changes to transaction txid. There is no return value from this call as the actual transaction is not verified or executed at this stage.
7. commit(txid) Commit all enqueued operations for transaction txid. If any of them fail, the implementation must roll back. Returns true on success, false otherwise.
8. cancel(txid) Cancel all enqueued operations for transaction txid. No incomplete changes must be present in the database. Returns true on success, false otherwise.
9. error(txid) Get extended error information and error messages for a particular txid. Returns a string containing detailed error information.

This API is similar to most simplistic transaction APIs. It reflects the relatively low isolation level of the ODL datastore. Specifically, `read` and `exists` are non-transactional; they use absolute scope and not within a scope of an ongoing transaction.

A JSON-RPC 2.0 service which implements the remote datastore specification must support all operations from 1-8, including the relevant transaction semantics. At present, `error(txid)` is not yet used by the ODL JSON RPC 2.0 extension.

For more examples on data representation, data addressing, and so on, please see the YANG-modelled JSON RPC 2.0 draft at https://tools.ietf.org/html/draft-yang-json-rpc

CHAPTER

TWO

JSON-RPC USER GUIDE

2.1 Overview

JSON-RPC 2.0 is a lightweight remote procedure call and notification specification maintained by http://www.jsonrpc.org/. OpenDaylight uses the YANG-modelled JSON-RPC 2.0 specification as described in the IETF DRAFT: https://tools.ietf.org/html/draft-yang-json-rpc-00

2.2 Supported Transports

The JSON-RPC 2.0 plugin in OpenDaylight supports both ODL as server and as a client with ZMQ, HTTP and Websocket. HTTP can be used only with *requester* and *responder* pattern.

2.3 Additional Information

JSON-RPC 2.0 is primarily intended for developers of interfaces and extensions that are used to connect OpenDaylight to external systems. For additional information, please see the JSON-RPC Developer Guide.