Web UI

Release: NSO 6.2.5

Published: May 17, 2010

Last Modified: May 14, 2024

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

CHAPTER 1 **Web UI Development** 1

Introduction 1

Example of a common flow 2

Example of a JSON-RPC client 6

Example of a Comet client 9

CHAPTER 2 **The JSON-RPC API** 13

JSON-RPC 13

Protocol overview 13

Common concepts 14

FAQ 16

Methods - commands 18

Method get_cmds 18

Method init_cmd 18

Method send_cmd_data 19

Method start_cmd 19

Method suspend_cmd 20

Method resume_cmd 20

Method stop_cmd 20

Methods - commands - subscribe 21

Method get_subscriptions 21

Method subscribe_cdboper 21

Method subscribe_changes 22

Method subscribe_poll_leaf 22

Method subscribe_upgrade 23

Method subscribe_jsonrpc_batch 23

Method subscribe_progress_trace 24

Method start_subscription 25

Method unsubscribe 25

Methods - data 26

Method create 26

Method delete 26

Method exists 26

Method get_case 27

Method show_config 27

Method load 27

Methods - data - attrs 28

Method get_attrs 28

Method set_attrs 28

Methods - data - leafs 28

Method get_value 28

Method get_values 29

Method set_value 30

Methods - data - leafref 31

Method deref 31

Method get_leafref_values 31

Methods - data - lists 32

Method rename_list_entry 32

Method copy_list_entry 33

Method move_list_entry 33

Method append_list_entry 34

Method count_list_keys 34

Method get_list_keys 34

Methods - data - query 35

Method query 35

Method start_query 35

Method run_query 37

Method reset_query 38

Method stop_query 39

Methods - database 39

Method reset_candidate_db 39

Method lock_db 39

Method unlock_db 40

Method copy_running_to_startup_db **40**

Methods - general **40**

Method comet **40**

Method get_system_setting **42**

Method abort **42**

Method eval_XPath **43**

Methods - messages **43**

Method send_message **43**

Method subscribe_messages **43**

Methods - rollbacks **44**

Method get_rollbacks **44**

Method get_rollback **44**

Method install_rollback **45**

Method load_rollback **45**

Methods - schema **46**

Method get_description **46**

Method get_schema **46**

Method hide_schema **50**

Method unhide_schema **50**

Method get_module_prefix_map **50**

Method run_action **51**

Methods - session **53**

Method login **53**

Method challenge_response **55**

Method logout **56**

Method kick_user **57**

Methods - session data **57**

Method get_session_data **57**

Method put_session_data **57**

Method erase_session_data **57**

Methods - transaction **58**

Method get_trans **58**

Method new_trans **58**

Method delete_trans **60**

Method set_trans_comment **60**

Method set_trans_label 60

Methods - transaction - changes 60

Method is_trans_modified 60

Method get_trans_changes 61

Method validate_trans 61

Method get_trans_conflicts 62

Method resolve_trans 62

Methods - transaction - commit changes 63

Method validate_commit 63

Method clear_validate_lock 63

Method commit 64

Method abort_commit 66

Method confirm_commit 66

Methods - transaction - webui 66

Method get_webui_trans 66

Method new_webui_trans 67

Methods - NSO specific 67

Method get_template_variables 67

Method list_packages 67

CHAPTER 3 **The web server** 69

Introduction 69

Web server capabilities 69

CGI support 70

Storing TLS data in database 71

Package upload 73

CHAPTER 4 **Single Sign-On** 75

CHAPTER 1

Web UI Development

- Introduction, page 1
- Example of a common flow, page 2
- Example of a JSON-RPC client, page 6
- Example of a Comet client, page 9

Introduction

Web UI development is thought to be in the hands of the customer's frontend developers - they will know best the requirements and how to fulfill those requirements in terms of esthetics, functionality and tool chain (frameworks, libraries, external data sources and services).

NSO comes with a nortbound interface in the shape of a JSON-RPC API. This API is designed with Web UI applications in mind, and it complies with the JSON-RPC 2.0 specification [https://www.jsonrpc.org/specification], while using HTTP/S as the transport mechanism.

The JSON-RPC API contains a handful of methods with well defined input *method* and *params*, along with the output *result*.

In addition, the API also implements a Comet model, as long polling, to allow the client to subscribe to different server events and receive event notifications about those events in near real time.

You can call these from a browser via AJAX (e.g. XMLHTTPRequest, jQuery [https://jquery.com/]) or from the command line (e.g. curl [https://github.com/bagder/curl], httpie [https://github.com/jkbr/httpie]):

```
// with jQuery
$.ajax({
  type: 'post',
  url: '/jsonrpc',
  contentType: 'application/json',
  data: JSON.stringify({
  jsonrpc: '2.0',
  id: 1,
  method: 'login',
  params: {
    'user': 'joe',
    'passwd': 'SWkkasE32'
  }
  }),
  dataType: 'json'
})
.done(function(data) {
  if (data.result)
  alert(data.result);
  else
  alert(data.error.type);
  });
```

or

```
      # with curl
      curl \
      -X POST \
      -H 'Content-Type: application/json' \
      -d '{"jsonrpc": "2.0", "id": 1,
           "method": "login",
           "params": {"user": "joe",
                      "passwd": "SWkkasE32"}}' \
      http://127.0.0.1:8008/jsonrpc

      # with httpie
      http POST http://127.0.0.1:8008/jsonrpc \
      jsonrpc=2.0 id:=1 \
      method=login \
      params:='{"user": "joe", "passwd": "SWkkasE32"}'
```

Example of a common flow

You can read in the JSON-RPC API chapter about all the available methods and their signatures, but here is a working example of how a common flow would look like:

- login
- create a new read transaction
- read a value
- create a new webui (read-write) transaction, in preparation for changing the value
- change a value
- commit (save) the changes
- meanwhile, subscribe to changes and receive a notification

In the release package, under `${NCS_DIR}/var/ncs/webui/example`, you will find working code to run the example below.

```
      /*jshint devel:true*/
// !!!
// The following code is purely for example purposes.
// The code has inline comments for a better understanding.
// Your mileage might vary.
// !!!

define([
  'lodash',
  'bluebird',
  './JsonRpc',
  './Comet'
], function(
  _,
  Promise,
  JsonRpc,
  Comet
) {
  'use strict';
```

```
// CHANGE AT LEAST THESE
// IN ORDER TO MAKE THIS EXAMPLE WORK IN YOUR SETUP
var jsonrpcUrl = '/jsonrpc', // 'http://localhost:8008/jsonrpc';
    path = '/dhcp:dhcp/max-lease-time',
    value = Math.floor(Math.random() * 800) + 7200;

var log,
    jsonRpc,
    comet,
    funs = {},
    ths = {
      read: undefined,
      webui: undefined
    };

// UTILITY
log = function(msg) {
  document.body.innerHTML =
    '<pre>' +
    msg +
    '</pre>' +
    document.body.innerHTML;
};

// SETUP
jsonRpc = new JsonRpc({
  url: jsonrpcUrl,
  onError: function(method, params, deferred, reply) {
    var error = reply.error,
        msg = [method,
               params,
               reply.id,
               error.code,
               error.type,
               error.message
              ];

    if (method === 'comet') {
      return;
    }

    window.alert('JsonRpc error: ' + msg);
  }
});

comet = new Comet({
  jsonRpc: jsonRpc,
  onError: function(reply) {
    var error = reply.error,
        msg = [reply.id, error.code, error.type, error.message];

    window.alert('Comet error: ' + msg);
  }
});

// CALLS FOR A COMMON SCENARIO
funs.login = function() {
  log('Logging in as admin:admin...');
  return jsonRpc.call('login', {
    user: 'admin',
    passwd: 'admin'
  }).done(function() {
    log('Logged in.');
  });
};

funs.getSystemSetting = function() {
  log('Getting system settings...');
  return jsonRpc.call('get_system_setting').done(function(result) {
    log(JSON.stringify(result, null, 1));
  });
};

funs.newReadTrans = function() {
  log('Create a new read-only transaction...');
  return jsonRpc.call('new_read_trans', {
    db: 'running'
  }).done(function(result) {
    ths.read = result.th;
    log('Read-only transaction with th (transaction handle) id: ' +
        result.th + '.');
  });
};

funs.newWebuiTrans = function() {
  log('Create a new webui (read-write) transaction...');
  return jsonRpc.call('new_webui_trans', {
    conf_mode: 'private',
    db: 'candidate'
  }).done(function(result) {
    ths.webui = result.th;
    log('webui (read-write) transaction with th (transaction handle) id: ' +
        result.th + '.');
  });
};

funs.getValue = function(args /*{th, path}*/) {
  log('Get value for ' + args.path + ' in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('get_value', {
    th: args.th,
    path: path
  }).done(function(result) {
    log(path + ' is now set to: ' + result.value + '.');
  });
};

funs.setValue = function(args /*{th, path, value}*/) {
  log('Set value for ' + args.path +
      ' to ' + args.value +
      ' in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('set_value', {
    th: args.th,
    path: path,
    value: args.value
  }).done(function(result) {
    log(path + ' is now set to: ' + result.value + '.');
  });
```

```
};

funs.validate = function(args /*{th}*/) {
  log('Validating changes in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('validate_commit', {
    th: args.th
  }).done(function() {
    log('Validated.');
  });
};

funs.commit = function(args /*{th}*/) {
  log('Commiting changes in ' + args.th + ' transaction...');
  if (typeof args.th === 'string') {
    args.th = ths[args.th];
  }
  return jsonRpc.call('commit', {
    th: args.th
  }).done(function() {
    log('Commited.');
  });
};

funs.subscribeChanges = function(args /*{path, handle}*/) {
  log('Subcribing to changes of ' + args.path +
      ' (with handle ' + args.handle + ')...');
  return jsonRpc.call('subscribe_changes', {
    comet_id: comet.id,
    path: args.path,
    handle: args.handle,
  }).done(function(result) {
    log('Subscribed with handle id ' + result.handle + '.');
  });
};

// RUN
Promise.resolve([
  funs.login,
  funs.getSystemSetting,
  funs.newReadTrans,
  function() {
    return funs.getValue({th: 'read', path: path});
  },
  function() {
    var handle = comet.id + '-max-lease-time';
    comet.on(handle, function(msg) {
      log('>>> Notification >>>\n' +
          JSON.stringify(msg, null, 2) +
          '\n<<< Notification <<<');
    });
    return funs.subscribeChanges({th: 'read', path: path, handle: handle});
  },
  funs.newWebuiTrans,
  function() {
    return funs.setValue({th: 'webui', path: path, value: value.toString()});
  },
  function() {
    return funs.getValue({th: 'webui', path: path});
  },
    function() {
      return funs.validate({th: 'webui'});
    },
    function() {
      return funs.commit({th: 'webui'});
    },
    function() {
      return new Promise(function(resolve) {
        log('Waiting 2.5 seconds before one last call to get_value');
        window.setTimeout(function() {
          resolve();
        }, 2500);
      });
    },
    function() {
      return funs.getValue({th: 'read', path: path});
    },
  ]).each(function(fn){
    return fn().then(function() {
      log('--------------------------------------------------');
    });
  });
});


// Local Variables:
// mode: js
// js-indent-level: 2
// End:
```

Example of a JSON-RPC client

In the example above describing a common flow, a reference is made to using a JSON-RPC client to make the RPC calls.

An example implementation of a JSON-RPC client, used in the example above:

```
      /*jshint devel:true*/
// !!!
// The following code is purely for example purposes.
// The code has inline comments for a better understanding.
// Your mileage might vary.
// !!!

define([
  'jquery',
  'lodash'
], function(
  $,
  _
) {
  'use strict';

  var JsonRpc;

  JsonRpc = function(params) {
    $.extend(this, {
      // API

      // Call a JsonRpc method with params
      call: undefined,
      // API (OPTIONAL)

      // Decide what to do when there is no active session
      onNoSession: undefined,
      // Decide what to do when the request errors
      onError: undefined,
      // Set an id to start using in requests
      id: 0,
      // Set another url for the JSON-RPC API
      url: '/jsonrpc',


      // INTERNAL

      makeOnCallDone: undefined,
      makeOnCallFail: undefined
    }, params || {});

    _.bindAll(this, [
      'call',
      'onNoSession',
      'onError',
      'makeOnCallDone',
      'makeOnCallFail'
    ]);
  };

  JsonRpc.prototype = {
    call: function(method, params, timeout) {
      var deferred = $.Deferred();

      // Easier to associate request/response logs
      // when the id is unique to each request
      this.id = this.id + 1;

      $.ajax({
        // HTTP method is always POST for the JSON-RPC API
        type: 'POST',
        // Let's show <method> rather than just "jsonrpc"
        // in the browsers' Developer Tools - Network tab - Name column
        url: this.url + '/' + method,
        // Content-Type is mandatory
        // and is always "application/json" for the JSON-RPC API
        contentType: 'application/json',
        // Optionally set a timeout for the request
        timeout: timeout,
        // Request payload
        data: JSON.stringify({
          jsonrpc: '2.0',
          id: this.id,
          method: method,
          params: params
        }),
        dataType: 'json',
        // Just in case you are doing cross domain requests
        // NOTE: make sure you are setting CORS headers similarly to
        // --
        // Access-Control-Allow-Origin: http://server1.com, http://server2.com
        // Access-Control-Allow-Credentials: true
        // Access-Control-Allow-Headers: Origin, Content-Type, Accept
        // Access-Control-Request-Method: POST
        // --
```

```
        // if you want to allow JSON-RPC calls from server1.com and server2.com
        crossDomain: true,
        xhrFields: {
          withCredentials: true
        }
      })
      // When done, or on failure,
      // call a function that has access to both
      // the request and the response information
      .done(this.makeOnCallDone(method, params, deferred))
      .fail(this.makeOnCallFail(method, params, deferred));
      return deferred.promise();
    },

    makeOnCallDone: function(method, params, deferred) {
      var me = this;

      return function(reply/*, status, xhr*/) {
        if (reply.error) {
          return me.onError(method, params, deferred, reply);
        }
        deferred.resolve(reply.result);
      };
    },

    onNoSession: function() {
      // It is common practice that when missing a session identifier
      // or when the session crashes or it times out due to inactivity
      // the user is taken back to the login page
      _.defer(function() {
        window.location.href = 'login.html';
      });
    },

    onError: function(method, params, deferred, reply) {
      if (reply.error.type === 'session.missing_sessionid' ||
          reply.error.type === 'session.invalid_sessionid') {
        this.onNoSession();
      }

      deferred.reject(reply.error);
    },

    makeOnCallFail: function(method, params, deferred) {
      return function(xhr, status, errorMessage) {
        var error;

        error = $.extend(new Error(errorMessage), {
          type: 'ajax.response.error',
          detail: JSON.stringify({method: method, params: params})
        });

        deferred.reject(error);
      };
    }
  };

  return JsonRpc;
});

// Local Variables:
// mode: js
// js-indent-level: 2
// End:
```

Example of a Comet client

In the example above describing a common flow, a reference is made to starting a Comet channel and subscribing to changes on a specific path.

An example implementation of a Comet client, used in the example above:

```
      /*jshint devel:true*/
// !!!
// The following code is purely for example purposes.
// The code has inline comments for a better understanding.
// Your mileage might vary.
// !!!

define([
  'jquery',
  'lodash',
  './JsonRpc'
], function(
  $,
  _,
  JsonRpc
) {
  'use strict';

  var Comet;

  Comet = function(params) {
    $.extend(this, {
      // API

      // Add a callback for a notification handle
      on: undefined,
      // Remove a specific callback or all callbacks for a notification handle
      off: undefined,
      // Stop all comet notifications
      stop: undefined,

      // API (OPTIONAL)

      // Decide what to do when the comet errors
      onError: undefined,
      // Optionally set a different id for this comet channel
      id: 'main-1.' + String(Math.random()).substring(2),
      // Optionally give an existing JsonRpc client
      jsonRpc: new JsonRpc(),
      // Optionally wait 1 second in between polling the comet channel
      sleep: 1 * 1000,

      // INTERNAL

      handlers: [],
      polling: false,
      poll: undefined,
      onPollDone: undefined,
      onPollFail: undefined
    }, params || {});

    _.bindAll(this, [
      'on',
      'off',
      'stop',
      'onError',
      'poll',
      'onPollDone',
      'onPollFail'
    ]);
  };

  Comet.prototype = {
    on: function(handle, callback) {
      if (!callback) {
        throw new Error('Missing a callback for handle ' + handle);
      }

      // Add a handler made of handle id and a callback function
      this.handlers.push({handle: handle, callback: callback});

      // Start polling
      _.defer(this.poll);
    },

    off: function(handle, callback) {
      if (!handle) {
        throw new Error('Missing a handle');
      }

      // Remove all handlers matching the handle,
      // and optionally also the callback function
      _.remove(this.handlers, {handle: handle, callback: callback});

      // If there are no more handlers matching the handle,
      // then unsubscribe from notifications, in order to releave
      // the server and the network from redundancy
      if (!_.find(this.handlers, {handle: handle}).length) {
        this.jsonRpc.call('unsubscribe', {handle: handle});
      }
    },

    stop: function() {
      var me = this,
          deferred = $.Deferred(),
          deferreds = [];

      if (this.polling) {
        // Unsubcribe from all known notifications, in order to releave
        // the server and the network from redundancy
        _.each(this.handlers, function(handler) {
          deferreds.push(me.jsonRpc('unsubscribe', {
            handle: handler.handle
          }));
        });

        $.when.apply($, deferreds).done(function() {
          deferred.resolve();
        }).fail(function(err) {
          deferred.reject(err);
        }).always(function() {
          me.polling = false;
          me.handlers = [];
        });
      } else {
        deferred.resolve();
      }

      return deferred.promise();
    },

    poll: function() {
      var me = this;

      if (this.polling) {
        return;
      }

      this.polling = true;

      this.jsonRpc.call('comet', {
        comet_id: this.id
      }).done(function(notifications) {
        me.onPollDone(notifications);
      }).fail(function(err) {
        me.onPollFail(err);
      }).always(function() {
        me.polling = false;
      });
    },

    onPollDone: function(notifications) {
      var me = this;

      // Polling has stopped meanwhile
      if (!this.polling) {
        return;
      }

      _.each(notifications, function(notification) {
        var handle = notification.handle,
            message = notification.message,
            handlers = _.where(me.handlers, {handle: handle});

        // If we received a notification that we cannot handle,
        // then unsubcribe from it, in order to releave
        // the server and the network from redundancy
        if (!handlers.length) {
          return this.jsonRpc.call('unsubscribe', {handle: handle});
        }

        _.each(handlers, function(handler) {
          _.defer(function() {handler.callback(message);});
        });
      });

      _.defer(this.poll);
    },

    onPollFail: function(error) {
      switch (error.type) {
        case 'comet.duplicated_channel':
        this.onError(error);
        break;

        default:
        this.onError(error);
        _.wait(this.poll, this.sleep);
      }
    },

    onError: function(reply) {
      var error = reply.error,
          msg = [reply.id, error.code, error.type, error.message].join(' ');

      console.error('Comet error: ' + msg);
    }
  };

  return Comet;
});

// Local Variables:
// mode: js
// js-indent-level: 2
// End:
```

CHAPTER 2

The JSON-RPC API

- JSON-RPC, page 13
- Methods - commands, page 18
- Methods - commands - subscribe, page 21
- Methods - data, page 26
- Methods - data - attrs, page 28
- Methods - data - leafs, page 28
- Methods - data - leafref, page 31
- Methods - data - lists, page 32
- Methods - data - query, page 35
- Methods - database, page 39
- Methods - general, page 40
- Methods - messages, page 43
- Methods - rollbacks, page 44
- Methods - schema, page 46
- Methods - session, page 53
- Methods - session data, page 57
- Methods - transaction, page 58
- Methods - transaction - changes, page 60
- Methods - transaction - commit changes, page 63
- Methods - transaction - webui, page 66
- Methods - NSO specific, page 67

JSON-RPC

Protocol overview

The JSON-RPC 2.0 Specification [https://www.jsonrpc.org/specification] contains all the details you need in order to understand the protocol but here is the short version.

A request payload typically looks like this:

```
{"jsonrpc": "2.0",
 "id": 1,
 "method": "subtract",
 "params": [42, 23]}
```

where the *method* and *params* properties are as defined in this manual page.

A response payload typically looks like this:

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": 19}
```

or

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32601,
   "type": "rpc.request.method.not_found",
   "message": "Method not found"}}
```

The request *id* param is returned as-is in the response to make it easy to pair requests and responses.

The batch JSON-RPC standard is dependent on matching requests and responses by *id*, since the server processes requests in any order it sees fit e.g.:

```
[{"jsonrpc": "2.0",
  "id": 1,
  "method": "subtract",
  "params": [42, 23]}
,{"jsonrpc": "2.0",
  "id": 2,
  "method": "add",
  "params": [42, 23]}]
```

with a possible response like (first result for "add", second result for "substract"):

```
[{"jsonrpc": "2.0",
  "id": 2,
  "result": 65}
,{"jsonrpc": "2.0",
  "id": 1,
  "result": 19}]
```

Common concepts

The URL for the JSON-RPC API is `/jsonrpc`. For logging and debugging purposes, you can add anything as a subpath to the URL, for example turning the URL into `/jsonrpc/<method>` which will allow you to see the exact method in different browsers' *Developer Tools* - *Network* tab - *Name* column, rather than just an opaque "jsonrpc".

For brevity, in the upcoming descriptions of each methods, only the input *params* and the output *result* are mentioned, although they are part of a fully formed JSON-RPC payload.

Authorization is based on HTTP cookies. The response to a successful call to *login* would create a session, and set a HTTP-only cookie, and even a HTTP-only secure cookie over HTTPS, named *sessionid*. All subsequent calls are authorized by the presence and the validity of this cookie.

The *th* param is a transaction handle identifier as returned from a call to *new_read_trans* or *new_write_trans*.

The *comet_id* param is a unique id (decided by the client) which must be given first in a call to the *comet* method, and then to upcoming calls which trigger comet notifications.

The *handle* param needs to a semantic value (not just a counter) prefixed with the comet id (for disambiguation), and overrides the handle that would have otherwise been returned by the call. This gives more freedom to the client and set semantic handles.

Common errors

The JSON-RPC specification defines the following error *code* values:

- -32700 - Invalid JSON was received by the server. An error occurred on the server while parsing the JSON text.
- -32600 - The JSON sent is not a valid Request object.
- -32601 - The method does not exist / is not available.
- -32602 - Invalid method parameter(s).
- -32603 - Internal JSON-RPC error.
- -32000 to -32099 - Reserved for application defined errors (see below)

To make server errors easier to read, along the numeric *code*, we use a *type* param that yields a literal error token. For all application defined errors, the *code* is always -32000. It's best to ignore the *code* and just use the *type* param.

```
{"jsonrpc": "2.0",
 "id": 1,
 "method": "login",
 "params":
 {"foo": "joe",
  "bar": "SWkkasE32"}}
```

which results in:

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32602,
  "type": "rpc.method.unexpected_params",
  "message": "Unexpected params",
  "data":
  {"param": "foo"}}}
```

The *message* param is a free text string in English meant for human consumption, which is a one-to-one match with the *type* param. To remove noise from the examples, this param is omitted from the following descriptions.

An additional method-specific *data* param may be added to give further details on the error, most predominantly a *reason* param which is also a free text string in English meant for human consumption. To remove noise from the examples, this param is omitted from the following descriptions. But any additional *data* params will be noted by each method description.

Application defined errors

All methods may return one of the following JSON RPC or application defined errors, in addition to others, specific to each method.

```
{"type": "rpc.request.parse_error"}
{"type": "rpc.request.invalid"}
{"type": "rpc.method.not_found"}
```

```
{"type": "rpc.method.invalid_params", "data": {"param": <string>}}
{"type": "rpc.internal_error"}


{"type": "rpc.request.eof_parse_error"}
{"type": "rpc.request.multipart_broken"}
{"type": "rpc.request.too_big"}
{"type": "rpc.request.method_denied"}


{"type": "rpc.method.unexpected_params", "data": {"param": <string>}}
{"type": "rpc.method.invalid_params_type", "data": {"param": <string>}}
{"type": "rpc.method.missing_params", "data": {"param": <string>}}
{"type": "rpc.method.unknown_params_value", "data": {"param": <string>}}


{"type": "rpc.method.failed"}
{"type": "rpc.method.denied"}
{"type": "rpc.method.timeout"}

{"type": "session.missing_sessionid"}
{"type": "session.invalid_sessionid"}
{"type": "session.overload"}
```

FAQ

What are the security characteristics of the JSON-RPC api?

JSON-RPC runs on top the embedded web server (see "The web server" chapter), which accepts HTTP and/or HTTPS.

The JSON-RPC session ties the client and the server via an HTTP cookie, named "sessionid" which contains a randomly server-generated number. This cookie is not only secure (when the requests come over HTTPS), meaning that HTTPS cookies do not leak over HTTP, but even more importantly this cookie is also http-only, meaning that only the server and the browser (e.g. not the JavaScript code) have access to the cookie. Furthermore, this cookie is a session cookie, meaning that a browser restart would delete the cookie altogether.

The JSON-RPC session lives as long as the user does not request to logout, as long as the user is active within a 30 minute (default value, which is configurable) time frame, as long as there are no severe server crashes. When the session dies, the server will reply with the intention to delete any "sessionid" cookies stored in the browser (to prevent any leaks).

When used in a browser, the JSON-RPC API does not accept cross-domain requests by default, but can be configured to do so via the custom headers functionality in the embedded web server, or by adding a reverse-proxy (see "The web server" chapter).

What is the proper way to use the JSON-RPC api in a cors setup?

The embedded server allows for custom headers to be se, in this case CORS headers, like:

```
Access-Control-Allow-Origin: http://webpage.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Headers: Origin, Content-Type, Accept
Access-Control-Request-Method: POST
```

A server hosted at http://server.com responding with these headers, would mean that the JSON-RPC API can be contacted from a browser which is showing a web page from http://webpage.com, and will allow the browser to make POST requests, with a limited amount of headers and with credentials (i.e. cookies).

This is not enough though, because the browser also needs to be told that your JavaScript code really wants to make a CORS request. A jQuery example would show like this:

```
// with jQuery
$.ajax({
  type: 'post',
  url: 'http://server.com/jsonrpc',
  contentType: 'application/json',
  data: JSON.stringify({
    jsonrpc: '2.0',
    id: 1,
    method: 'login',
    params: {
      'user': 'joe',
      'passwd': 'SWkkasE32'
    }
  }),
  dataType: 'json',
  crossDomain: true,       // CORS specific
  xhrFields: {             // CORS specific
    withCredentials: true  // CORS specific
  }                        // CORS specific
})
```

Without this setup, you will notice that the browser will not send the "sessionid" cookie on post-login JSON-RPC calls.

What is a tag/keypath?

A *tagpath* is a path pointing to a specific position in a YANG module's schema.

A *keypath* is a path pointing to specific position in a YANG module's instance.

These kind of paths are used for several of the API methods (e.g. *set_value*, *get_value*, *subscribe_changes*), and could be seen as XPath path specifications in abbreviated format.

Lets look at some examples using the following YANG module as input:

```
module devices {
    namespace "http://acme.com/ns/devices";
    prefix d;

    container config {
        leaf description { type string; }
        list device {
            key "interface";
            leaf interface { type string; }
            leaf date { type string; }
        }
    }
}
```

Valid tagpaths:

- `/d:config/description`
- `/d:config/device/interface`

Valid keypaths:

- `/d:config/device{eth0}/date` - the date leaf value within a device with an *interface* key set to *eth0*

Note how the prefix is prepended to the first tag in the path. This prefix is compulsory.

Restricting access to methods

The AAA infrastructure can be used to restrict access to library functions using command rules:

```
<cmdrule xmlns="http://tail-f.com/yang/acm">
  <name>webui</name>
  <context xmlns="http://tail-f.com/yang/acm">webui</context>
  <command>::jsonrpc:: get_schema</command>
  <access-operations>read exec</access-operations>
  <action>deny</action>
</cmdrule>
```

Note how the command is prefixed with "::jsonrpc:: ". This tells the AAA engine to apply the command rule to JSON-RPC API functions.

You can read more about command rules in "The AAA infrastructure" chapter in this User Guide.

What is session.overload error?

A series of limits are imposed on the load that one session can put on the system.

This reduces the risk that a session takes overs the whole system and brings it into a DoS situation.

The response will include details about the limit that triggered the error.

Known limits:

- only 10000 commands/subscriptions are allowed per session

Methods - commands

Method get_cmds

Get a list of the session's batch commands

Params

```
{}
```

Result

```
{"cmds": <array of cmd>}

cmd =
 {"params": <object>,
  "comet_id": <string>,
  "handle": <string>,
  "tag": <"string">,
  "started": <boolean>,
  "stopped": <boolean; should be always false>}
```

Method init_cmd

Starts a batch command

NOTE: the *start_cmd* method must be called to actually get the batch command to generate any messages, unless the *handle* is provided as input.

NOTE: As soon as the batch command prints anything on stdout it will be sent as a message and turn up as a result to your polling call to the *comet* method.

Params

```
{"th": <integer>,
 "name": <string>,
 "args": <string>,
 "emulate": <boolean, default: false>,
 "width": <integer, default: 80>,
 "height": <integer, default: 24>,
 "scroll": <integer, default: 0>,
 "comet_id": <string>,
 "handle": <string, optional>}
```

The *name* param is one on the named commands defined in ncs.conf.

The *args* param any extra arguments to be provided to the command expect for the ones specified in ncs.conf.

The *emulate* param specifies if terminal emulation should be enabled.

The *width*, *height*, *scroll* properties define the screen properties.

Result

```
{"handle": <string>}
```

A handle to the batch command is returned (equal to *handle* if provided).

Method send_cmd_data

Sends data to batch command started with *init_cmd*

Params

```
{"handle": <string>,
 "data": <string>}
```

The *handle* param is as returned from a call to *init_cmd* and the *data* param is what is to be sent to the batch command started with *init_cmd*.

Result

```
{}
```

Errors (specific)

```
{"type": "cmd.not_initialized"}
```

Method start_cmd

Signals that a batch command can start to generate output.

NOTE: This method must be called to actually start the activity initiated by calls to one of the methods *init_cmd*.

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Method suspend_cmd

Suspends output from a batch command

NOTE: the *init_cmd* method must have been called with the *emulate* param set to true for this to work

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Method resume_cmd

Resumes a batch command started with *init_cmd*

NOTE: the *init_cmd* method must have been called with the *emulate* param set to true for this to work

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Method stop_cmd

Stops a batch command

NOTE: This method must be called to stop the activity started by calls to one of the methods *init_cmd*.

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *init_cmd*.

Result

```
{}
```

Methods - commands - subscribe

Method get_subscriptions

Get a list of the session's subscriptions

Params

```
{}
```

Result

```
{"subscriptions": <array of subscription>}

subscription =
 {"params": <object>,
  "comet_id": <string>,
  "handle": <string>,
  "tag": <"string">,
  "started": <boolean>,
  "stopped": <boolean; should be always false>}
```

Method subscribe_cdboper

Starts a subscriber to operational data in CDB. Changes done to configuration data will not be seen here.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>,
 "path": <string>}
```

The *path* param is a keypath restricting the subscription messages to only be about changes done under that specific keypath.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an array of changes of the same type as returned by the *subscribe_changes* method. See below.

Errors (specific)

```
{"type": "db.cdb_operational_not_enabled"}
```

Method subscribe_changes

Starts a subscriber to configuration data in CDB. Changes done to operational data in CDB data will not be seen here. Furthermore, subscription messages will only be generated when a transaction is successfully committed.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>,
 "path": <string>,
 "skip_local_changes": <boolean, default: false>,
 "hide_changes": <boolean, default: false>,
 "hide_values": <boolean, default: false>}
```

The *path* param is a keypath restricting the subscription messages to only be about changes done under that specific keypath.

The *skip_local_changes* param specifies if configuration changes done by the owner of the read-write transaction should generate subscription messages.

The *hide_changes* and *hide_values* params specify a lower level of information in subscription messages, in case it is enough to receive just a "ping" or a list of changed keypaths, respectively, but not the new values resulted in the changes.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an object such as:

```
{"db": <"running" | "startup" | "candidate">,
 "user": <string>,
 "ip": <string>,
 "changes": <array>}
```

The *user* and *ip* properties are the username and ip-address of the committing user.

The *changes* param is an array of changes of the same type as returned by the *changes* method. See above.

Method subscribe_poll_leaf

Starts a polling subscriber to any type of operational and configuration data (outside of CDB as well).

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"th": <integer>,
 "path": <string>,
 "interval": <integer between 0 and 3600>,
 "comet_id": <string>,
 "handle": <string, optional>}
```

The *path* param is a keypath pointing to a leaf value.

The *interval* is a timeout in seconds between when to poll the value.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of is a simple string value.

Method subscribe_upgrade

Starts a subscriber to upgrade messages.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>}
```

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an object such as:

```
{"upgrade_state": <"wait_for_init" | "init" | "abort" | "commit">,
 "timeout": <number, only if "upgrade_state" === "wait_for_init">}
```

Method subscribe_jsonrpc_batch

Starts a subscriber to JSONRPC messages for batch requests.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>}
```

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method having exact same structure like a JSONRPC response:

```
{"jsonrpc":"2.0",
 "result":"admin",
 "id":1}
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32602,
  "type": "rpc.method.unexpected_params",
  "message": "Unexpected params",
  "data":
  {"param": "foo"}}}
```

Method subscribe_progress_trace

Starts a subscriber to progress trace events.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>,
 "verbosity": <"normal" | "verbose" | "very_verbose" | "debug", default: "normal">
 "filter_context": <"webui" | "cli" | "netconf" | "rest" | "snmp" | "system" | string, optional>
```

The *verbosity* param specifies the verbosity of the progress trace.

The *filter_context* param can be used to only get progress events from a specific context For example, if *filter_context* is set to *cli* only progress trace events from the CLI are returned.

Result

```
{"handle": <string>}
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of that message will be an object such as:

```
{"timestamp": <string>,
 "duration": <string, optional if end of span>,
 "span-id": <string>,
 "parent-span-id": <string, optional if parent span exists>,
 "trace-id": <string>,
 "session-id": <integer>,
 "transaction-id": <integer, optional if transaction exists>,
 "datastore": <string, optional if transaction exists>,
 "context": <string>,
 "subsystem": <string, optional if in subsystem>,
 "message": <string>,
 "annotation": <string, optional if end of span>,
 "attributes":  <object with key-value attributes, optional>,
 "links":  <array with objects, each containing a trace-id and span-id
 key, optional>}
```

Method start_subscription

Signals that a subscribe command can start to generate output.

NOTE: This method must be called to actually start the activity initiated by calls to one of the methods *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade* **with no *handle*

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade*.

Result

```
{}
```

Method unsubscribe

Stops a subscriber

NOTE: This method must be called to stop the activity started by calls to one of the methods *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade*.

Params

```
{"handle": <string>}
```

The *handle* param is as returned from a call to *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade*.

Result

```
{}
```

Methods - data

Method create

Create a list entry, a presence container, or a leaf of type empty

Params

```
{"th": <integer>,
 "path": <string>}
```

The *path* param is a keypath pointing to data to be created.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked"}
```

Method delete

Deletes an existing list entry, a presence container, or an optional leaf and all its children (if any).

Note If the permission to delete is denied on a child, the 'warnings' array in the result will contain a warning 'Some elements could not be removed due to NACM rules prohibiting access.'. The delete method will still delete as much as is allowed by the rules. See "The AAA infrastructure" chapter in this User Guide for more information about permissions and authorization.

Params

```
{"th": <integer>,
 "path": <string>}
```

The *path* param is a keypath pointing to data to be deleted.

Result

```
{} |
                {"warnings": <array of strings>}
```

Errors (specific)

```
{"type": "db.locked"}
```

Method exists

Checks if optional data exists

Params

```
{"th": <integer>,
 "path": <string>}
```

The *path* param is a keypath pointing to data to be checked for existence.

Result

```
{"exists": <boolean>}
```

Method get_case

Get the case of a choice leaf

Params

```
{"th": <integer>,
 "path": <string>,
 "choice": <string>}
```

The *path* param is a keypath pointing to data that contains the choice leaf given by the *choice* param.

Result

```
{"case": <string>}
```

Method show_config

Retrieves configuration and operational data from the provided transaction

Params

```
{"th": <integer>,
 "path": <string>
 "result_as": <"string" | "json" | "json2", default: "string">
 "with_oper": <boolean, default: false>
 "max_size": <"integer", default: 0>}
```

The *path* param is a keypath to the configuration to be returned. *result_as* controls the output format, string for a compact string format, json for json compatible with restconf and json2 for a variant of the restconf json format. *max_size* sets the maximum size of the data field in kb, set to 0 to disable the limit.

Result

result_as string

```
{"config": <string>}
```

result_as json

```
{"data": <json>}
```

Method load

Load XML configuration into current transaction

Params

```
{"th": <integer>,
 "data": <string>
 "path": <string, default: "/">
 "format": <"json" | "xml", default: "xml">
 "mode": <"create" | "merge" | "replace", default: "merge">}
```

The *data* param is the data to be loaded into the transaction. *mode* controls how the data is loaded into the transaction, analogous with the CLI command load.*format* informs load about which format *data* is in. If *format* is xml the data must be an XML document encoded as a string. If *format* is json data can either be a JSON document encoded as a string or the JSON data itself.

Result

```
{}
```

Errors (specific)

```
{"row": <integer>, "message": <string>}
```

Methods - data - attrs

Method get_attrs

Get node attributes

Params

```
{"th": <integer>,
 "path": <string>,
 "names": <array of string>}
```

The *path* param is a keypath pointing to the node and the *names* param is a list of attribute names that you want to retrieve.

Result

```
{"attrs": <object of attribute name/value>}
```

Method set_attrs

Set node attributes

Params

```
{"th": <integer>,
 "path": <string>,
 "attrs": <object of attribute name/value>}
```

The *path* param is a keypath pointing to the node and the *attrs* param is an object that maps attribute names to their values.

Result

```
{}
```

Methods - data - leafs

Method get_value

Gets a leaf value

Params

```
{"th": <integer>,
 "path": <string>,
 "check_default": <boolean, default: false>}
```

The *path* param is a keypath pointing to a value.

The *check_default* param adds *is_default* to the result if set to true. *is_default* is set to true if the default value handling returned the value.

Result

```
{"value": <string>}
```

Example

Example 1. Method get_value

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "get_value",
         "params": {"th": 4711,
                    "path": "/dhcp:dhcp/max-lease-time"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {"value": "7200"}
}
```

Method get_values

Get leaf values

Params

```
{"th": <integer>,
 "path": <string>,
 "check_default": <boolean, default: false>,
 "leafs": <array of string>}
```

The *path* param is a keypath pointing to a container. The *leafs* param is an array of children names residing under the parent container in the YANG module.

The *check_default* param adds *is_default* to the result if set to true. *is_default* is set to true if the default value handling returned the value.

Result

```
{"values": <array of value/error>}

value  = {"value": <string>, "access": <access>}
error  = {"error": <string>, "access": <access>} |
         {"exists": true, "access": <access>} |
         {"not_found": true, "access": <access>}
access = {"read": true, write: true}
```

NOTE: The access object has no "read" and/or "write" properties if there are no read and/or access rights.

Method set_value

Sets a leaf value

Params

```
{"th": <integer>,
 "path": <string>,
 "value": <string | boolean | integer | array | null>,
 "dryrun": <boolean, default: false}
```

The *path* param is the keypath to give a new value as specified with the *value* param.

value can be an array when the *path* is a leaf-list node.

When *value* is *null*, the *set_value* method acts like *delete*.

When *dryrun* is *true*, this function can be used to test if a value is valid or not.

Note If this method is used for deletion and permission to delete is denied on a child, the 'warnings' array in the result will contain a warning "Some elements could not be removed due to NACM rules prohibiting access.'. The delete will still delete as much as is allowed by the rules. See "The AAA infrastructure" chapter in this User Guide for more information about permissions and authorization.

Result

```
{} |
                {"warnings": <array of strings>}
```

Errors (specific)

```
{"type": "data.already_exists"}
{"type": "data.not_found"}
{"type": "data.not_writable"}
{"type": "db.locked"}
```

Example

Example 2. Method set_value

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "set_value",
         "params": {"th": 4711,
                    "path": "/dhcp:dhcp/max-lease-time",
                    "value": "4500"}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result": {}
}
```

Methods - data - leafref

Method deref

Dereferences a leaf with a leafref type

Params

```
{"th": <integer>,
 "path": <string>,
 "result_as": <"paths" | "target" | "list-target", default: "paths">}
```

The *path* param is a keypath pointing to a leaf with a leafref type.

Result

```
{"paths": <array of string, a keypath to a leaf>}
```

```
{"target": <a keypath to a leaf>}
```

```
{"list-target": <a keypath to a list>}
```

Method get_leafref_values

Gets all possible values for a leaf with a leafref type

Params

```
{"th": <integer>,
 "path": <string>,
 "offset": <integer, default: 0>,
 "limit": <integer, default: -1>,
 "starts_with": <string, optional>,
 "skip_grouping": <boolean, default: false>,
 "keys": <object>}
```

The *th* param is as returned from a call to *new_read_trans* or *new_write_trans*. The *path* param is a keypath pointing to a leaf with a leafref type. *Note*: If the leafref is within an action or rpc, *th* should be created with an *action_path*.

The *offset* param is used to skip as many values as it is set to. E.g. an *offset* of 2 will skip the first 2 values. If not given the value defaults to '0', which means no values are skipped. Offset needs to be a non-negative integer or an "invalid params" error will be returned. An offset that is bigger than the length of the leafref list will result in an "method failed" error being returned.

NOTE: *offset* used together with *limit* (see below) can be used repeatedly to paginate the leafref values.

The *limit* param can be set to limit the number of returned values. E.g. a limit of 5 will return a list with 5 values. If not given the value defaults to '-1', which means no limit. Limit needs to be -1 or a non-negative integer or an "invalid params" error will be returned. A Limit of 0 will result in an empty list being returned

The *starts_with* param can be used to filter values by prefix.

The *skip_grouping* param is by default set to false and is only needed to be set to true if if a set of sibling leafref leafs points to a list instance with multiple keys *and* if *get_leafref_values* should return an array of possible leaf values instead an array of arrays with possible key value combinations.

The *keys* param is an optional array of values that should be set if a more than one leafref statement is used within action/rpc input parameters *and* if they refer to each other using `deref()` or `current()` XPath functions. For example consider this model:

```
rpc create-service {
  tailf:exec "./run.sh";
  input {
    leaf name {
      type leafref {
        path "/myservices/service/name";
      }
    }
    leaf if {
      type leafref {
        path "/myservices/service[name=current()/../name]/interfaces/name"
      }
    }
  }
  output {
    leaf result { type string; }
  }
}
```

The leaf *if* refers to leaf *name* in its XPath expression so to be able to successfully run *get_leafref_values* on that node you need to provide a valid value for the *name* leaf using the *keys* parameter. The *keys* parameter could for example look like this:

```
{"/create-service/name": "service1"}
```

Result

```
{"values": <array of string>,
 "source": <string> | false}
```

The *source* param will point to the keypath where the values originate. If the keypath cannot be resolved due to missing/faulty items in the *keys* parameter *source* will be *false*.

Methods - data - lists

Method rename_list_entry

Renames a list entry.

Params

```
{"th": <integer>,
 "from_path": <string>,
 "to_keys": <array of string>}
```

The *from_path* is a keypath pointing out the list entry to be renamed.

The list entry to be renamed will, under the hood, be deleted all together and then recreated with the content from the deleted list entry copied in.

The *to_keys* param is an array with the new key values. The array must contain a full set of key values.

Result

```
{}
```

Errors (specific)

```
{"type": "data.already_exists"}
{"type": "data.not_found"}
{"type": "data.not_writable"}
```

Method copy_list_entry

Copies a list entry.

Params

```
{"th": <integer>,
 "from_path": <string>,
 "to_keys": <array of string>}
```

The *from_path* is a keypath pointing out the list entry to be copied.

The *to_keys* param is an array with the new key values. The array must contain a full set of key values.

Copying between different ned-id versions works as long as the schema nodes being copied has not changed between the versions.

Result

```
{}
```

Errors (specific)

```
{"type": "data.already_exists"}
{"type": "data.not_found"}
{"type": "data.not_writable"}
```

Method move_list_entry

Moves an ordered-by user list entry relative to its siblings.

Params

```
{"th": <integer>,
 "from_path": <string>,
 "to_path": <string>,
 "mode": <"first" | "last" | "before" | "after">}
```

The *from_path* is a keypath pointing out the list entry to be moved.

The list entry to be moved can either be moved to the first or the last position, i.e. if the *mode* param is set to *first* or *last* the *to_path* keypath param has no meaning.

If the *mode* param is set to *before* or *after* the *to_path* param must be specified, i.e. the list entry will be moved to the position before or after the list entry which the *to_path* keypath param points to.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked"}
```

Method append_list_entry

Append a list entry to a leaf-list.

Params

```
{"th": <integer>,
 "path": <string>,
 "value": <string>}
```

The *path* is a keypath pointing to a leaf-list.

Result

```
{}
```

Method count_list_keys

Counts the number of keys in a list.

Params

```
{"th": <integer>
 "path": <string>}
```

The *path* parameter is a keypath pointing to a list.

Result

```
{"count": <integer>}
```

Method get_list_keys

Enumerates keys in a list.

Params

```
{"th": <integer>,
 "path": <string>,
 "chunk_size": <integer greater than zero, optional>,
 "start_with": <array of string, optional>,
 "lh": <integer, optional>}
```

The *th* parameter is the transaction handle.

The *path* parameter is a keypath pointing to a list. Required on first invocation - optional in following.

The *chunk_size* parameter is the number of requested keys in the result. Optional - default is unlimited.

The *start_with* parameter will be used to filter out all those keys that do not start with the provided strings. The parameter supports multiple keys e.g. if the list has two keys, then *start_with* can hold two items.

The *lh* (list handle) parameter is optional (on the first invocation) but must be used in following invocations.

Result

```
{"keys": <array of array of string>,
 "total_count": <integer>,
 "lh": <integer, optional>}
```

Each invocation of *get_list_keys* will return at most *chunk_size* keys. The returned *lh* must be used in following invocations to retrieve next chunk of keys. When no more keys are available the returned *lh* will be set to `-1`.

On the first invocation *lh* can either be omitted or set to `-1`.

Methods - data - query

Method query

Starts a new query attached to a transaction handle, retrieves the results, and stops the query immediately. This is a convenience method for calling *start_query*, *run_query* and *stop_query* in a one-time sequence.

This method should not be used for paginated results, as it results in performance degradation - use *start_query*, multiple *run_query*and *stop_query*instead.

Example

Example 3. Method query

```
curl \
    --cookie "sessionid=sess11635875109111642;" \
    -X POST \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "query",
         "params": {"th": 1,
                    "xpath_expr": "/dhcp:dhcp/dhcp:foo",
                    "result_as": "keypath-value"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"current_position": 2,
  "total_number_of_results": 4,
  "number_of_results": 2,
  "number_of_elements_per_result": 2,
  "results": ["foo", "bar"]}}
```

Method start_query

Starts a new query attached to a transaction handle. On success a query handle is returned to be in subsequent calls to *run_query*.

Params

```
{"th": <integer>,
 "xpath_expr": <string, optional if path is given>,
 "path": <string, keypath, optional if xpath_expr is given>,
 "selection": <array of xpath expressions, optional>
 "chunk_size": <integer greater than zero, optional>
 "initial_offset": <integer, optional>,
 "sort", <array of xpath expressions, optional>,
 "sort_order": <"ascending" | "descending", optional>,
 "include_total": <boolean, default: true>,
 "context_node": <string, keypath, optional>,
 "result_as": <"string" | "keypath-value" | "leaf_value_as_string", default: "string">}
```

The *xpath_expr* param is the primary XPath expression to base the query on. Alternatively, one can give a keypath as the *path* param, and internally the keypath will be translated into an XPath expression.

A query is a way of evaluating an XPath expression and returning the results in chunks. The primary XPath expression must evaluate to a node-set, i.e. the result. For each node in the result a *selection* Xpath expression is evaluated with the result node as its context node.

Note: The terminology used here is as defined in http://en.wikipedia.org/wiki/XPath.

For example, given this YANG snippet:

```
list interface {
  key name;
  unique number;
  leaf name {
    type string;
  }
  leaf number {
    type uint32;
    mandatory true;
  }
  leaf enabled {
    type boolean;
    default true;
  }
}
```

The *xpath_expr* could be `/interface[enabled='true']` and *selection* could be `{ "name", "number" }`.

Note that the *selection* expressions must be valid XPath expressions, e.g. to figure out the name of an interface and whether its number is even or not, the expressions must look like: `{ "name", "(number mod 2) == 0" }`.

The result are then fetched using *run_query*, which returns the result on the format specified by *result_as* param.

There are two different types of result:

- *string* result is just an array with resulting strings of evaluating the *selection* XPath expressions
- `keypath-value` result is an array the keypaths or values of the node that the *selection* XPath expression evaluates to.

This means that care must be taken so that the combination of *selection* expressions and return types actually yield sensible results (for example `1 + 2` is a valid *selection* XPath expression, and would result in the string *3* when setting the result type to *string* - but it is not a node, and thus have no keypath-value.

It is possible to sort the result using the built-in XPath function `sort-by()` but it is also also possible to sort the result using expressions specified by the *sort* param. These expressions will be used to construct a temporary index which will live as long as the query is active. For example to start a query sorting first on the enabled leaf, and then on number one would call:

```
$.post("/jsonrpc", {
  jsonrpc: "2.0",
  id: 1,
  method: "start_query",
  params:  {
    th: 1,
    xpath_expr: "/interface[enabled='true']",
    selection: ["name", "number", "enabled"],
    sort: ["enabled", "number"]
  }
})
    .done(...);
```

The *context_node* param is a keypath pointing out the node to apply the query on; only taken into account when the *xpath_expr* uses relatives paths. Lack of a *context_node*, turns relatives paths into absolute paths.

The *chunk_size* param specifies how many result entries to return at a time. If set to 0 a default number will be used.

The *initial_offset* param is the result entry to begin with (1 means to start from the beginning).

Result

```
{"qh": <integer>}
```

A new query handler handler id to be used when calling *run_query* etc

Example

Example 4. Method start_query

```
curl \
    --cookie "sessionid=sess11635875109111642;" \
    -X POST \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "start_query",
         "params": {"th": 1,
                    "xpath_expr": "/dhcp:dhcp/dhcp:foo",
                    "result_as": "keypath-value"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": 47}
```

Method run_query

Retrieves the result to a query (as chunks). For more details on queries please read the description of "start_query".

Params

```
{"qh": <integer>}
```

The *qh* param is as returned from a call to "start_query".

Result

```
{"position": <integer>,
 "total_number_of_results": <integer>,
 "number_of_results": <integer>,
 "chunk_size": <integer greater than zero, optional>,
 "result_as": <"string" | "keypath-value" | "leaf_value_as_string">,
 "results": <array of result>}

result = <string> |
         {"keypath": <string>, "value": <string>}
```

The *position* param is the number of the first result entry in this chunk, i.e. for the first chunk it will be 1.

How many result entries there are in this chunk is indicated by the *number_of_results* param. It will be 0 for the last chunk.

The *chunk_size* and the *result_as* properties are as given in the call to *start_query*.

The *total_number_of_results* param is total number of result entries retrieved so far.

The *result* param is as described in the description of *start_query*.

Example

Example 5. Method run_query

```
curl \
    --cookie "sessionid=sess11635875109111642;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "run_query",
         "params": {"qh": 22}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"current_position": 2,
  "total_number_of_results": 4,
  "number_of_results": 2,
  "number_of_elements_per_result": 2,
  "results": ["foo", "bar"]}}
```

Method reset_query

Reset/rewind a running query so that it starts from the beginning again. Next call to "run_query" will then return the first chunk of result entries.

Params

```
{"qh": <integer>}
```

The *qh* param is as returned from a call to *start_query*.

Result

```
{}
```

Example

Example 6. Method reset_query

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "reset_query",
         "params": {"qh": 67}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": true}
```

Method stop_query

Stops the running query identified by query handler. If a query is not explicitly closed using this call it will be cleaned up when the transaction the query is linked to ends.

Params

```
{"qh": <integer>}
```

The *qh* param is as returned from a call to "start_query".

Result

```
{}
```

Example

Example 7. Method stop_query

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "stop_query",
         "params": {"qh": 67}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": true}
```

Methods - database

Method reset_candidate_db

Resets the candidate datastore

Result

```
{}
```

Method lock_db

Takes a database lock

Params

```
{"db": <"startup" | "running" | "candidate">}
```

The *db* param specifies which datastore to lock.

Result

```
{}
```

Errors (specific)

```
{"type": "db.locked", "data": {"sessions": <array of string>}}
```

The `data.sessions` param is an array of strings describing the current sessions of the locking user, e.g. an array of "admin tcp (cli from 192.245.2.3) on since 2006-12-20 14:50:30 exclusive".

Method unlock_db

Releases a database lock

Params

```
{"db": <"startup" | "running" | "candidate">}
```

The *db* param specifies which datastore to unlock.

Result

```
{}
```

Method copy_running_to_startup_db

Copies the running datastore to the startup datastore

Result

```
{}
```

Methods - general

Method comet

Listens on a comet channel, i.e. all asynchronous messages from batch commands started by calls to *start_cmd*, *subscribe_cdboper*, *subscribe_changes*, *subscribe_messages*, *subscribe_poll_leaf* or *subscribe_upgrade* ends up on the comet channel.

You are expected to have a continuous long polling call to the *comet* method at any given time. As soon as the browser or server closes the socket, due to browser or server connect timeout, the *comet* method should be called again.

As soon as the *comet* method returns with values they should be dispatched and the *comet* method should be called again.

Params

```
{"comet_id": <string>}
```

Result

```
[{"handle": <integer>,
  "message": <a context specific json object, see example below>},
 ...]
```

Errors (specific)

```
{"type": "comet.duplicated_channel"}
```

Example

Example 8. Method comet

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "subscribe_changes",
         "params": {"comet_id": "main",
                    "path": "/dhcp:dhcp"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {"handle": "2"}}
```

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "batch_init_done",
         "params": {"handle": "2"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

```
curl \
    -m 15 \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "comet",
         "params": {"comet_id": "main"}}' \
    http://127.0.0.1:8008/jsonrpc
```

hangs... and finally...

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 [{"handle": "1",
   "message":
   {"db": "running",
    "changes":
    [{"keypath": "/dhcp:dhcp/default-lease-time",
      "op": "value_set",
      "value": "100"}],
    "user": "admin",
    "ip": "127.0.0.1"}}]}
```

In this case the admin user seems to have set `/dhcp:dhcp/default-lease-time` to *100*.

Method get_system_setting

Extracts system settings such as capabilities, supported datastores, etc.

Params

```
{"operation": <"capabilities" | "customizations" | "models" | "user" | "version" | "all" | "name
```

The *operation* param specifies which system setting to get:

- *capabilities* - the server-side settings are returned, e.g. is rollback and confirmed commit supported
- *customizations* - an array of all webui customizations
- *models* - an array of all loaded YANG modules are returned, i.e. prefix, namespace, name
- *user* - the username of the currently logged in user is returned
- *version* - the system version
- *all* - all of the above is returned.
- (DEPRECATED) *namespaces* - an object of all loaded YANG modules are returned, i.e. prefix to namespace

Result

```
{"user:" <string>,
 "models:" <array of YANG modules>,
 "version:" <string>,
 "customizations": <array of customizations>,
 "capabilities":
 {"rollback": <boolean>,
  "copy_running_to_startup": <boolean>,
  "exclusive": <boolean>,
  "confirmed_commit": <boolean>
 },
 "namespaces": <object of YANG modules prefix/namespace>}
```

The above is the result if using the *all* operation.

Method abort

Abort a JSON-RPC method by its associated id.

Params

```
{"id": <integer>}
```

The *id* param is the id of the JSON-RPC method to be aborted.

Result

```
{}
```

Method eval_XPath

Evaluates an xpath expression on the server side

Params

```
{"th": <integer>,
 "xpath_expr": <string>}
```

The *xpath_expr* param is the XPath expression to be evaluated.

Result

```
{"value": <string>}
```

Methods - messages

Method send_message

Sends a message to another user in the CLI or Web UI

Params

```
{"to": <string>,
 "message": <string>}
```

The *to* param is the user name of the user to send the message to and the *message* param is the actual message.

NOTE: The username "all" will broadcast the message to all users.

Result

```
{}
```

Method subscribe_messages

Starts a subscriber to messages.

NOTE: the *start_subscription* method must be called to actually get the subscription to generate any messages, unless the *handle* is provided as input.

NOTE: the *unsubscribe* method should be used to end the subscription.

NOTE: As soon as a subscription message is generated it will be sent as a message and turn up as result to your polling call to the *comet* method.

Params

```
{"comet_id": <string>,
 "handle": <string, optional>}
```

Result

```
<string>
```

A handle to the subscription is returned (equal to *handle* if provided).

Subscription messages will end up in the *comet* method and the format of these messages depend on what has happened.

When a new user has logged in:

```
{"new_user": <integer, a session id to be used by "kick_user">
 "me": <boolean, is it myself?>
 "user": <string>,
 "proto": <"ssh" | "tcp" | "console" | "http" | "https" | "system">,
 "ctx": <"cli" | "webui" | "netconf">
 "ip": <string, user's ip-address>,
 "login": <string, login timestamp>}
```

When a user logs out:

```
{"del_user": <integer, a session id>,
 "user": <string>}
```

When receiving a message:

```
{"sender": <string>,
 "message": <string>}
```

Methods - rollbacks

Method get_rollbacks

Lists all available rollback files

Result

```
{"rollbacks": <array of rollback>}

rollback =
 {"nr": <integer>,
  "creator": <string>,
  "date": <string>,
  "via": <"system" | "cli" | "webui" | "netconf">,
  "comment": <string>,
  "label": <string>}
```

The *nr* param is a rollback number to be used in calls to *load_rollback* etc.

The *creator* and *date* properties identify the name of the user responsible for committing the configuration stored in the rollback file and when it happened.

The *via* param identifies the interface that was used to create the rollback file.

The *label* and *comment* properties is as given calling the methods *set_comment* and *set_label* on the transaction.

Method get_rollback

Gets the content of a specific rollback file. The rollback format is as defined in a curly bracket format as defined in the CLI.

Params

```
{"nr": <integer>}
```

Result

```
<string, rollback file in curly bracket format>
```

Method install_rollback

Installs a specific rollback file into a new transaction and commits it. The configuration is restored to the one stored in the rollback file and no further operations are needed. It is the equivalent of creating a new private write private transaction handler with *new_write_trans*, followed by calls to the methods *load_rollback*, *validate_commit* and *commit*.

Note If the permission to rollback is denied on some nodes, the 'warnings' array in the result will contain a warning 'Some changes could not be applied due to NACM rules prohibiting access.'. The *install_rollback* will still rollback as much as is allowed by the rules. See "The AAA infrastructure" chapter in this User Guide for more information about permissions and authorization.

Params

```
{"nr": <integer>}
```

Result

```
{}
```

Method load_rollback

Rolls back within an existing transaction, starting with the latest rollback file, down to a specified rollback file, or selecting only the specified rollback file (also known as "selective rollback").

Note If the permission to rollback is denied on some nodes, the 'warnings' array in the result will contain a warning 'Some changes could not be applied due to NACM rules prohibiting access.'. The *load_rollback* will still rollback as much as is allowed by the rules. See "The AAA infrastructure" chapter in this User Guide for more information about permissions and authorization.

Params

```
{"th": <integer>,
 "nr": <integer>,
 "path": <string>,
 "selective": <boolean, default: false>}
```

The *nr* param is a rollback number returned by *get_rollbacks*.

The *path* param is a keypath that restrict the rollback to be applied only to a subtree.

The *selective* param, false by default, can restrict the rollback process to use only the rollback specified by *nr*, rather than applying all known rollbacks files starting with the latest down to the one specified by *nr*.

Result

```
{}
```

Methods - schema

Method get_description

Get description. To be able to get the description in the response the fxs file need to be compiled with the flag "--include-doc". This operation can be heavy so instead of calling get_description directly, we can confirm that there is a description before calling in "CS_HAS_DESCR" flag that we get from "get_schema" response.

Params

```
{"th": <integer>,
 "path": <string, optional>
```

A *path* is a tagpath/keypath pointing into a specific sub-tree of a YANG module.

Result

```
{"description": <string>}
```

Method get_schema

Exports a JSON schema for a selected part (or all) of a specific YANG module (with optional instance data inserted)

Params

```
{"th": <integer>,
 "namespace": <string, optional>,
 "path": <string, optional>,
 "levels": <integer, default: -1>,
 "insert_values": <boolean, default: false>,
 "evaluate_when_entries": <boolean, default: false>,
 "stop_on_list": <boolean, default: false>,
 "cdm_namespace": <boolean, default: false>}
```

One of the properties *namespace* or *path* must be specified.

A *namespace* is as specified in a YANG module.

A *path* is a tagpath/keypath pointing into a specific sub-tree of a YANG module.

The *levels* param limits the maximum depth of containers and lists from which a JSON schema should be produced (-1 means unlimited depth).

The *insert_values* param signals that instance data for leafs should be inserted into the schema. This way the need for explicit forthcoming calls to *get_elem* are avoided.

The *evaluate_when_entries* param signals that schema entries should be included in the schema even though their "when" or "tailf:display-when" statements evaluate to false, i.e. instead a boolean *evaluated_when_entry* param is added to these schema entries.

The *stop_on_list* param limits the schema generation to one level under the list when true.

The *cdm_namespace* param signals inclusion of cdm-namespace entries where appropriate.

Result

```
{"meta":
 {"namespace": <string, optional>,
  "keypath": <string, optional>,
  "prefix": <string>,
  "types": <array of type>},
 "data": <array of child>}

type = <array of {<string, type name with prefix>: <type_stack>}>

type_stack = <array of type_stack_entry>

type_stack_entry =
 {"bits": <array of string>, "size": <32 | 64>} |
 {"leaf_type": <type_stack>, "list_type": <type_stack>} |
 {"union": <array of type_stack>} |
 {"name": <primitive_type | "user_defined">,
  "info": <string, optional>,
  "readonly": <boolean, optional>,
  "facets": <array of facet, only if not primitive type>}

primitive_type =
 "empty" |
 "binary" |
 "bits" |
 "date-and-time" |
 "instance-identifier" |
 "int64" |
 "int32" |
 "int16" |
 "uint64" |
 "uint32" |
 "uint16" |
 "uint8" |
 "ip-prefix" |
 "ipv4-prefix" |
 "ipv6-prefix" |
 "ip-address-and-prefix-length" |
 "ipv4-address-and-prefix-length" |
 "ipv6-address-and-prefix-length" |
 "hex-string" |
 "dotted-quad" |
 "ip-address" |
 "ipv4-address" |
 "ipv6-address" |
 "gauge32" |
 "counter32" |
 "counter64" |
 "object-identifier"

facet_entry =
 {"enumeration": {"label": <string>, "info": <string, optional>}} |
 {"fraction-digits": {"value": <integer>}} |
 {"length": {"value": <integer>}} |
 {"max-length": {"value": <integer>}} |
 {"min-length": {"value": <integer>}} |
 {"leaf-list": <boolean>} |
 {"max-inclusive": {"value": <integer>}} |
 {"max-length": {"value": <integer>}} |
 {"range": {"value": <array of range_entry>}} |
 {"min-exclusive": {"value": <integer>}} |
 {"min-inclusive": {"value": <integer>}} |
 {"min-length": {"value": <integer>}} |
 {"pattern": {"value": <string, regular expression>}} |
 {"total-digits": {"value": <integer>}}

range_entry =
 "min" |
 "max" |
 <integer> |
 [<integer, min value>, <integer, max value>]

child =
 {"kind": <kind>,
  "name": <string>,
  "qname": <string, same as "name" but with prefix prepended>,
  "info": <string>,
  "namespace": <string>,
  "xml-namespace": <string>,
  "is_action_input": <boolean>,
  "is_action_output": <boolean>,
  "is_cli_preformatted": <boolean>,
  "is_mount_point": <boolean>
  "presence": <boolean>,
  "ordered_by": <boolean>,
  "is_config_false_callpoint": <boolean>,
  "key": <boolean>,
  "exists": <boolean>,
  "value": <string | number | boolean>,
  "is_leafref": <boolean>,
  "leafref_target": <string>,
  "when_targets": <array of string>,
  "deps": <array of string>
  "hidden": <boolean>,
  "default_ref":
  {"namespace": <string>,
   "tagpath": <string>
  },
  "access":
  {"create": <boolean>,
   "update": <boolean>,
   "delete": <boolean>,
   "execute": <boolean>
  },
  "config": <boolean>,
  "readonly": <boolean>,
  "suppress_echo": <boolean>,
  "type":
  {"name": <primitive_type>,
   "primitive": <boolean>
  }
  "generated_name": <string>,
  "units": <string>,
  "leafref_groups": <array of string>,
  "active": <string, active case, only if "kind" is "choice">,
  "cases": <array of case, only of "kind" is "choice">,
  "default": <string | number | boolean>,
  "mandatory": <boolean>,
  "children": <children>
 }

kind =
 "module" |
 "access-denies" |
 "list-entry" |
 "choice" |
 "key" |
 "leaf-list" |
 "action" |
 "container" |
 "leaf" |
 "list" |
 "notification"

case_entry =
 {"kind": "case",
  "name": <string>,
  "children": <array of child>
 }
```

This is a fairly complex piece of JSON but it essentially maps what is seen in a YANG module. Keep that in mind when scrutinizing the above.

The *meta* param contains meta-information about the YANG module such as namespace and prefix but it also contains type stack information for each type used in the YANG module represented in the *data* param. Together with the *meta* param, the *data* param constitutes a complete YANG module in JSON format.

Example

Example 9. Method get_schema

```
curl \
    --cookie "sessionid=sess11635875109111642;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "get_schema",
         "params": {"th": 2,
                    "path": "/aaa:aaa/authentication/users/user{admin}",
                    "levels": -1,
                    "insert_values": true}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"meta":
  {"namespace": "http://tail-f.com/ns/aaa/1.1",
   "keypath": "/aaa:aaa/authentication/users/user{admin}",
   "prefix": "aaa",
   "types":
   {"http://tail-f.com/ns/aaa/1.1:passwdStr":
    [{"name": "http://tail-f.com/ns/aaa/1.1:passwdStr"},
     {"name": "MD5DigestString"}]}}},
 "data":
 {"kind": "list-entry",
  "name": "user",
  "qname": "aaa:user",
  "access":
  {"create": true,
   "update": true,
   "delete": true},
  "children":
```

```
  [{"kind": "key",
    "name": "name",
    "qname": "aaa:name",
    "info": {"string": "Login name of the user"},
    "mandatory": true,
    "access": {"update": true},
    "type": {"name": "string", "primitive": true}},
   ...]}}
```

Method hide_schema

Hides data which has been adorned with a "hidden" statement in YANG modules. "hidden" statements is an extension defined in the tail-common YANG module (http://tail-f.com/yang/common).

Params

```
{"th": <integer>,
 "group_name": <string>
```

The *group_name* param is as defined by a "hidden" statement in a YANG module.

Result

```
{}
```

Method unhide_schema

Unhides data which has been adorned with a "hidden" statement in YANG modules. "hidden" statements is an extension defined in the tail-common YANG module (http://tail-f.com/yang/common).

Params

```
{"th": <integer>,
 "group_name": <string>,
 "passwd": <string>}
```

The *group_name* param is as defined by a "hidden" statement in a YANG module.

The *passwd* param is a password needed to hide the data that has been adorned with a "hidden" statement. The password is as defined in the ncs.conf file.

Result

```
{}
```

Method get_module_prefix_map

Returns a map from module name to module prefix.

Params

Method takes no parameters.

Result

```
<key-value object>
```

```
result = {"module-name": "module-prefix"}
```

Example

Example 10. Method get_module_prefix_map

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", id: 1,
         "method": "get_module_prefix_map",
         "params": {}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {
     "cli-builtin": "cli-builtin",
     "confd_cfg": "confd_cfg",
     "iana-crypt-hash": "ianach",
     "ietf-inet-types": "inet",
     "ietf-netconf": "nc",
     "ietf-netconf-acm": "nacm",
     "ietf-netconf-monitoring": "ncm",
     "ietf-netconf-notifications": "ncn",
     "ietf-netconf-with-defaults": "ncwd",
     "ietf-restconf": "rc",
     "ietf-restconf-monitoring": "rcmon",
     "ietf-yang-library": "yanglib",
     "ietf-yang-types": "yang",
     "tailf-aaa": "aaa",
     "tailf-acm": "tacm",
     "tailf-common-monitoring2": "tfcg2",
     "tailf-confd-monitoring": "tfcm",
     "tailf-confd-monitoring2": "tfcm2",
     "tailf-kicker": "kicker",
     "tailf-netconf-extensions": "tfnce",
     "tailf-netconf-monitoring": "tncm",
     "tailf-netconf-query": "tfncq",
     "tailf-rest-error": "tfrerr",
     "tailf-rest-query": "tfrestq",
     "tailf-rollback": "rollback",
     "tailf-webui": "webui",
    }
}
```

Method run_action

Invokes an action or rpc defined in a YANG module.

Params

```
{"th": <integer>,
 "path": <string>,
 "params": <json, optional>
 "format": <"normal" | "bracket" | "json", default: "normal">,
 "comet_id": <string, optional>,
 "handle": <string, optional>,
 "details": <"normal" | "verbose" | "very_verbose" | "debug", optional>}
```

Actions are as specified in th YANG module, i.e. having a specific name and a well defined set of parameters and result. the *path* param is a keypath pointing to an action or rpc in and the *params* param is a JSON object with action parameters.

The *format* param defines if the result should be an array of key values or a pre-formatted string on bracket format as seen in the CLI. The result is also as specified by the YANG module.

Both a *comet_id* and *handle* need to be provided in order to receive notifications.

The *details* param can be given together with *comet_id* and *handle* in order to get progress trace for the action. *details* specifies the verbosity of the progress trace. After the action has been invoked, the *comet* method can be used to get the progress trace for the action. If the *details* param is omitted progress trace will be disabled.

NOTE This method is often used to call an action that uploads binary data (e.g. images) and retrieving them at a later time. While retrieval is not a problem, uploading is a problem, because JSON-RPC request payloads have a size limitation (e.g. 64 kB). The limitation is needed for performance concerns because the payload is first buffered, before the JSON string is parsed and the request is evaluated. When you have scenarios that need binary uploads, please use the CGI functionality instead which has a size limitation that can be configured, and which is not limited to JSON payloads, so one can use streaming techniques.

Result

```
<string | array of result | key-value object>
```

```
result = {"name": <string>, "value": <string>}
```

Errors (specific)

```
{"type": "action.invalid_result", "data": {"path": <string, path to invalid result>}}
```

Example

Example 11. Method run_action

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", id: 1,
         "method": "run_action",
         "params": {"th": 2,
                    "path": "/dhcp:dhcp/set-clock",
                    "params": {"clockSettings": "2014-02-11T14:20:53.460%2B01:00"}}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": [{"name":"systemClock", "value":"0000-00-00T03:00:00+00:00"},
            {"name":"inlineContainer/bar", "value":"false"},
            {"name":"hardwareClock","value":"0000-00-00T04:00:00+00:00"}]}
```

```
curl \
    -s \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d'{"jsonrpc": "2.0", "id": 1,
        "method": "run_action",
        "params": {"th": 2,
                   "path": "/dhcp:dhcp/set-clock",
                   "params": {"clockSettings":
    "2014-02-11T14:20:53.460%2B01:00"},
                   "format": "bracket"}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result": "systemClock 0000-00-00T03:00:00+00:00\ninlineContainer  {\n    \
                    bar false\n}\nhardwareClock 0000-00-00T04:00:00+00:00\n"}

curl \
    -s \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d'{"jsonrpc": "2.0", "id": 1,
        "method": "run_action",
        "params": {"th": 2,
                   "path": "/dhcp:dhcp/set-clock",
                   "params": {"clockSettings":
    "2014-02-11T14:20:53.460%2B01:00"},
                   "format": "json"}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result": {"systemClock": "0000-00-00T03:00:00+00:00",
            "inlineContainer": {"bar": false},
            "hardwareClock": "0000-00-00T04:00:00+00:00"}}
```

Methods - session

Method login

Creates a user session and sets a browser cookie

Params

```
{}
```

```
{"user": <string>, "passwd": <string>, "ack_warning": <boolean, default: false>}
```

There are two versions of the *login* method. The method with no parameters only invokes Package Authentication, since credentials can be supplied with the whole HTTP request. The method with parameters is used when credentials may need to be supplied with the method parameters, this method invokes all authentication methods including Package Authentication.

The *user* and *passwd* are the credentials to be used in order to create a user session. The common AAA engine in NSO is used to verify the credentials.

If the method fails with a warning, the warning needs to be displayed to the user, along with a checkbox to allow the user to acknowledge the warning. The acknowledgement of the warning translates to setting *ack_warning* to true.

Result

```
{"warning": <string, optional>}
```

NOTE The response will have a `Set-Cookie` HTTP header with a *sessionid* cookie which will be your authentication token for upcoming JSON-RPC requests.

The *warning* is a free-text string that should be displayed to the user after a successful login. This is not to be mistaken with a failed login that has a *warning* as well. In case of a failure, the user should also acknowledge the warning, not just have it displayed for optional reading.

Multi factor authentication

```
{"challenge_id": <string>, "challenge_prompt": <string>}
```

NOTE A challenge response will have a *challenge_id* and *challenge_prompt* which needs to be responded to with an upcoming JSON-RPC challenge_response requests.

Note The challenge_prompt may be multi line, why it is base64 encoded.

Example

Example 12. Method login

```
curl \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "login",
         "params": {"user": "joe",
                    "passwd": "SWkkasE32"}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32000,
  "type": "rpc.method.failed",
  "message": "Method failed"}}

curl \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "login",
         "params": {"user": "admin",
                    "passwd": "admin"}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

NOTE sessionid cookie is set at this point in your User Agent (browser). In our examples, we set the cookie explicitly in the upcoming requests for clarity.

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "get_trans"}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "result": {"trans": []}}
```

Method challenge_response

Creates a user session and sets a browser cookie

Params

```
{"challenge_id": <string>, "response": <string>, "ack_warning": <boolean, default: false>}
```

The *challenge_id* and *response* is the multi factor response to be used in order to create a user session. The common AAA engine in NSO is used to verify the response.

If the method fails with a warning, the warning needs to be displayed to the user, along with a checkbox to allow the user to acknowledge the warning. The acknowledgement of the warning translates to setting *ack_warning* to true.

Result

```
{"warning": <string, optional>}
```

NOTE The response will have a `Set-Cookie` HTTP header with a *sessionid* cookie which will be your authentication token for upcoming JSON-RPC requests.

The *warning* is a free-text string that should be displayed to the user after a successful challenge response. This is not to be mistaken with a failed challenge response that has a *warning* as well. In case of a failure, the user should also acknowledge the warning, not just have it displayed for optional reading.

Example

Example 13. Method challenge response

```
curl \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "challenge_response",
         "params": {"challenge_id": "123",
                    "response": "SWkkasE32"}}' \
    http://127.0.0.1:8008/jsonrpc

{"jsonrpc": "2.0",
 "id": 1,
 "error":
 {"code": -32000,
  "type": "rpc.method.failed",
  "message": "Method failed"}}

curl \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "challenge_response",
         "params": {"challenge_id": "123",
                    "response": "SWEddrk1"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

NOTE sessionid cookie is set at this point in your User Agent (browser). In our examples, we set the cookie explicitly in the upcoming requests for clarity.

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "get_trans"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {"trans": []}}
```

Method logout

Removes a user session and invalidates the browser cookie

The HTTP cookie identifies the user session so no input parameters are needed.

Params

None.

Result

```
{}
```

Example

Example 14. Method logout

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "logout"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": {}}
```

```
curl \
    --cookie "sessionid=sess4245223558720207078;" \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "logout"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "error":
```

```
 {"code": -32000,
  "type": "session.invalid_sessionid",
  "message": "Invalid sessionid"}}
```

Method kick_user

Kills a user session, i.e. kicking out the user

Params

```
{"user": <string | number>}
```

The *user* param is either the username of a logged in user or session id.

Result

```
{}
```

Methods - session data

Method get_session_data

Gets session data from the session store

Params

```
{"key": <string>}
```

The *key* param for which to get the stored data for. Read more about the session store in the *put_session_data* method.

Result

```
{"value": <string>}
```

Method put_session_data

Puts session data into the session store. The session store is small key-value server-side database where data can be stored under a unique key. The data may be an arbitrary object, but not a function object. The object is serialized into a JSON string and then stored on the server.

Params

```
{"key": <string>,
 "value": <string>}
```

The *key* param is the unique key for which the data in the *value* param is to be stored.

Result

```
{}
```

Method erase_session_data

Erases session data previously stored with "put_session_data".

Params

```
{"key": <string>}
```

The *key* param for which all session data will be erased. Read more about the session store in the *put_session_data* method.

Result

```
{}
```

Methods - transaction

Method get_trans

Lists all transactions

Params

None.

Result

```
{"trans": <array of transaction>}

transaction =
 {"db": <"running" | "startup" | "candidate">,
  "mode": <"read" | "read_write", default: "read">,
  "conf_mode": <"private" | "shared" | "exclusive", default: "private">,
  "tag": <string>,
  "th": <integer>}
```

Example

Example 15. Method get_trans

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "get_trans"}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 {"trans":
  [{"db": "running",
    "th": 2}]}}
```

Method new_trans

Creates a new transaction

Params

```
{"db": <"startup" | "running" | "candidate", default: "running">,
 "mode": <"read" | "read_write", default: "read">,
 "conf_mode": <"private" | "shared" | "exclusive", default: "private">,
 "tag": <string>,
 "action_path": <string>,
 "th": <integer>,
 "on_pending_changes": <"reuse" | "reject" | "discard", default: "reuse">}
```

The *conf_mode* param specifies which transaction semantics to use when it comes to lock and commit strategies. These three modes mimics the modes available in the CLI.

The meaning of *private*, *shared* and *exclusive* have slightly different meaning depending on how the system is configured; with a writable running, startup or candidate configuration.

private (*writable running enabled*) - Edit a private copy of the running configuration, no lock is taken.

private (*writable running disabled, startup enabled*) - Edit a private copy of the startup configuration, no lock is taken.

exclusive (*candidate enabled*) - Lock the running configuration and the candidate configuration and edit the candidate configuration.

exclusive (*candidate disabled, startup enabled*) - Lock the running configuration (if enabled) and the startup configuration and edit the startup configuration.

shared (*writable running enabled, candidate enabled*) - Is a deprecated setting.

The *tag* param is a way to tag transactions with a keyword, so that they can be filtered out when you call the *get_trans* method.

The *action_path* param is a keypath pointing to an action or rpc. Use *action_path* when you need to read action/rpc input parameters.

The *th* param is a way to create transactions within other read_write transactions.

The *on_pending_changes* param decides what to do if the candidate already has been written to, e.g. a CLI user has started a shared configuration session and changed a value in the configuration (without committing it). If this parameter is omitted the default behavior is to silently reuse the candidate. If "reject" is specified the call to the "new_trans" method will fail if the candidate is non-empty. If "discard" is specified the candidate is silently cleared if it is non-empty.

Result

```
{"th": <integer>}
```

A new transaction handler id

Errors (specific)

```
{"type": "trans.confirmed_commit_in_progress"}
{"type": "db.locked", "data": {"sessions": <array of string>}}
```

The `data.sessions` param is an array of strings describing the current sessions of the locking user, e.g. an array of "admin tcp (cli from 192.245.2.3) on since 2006-12-20 14:50:30 exclusive".

Example

Example 16. Method new_trans

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
         "method": "new_trans",
         "params": {"db": "running",
                    "mode": "read"}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result": 2}
```

Method delete_trans

Deletes a transaction created by *new_trans* or *new_webui_trans*

Params

```
{"th": <integer>}
```

Result

```
{}
```

Method set_trans_comment

Adds a comment to the active read-write transaction. This comment will be stored in rollback files and can be seen with a call to *get_rollbacks*.

Params

```
{"th": <integer>}
```

Result

```
{}
```

Method set_trans_label

Adds a label to the active read-write transaction. This label will be stored in rollback files and can be seen with a call to *get_rollbacks*.

Params

```
{"th": <integer>}
```

Result

```
{}
```

Methods - transaction - changes

Method is_trans_modified

Checks if any modifications has been done to a transaction

Params

```
{"th": <integer>}
```

Result

```
{"modified": <boolean>}
```

Method get_trans_changes

Extracts modifications done to a transaction

Params

```
{"th": <integer>},
 "output": <"compact" | "legacy", default: "legacy">
```

The *output* parameter controls the result content. *legacy* format include old and value for all operation types even if their value is undefined. undefined values are represented by an empty string. *compact* format excludes old and value if their value is undefined.

Result

```
{"changes": <array of change>}

change =
 {"keypath": <string>,
  "op": <"created" | "deleted" | "modified" | "value_set">,
  "value": <string,>,
  "old": <string>
 }
```

The *value* param is only interesting if *op* is set to one of *modified* or *value_set*.

The *old* param is only interesting if *op* is set to *modified*.

Example

Example 17. Method get_trans_changes

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H 'Content-Type: application/json' \
    -d '{"jsonrpc": "2.0", "id": 1,
        "method": "changes",
        "params": {"th": 2}}' \
    http://127.0.0.1:8008/jsonrpc
```

```
{"jsonrpc": "2.0",
 "id": 1,
 "result":
 [{"keypath":"/dhcp:dhcp/default-lease-time",
   "op": "value_set",
   "value": "100",
   "old": ""}]}
```

Method validate_trans

Validates a transaction.

Params

```
{"th": <integer>}
```

Result

```
{}
```

or

```
{"warnings": <array of warning>}
```

```
warning = {"paths": <array of string>, "message": <string>}
```

Errors (specific)

```
{"type": "trans.resolve_needed", "data": {"users": <array string>}}
```

The *data.users* param is an array of conflicting usernames.

```
{"type": "trans.validation_failed", "data": {"errors": <array of error>}}
```

```
error = {"paths": <array of string>, "message": <string>}
```

The *data.errors* param points to a keypath that is invalid.

Method get_trans_conflicts

Gets the conflicts registered in a transaction

Params

```
{"th": <integer>}
```

Result

```
{"conflicts:" <array of conflicts>}

conflict =
 {"keypath": <string>,
  "op": <"created" | "deleted" | "modified" | "value_set">,
  "value": <string>,
  "old": <string>}
```

The *value* param is only interesting if *op* is set to one of *created*, *modified* or *value_set*.

The *old* param is only interesting if *op* is set to *modified*.

Method resolve_trans

Tells the server that the conflicts have been resolved

Params

```
{"th": <integer>}
```

Result

```
{}
```

Methods - transaction - commit changes

Method validate_commit

Validates a transaction before calling *commit*. If this method succeeds (with or without warnings) then the next operation *must* be a call to either *commit* or *clear_validate_lock*. The configuration will be locked for access by other users until one of these methods are called.

Params

```
{"th": <integer>}
```

```
{"comet_id": <string, optional>}
```

```
{"handle": <string, optional>}
```

```
{"details": <"normal" | "verbose" | "very_verbose" | "debug", optional>}
```

```
{"flags": <flags, default: []>}
flags = <array of string or bitmask>
```

The *comet_id*, *handle*, and *details* params can be given together in order to get progress tracing for the validate_commit operation. The same *comet_id* can also be used to get the progress trace for any coming commit operations. In order to get progress tracing for commit operations, these three parameters have to be provided with the validate_commit operation. The *details* parameter specifies the verbosity of the progress trace. After the operation has been invoked, the *comet* method can be used to get the progress trace for the operation.

See the *commit* method for available flags.

NOTE: If you intend to pass *flags* to the *commit* method, it is recommended to pass the same *flags* to *validate_commit* since they may have an effect during the validate step.

Result

```
{}
```

or

```
{"warnings": <array of warning>}
```

```
warning = {"paths": <array of string>, "message": <string>}
```

Errors (specific)

Same as for the *validate_trans* method.

Method clear_validate_lock

Releases validate lock taken by *validate_commit*

Params

```
{"th": <integer>}
```

Result

```
{}
```

Method commit

Copies the configuration into the running datastore.

Params

```
{"th": <integer>,
 "timeout": <integer, default: 0>,
 "release_locks": <boolean, default: true>,
 "rollback-id": <boolean, default: true>}
```

The commit with a *timeout* parameter represents a confirmed commit.

If rollback-id is set to true the response will include the id of the rollback file created during the commit if any.

Commit behaviour can be changed via an extra *flags* param:

```
{"flags": <flags, default: []>}
```

```
flags = <array of string or bitmask>
```

The *flags* param is a list of flags that can change the commit behavior:

- dry-run=FORMAT - Where FORMAT is the desired output format: xml, cli or native. Validate and display the configuration changes but do not perform the actual commit. *Neither* CDB *nor* the devices are affected. Instead the effects that would have taken place is showed in the returned output.
- dry-run-reverse - Used with the dry-run=native flag this will display the device commands for getting back to the current running state in the network if the commit is successfully executed. Beware that if any changes are done later on the same data the reverse device commands returned are invalid.
- no-revision-drop - NSO will not run its data model revision algorithm, which requires all participating managed devices to have all parts of the data models for all data contained in this transaction. Thus, this flag forces NSO to never silently drop any data set operations towards a device.
- no-overwrite - NSO will check that the data that should be modified has not changed on the device compared to NSO's view of the data. Can't be used with no-out-of-sync-check.
- no-networking - Do not send data to the devices; this is a way to manipulate CDB in NSO without generating any southbound traffic.
- no-out-of-sync-check - Continue with the transaction even if NSO detects that a device's configuration is out of sync. Can't be used with no-overwrite.
- no-deploy - Commit without invoking the service create method, i.e, write the service instance data without activating the service(s). The service(s) can later be re-deployed to write the changes of the service(s) to the network.
- reconcile=OPTION - Reconcile the service data. All data which existed before the service was created will now be owned by the service. When the service is removed that data will also be removed. In technical terms the reference count will be decreased by one for everything which existed prior to the service. If manually configured data exists below in the configuration tree that data is kept unless the option *discard-non-service-config* is used.
- use-lsa - Force handling of the LSA nodes as such. This flag tells NSO to propagate applicable commit flags and actions to the LSA nodes without applying them on the upper NSO node itself. The commit flags affected are **dry-run**, **no-networking**, **no-out-of-sync-check**, **no-overwrite** and **no-revision-drop**.
- no-lsa - Do not handle any of the LSA nodes as such. These nodes will be handled as any other device.

- commit-queue=MODE - Where MODE is: async, sync or bypass. Commit the transaction data to the commit queue. If the *async* value is set the operation returns successfully if the transaction data has been successfully placed in the queue. The *sync* value will cause the operation to not return until the transaction data has been sent to all devices, or a timeout occurs. The *bypass* value means that if `/devices/global-settings/commit-queue/enabled-by-default` is *true* the data in this transaction will bypass the commit queue. The data will be written directly to the devices.
- commit-queue-atomic=ATOMIC - Where ATOMIC is: true or false. Sets the atomic behaviour of the resulting queue item. If ATOMIC is set to false, the devices contained in the resulting queue item can start executing if the same devices in other non-atomic queue items ahead of it in the queue are completed. If set to true, the atomic integrity of the queue item is preserved.
- commit-queue-block-others - The resulting queue item will block subsequent queue items, which use any of the devices in this queue item, from being queued.
- commit-queue-lock - Place a lock on the resulting queue item. The queue item will not be processed until it has been unlocked, see the actions **unlock** and **lock** in `/devices/commit-queue/queue-item`. No following queue items, using the same devices, will be allowed to execute as long as the lock is in place.
- commit-queue-tag=TAG - Where TAG is a user defined opaque tag. The tag is present in all notifications and events sent referencing the specific queue item.
- commit-queue-timeout=TIMEOUT - Where TIMEOUT is infinity or a positive integer. Specifies a maximum number of seconds to wait for the transaction to be committed. If the timer expires, the transaction data is kept in the commit-queue, and the operation returns successfully. If the timeout is not set, the operation waits until completion indefinitely.
- commit-queue-error-option=OPTION - Where OPTION is: continue-on-error, rollback-on-error or stop-on-error. Depending on the selected error option NSO will store the reverse of the original transaction to be able to undo the transaction changes and get back to the previous state. This data is stored in the `/devices/commit-queue/completed` tree from where it can be viewed and invoked with the **rollback** action. When invoked the data will be removed. The *continue-on-error* value means that the commit queue will continue on errors. No rollback data will be created. The *rollback-on-error* value means that the commit queue item will roll back on errors. The commit queue will place a lock with `block-others` on the devices and services in the failed queue item. The **rollback** action will then automatically be invoked when the queue item has finished its execution. The lock is removed as part of rollback. The *stop-on-error* means that the commit queue will place a lock with `block-others` on the devices and services in the failed queue item. The lock must then either manually be released when the error is fixed or the **rollback** action under `/devices/commit-queue/completed` be invoked.

Note Read about error recovery in the section called “Commit Queue” in *User Guide* for a more detailed explanation.

- trace-id=TRACE_ID - Use the provided trace id as part of the log messages emitted while processing. If no trace id is given, NSO is going to generate and assign a trace id to the processing.

For backwards compatibility, the *flags* param can also be a bit mask with the following limit values:

- `1 << 0` - Do not release locks, overridden by the *release_locks* if set
- `1 << 2` - Do not drop revision

- If a call to *confirm_commit* is not done within *timeout* seconds an automatic rollback is performed. This method can also be used to "extend" a confirmed commit that is already in progress, i.e. set a new timeout or add changes.
- A call to *abort_commit* can be made to abort the confirmed commit.

NOTE: Must be preceded by a call to *validate_commit*

NOTE: The transaction handler is deallocated as a side effect of this method

Result

Successful commit without any arguments

```
{}
```

Successful commit with *rollback-id=true*

```
{"rollback-id": {"fixed": 10001}}
```

Successful commit with *commit-queue=async*

```
{"commit_queue_id": <integer>}
```

The *commit_queue_id* is returned if the commit entered the commit queue, either by specifying *commit-queue=async* or by enabling it in the configuration.

Errors (specific)

```
{"type": "trans.confirmed_commit_in_progress"}
```

```
{"type": "trans.confirmed_commit_is_only_valid_for_candidate"}
```

```
{"type": "trans.confirmed_commit_needs_config_writable_through_candidate"}
```

```
{"type": "trans.confirmed_commit_not_supported_in_private_mode"}
```

Method abort_commit

Aborts the active read-write transaction

Result

```
{}
```

Method confirm_commit

Confirms the currently pending confirmed commit

Result

```
{}
```

Methods - transaction - webui

Method get_webui_trans

Gets the webui read-write transaction

Result

```
{"trans": <array of trans>}
```

```
trans =
 {"db": <"startup" | "running" | "candidate", default: "running">,
  "conf_mode": <"private" | "shared" | "exclusive", default: "private">,
  "th": <integer>
 }
```

Method new_webui_trans

Creates a read-write transaction that can be retrieved by 'get_webui_trans'.

Params

```
{"db": <"startup" | "running" | "candidate", default: "running">,
 "conf_mode": <"private" | "shared" | "exclusive", default: "private">
 "on_pending_changes": <"reuse" | "reject" | "discard", default: "reuse">}
```

See 'new_trans' for semantics of the parameters and specific errors.

The *on_pending_changes* param decides what to do if the candidate already has been written to, e.g. a CLI user has started a shared configuration session and changed a value in the configuration (without committing it). If this parameter is omitted the default behavior is to silently reuse the candidate. If "reject" is specified the call to the "new_webui_trans" method will fail if the candidate is non-empty. If "discard" is specified the candidate is silently cleared if it is non-empty.

Result

```
{"th": <integer>}
```

A new transaction handler id

Methods - NSO specific

Method get_template_variables

Extracts all variables from a NSO service/device template

Params

```
{"th": <integer>,
 "name": <string>}
```

The *name* param is the name of the template to extract variables from.

Result

```
{"template_variables": <array of string>}
```

Method list_packages

Lists packages in NSO

Params

```
{"status": <"installable" | "installed" | "loaded" | "all", default: "all">}
```

The *status* param specifies which package status to list:

- *installable* - an array of all packages that can be installed
- *installed* - an array of all packages that are installed, but not loaded
- *loaded* - an array of all loaded packages
- *all* - all of the above is returned.

Result

```
{"packages": <array of key-value objects>}
```

CHAPTER 3

The web server

- Introduction, page 69
- Web server capabilities, page 69
- CGI support, page 70
- Storing TLS data in database, page 71
- Package upload, page 73

Introduction

This document describes an embedded basic web server that can deliver static and Common Gateway Interface (CGI) dynamic content to a web client, commonly a browser. Due to the limitations of this web server, and/or of its configuration capabilities, a proxy server such as Nginx is recommended to address special requirements.

Web server capabilities

The web server can be configured through settings in ncs.conf - see the manual pages of the section called “CONFIGURATION PARAMETERS” in *Manual Pages* .

Here is a brief overview of what you can configure on the web server:

- "toggle web server": the web server can be turned on or off
- "toggle transport": enable HTTP and/or HTTPS, set IPs, ports, redirects, certificates, etc.
- "hostname": set the hostname of the web server and decide whether to block requests for other hostnames
- "/": set the docroot from where all static content is served
- "/login": set the docroot from where static content is served for URL paths starting with /login
- "/custom": set the docroot from where static content is served for URL paths starting with /custom
- "/cgi": toggle CGI support and set the docroot from where dynamic content is served for URL paths starting with /cgi
- "non-authenticated paths": by default all URL paths, except those needed for the login page are hidden from non-authenticated users; authentication is done by calling the JSONRPC "login" method
- "allow symlinks": allow symlinks from under the docroot
- "cache": set the cache time window for static content

- "log": several logs are available to configure in terms of file paths - an access log, a full HTTP traffic/trace log and a browser/JavaScript log
- "custom headers": set custom headers across all static and dynamic content, including requests to "/jsonrpc".

In addition to what is configurable, the web server also GZip-compresses responses automatically if the browser handles such responses, either by compressing the response on the fly, or, if requesting a static file, like "/bigfile.txt", by responding with the contents of "/bigfile.txt.gz", if there is such a file.

CGI support

The web server includes CGI functionality, disabled by default. Once you enable it in ncs.conf - see the manual pages of the section called “CONFIGURATION PARAMETERS” in *Manual Pages* , you can write CGI scripts, that will be called with the following NSO environment variables prefixed with NCS_ when a user has logged-in via JSON-RPC:

- "JSONRPC_SESSIONID": the JSON-RPC session id (cookie)
- "JSONRPC_START_TIME": the start time of the JSON-RPC session
- "JSONRPC_END_TIME": the end time of the JSON-RPC session
- "JSONRPC_READ": the latest JSON-RPC read transaction
- "JSONRPC_READS": a comma-separated list of JSON-RPC read transactions
- "JSONRPC_WRITE": the latest JSON-RPC write transaction
- "JSONRPC_WRITES": a comma-separated of JSON-RPC write transactions
- "MAAPI_USER": the MAAPI username
- "MAAPI_GROUPS": a comma-separated list of MAAPI groups
- "MAAPI_UID": the MAAPI UID
- "MAAPI_GID": the MAAPI GID
- "MAAPI_SRC_IP": the MAAPI source IP address
- "MAAPI_SRC_PORT": the MAAPI source port
- "MAAPI_USID": the MAAPI USID
- "MAAPI_READ": the latest MAAPI read transaction
- "MAAPI_READS": a comma-separated list of MAAPI read transactions
- "MAAPI_WRITE": the latest MAAPI write transaction
- "MAAPI_WRITES": a comma-separated of MAAPI write transactions

Server or HTTP specific information is also exported as environment variables:

- "SERVER_SOFTWARE":
- "SERVER_NAME":
- "GATEWAY_INTERFACE":
- "SERVER_PROTOCOL":
- "SERVER_PORT":
- "REQUEST_METHOD":
- "REQUEST_URI":
- "DOCUMENT_ROOT":
- "DOCUMENT_ROOT_MOUNT":
- "SCRIPT_FILENAME":
- "SCRIPT_TRANSLATED":

- "PATH_INTO":
- "PATH_TRANSLATED":
- "SCRIPT_NAME":
- "REMOTE_ADDR":
- "REMOTE_HOST":
- "SERVER_ADDR":
- "LOCAL_ADDR":
- "QUERY_STRING":
- "CONTENT_TYPE":
- "CONTENT_LENGTH":
- "HTTP_*": HTTP headers e.g. "Accept" value is exported as HTTP_ACCEPT

Storing TLS data in database

The `tailf-tls.yang` YANG module defines a structure to store TLS data in the database. It is possible to store the private key, the private key's passphrase, the public key certificate, and CA certificates.

In order to enable the web server to fetch TLS data from the database, `ncs.conf` needs to be configured

Example 18. Configuring NSO to read TLS data from database

```
<webui>
  <transport>
    <ssl>
      <enabled>true</enabled>
      <ip>0.0.0.0</ip>
      <port>8889</port>
      <read-from-db>true</read-from-db>
    </ssl>
  </transport>
</webui>
```

Note that the options *key-file*, *cert-file*, and *ca-cert-file*, are ignored when *read-from-db* is set to true. See the ncs.conf.5 man page for more details.

The database is populated with TLS data by configuring the */tailf-tls:tls/private-key*, */tailf-tls:tls/certificate*, and, optionally, */tailf-tls/ca-certificates*. It is possible to use password protected private keys, then the *passphrase* leaf in the *private-key* container needs to be set to the password of the encrypted private key. Unencrypted private key data can be supplied in both PKCS#8 and PKCS#1 format, while encrypted private key data needs to be supplied in PKCS#1 format.

In the following example a password protected private key, the passphrase, a public key certificate, and two CA certificates are configured with the CLI.

Example 19. Populating the database with TLS data

```
admin@io> configure
Entering configuration mode private
[ok][2019-06-10 19:54:21]

[edit]
admin@io% set tls certificate cert-data
(<unknown>):
```

```
[Multiline mode, exit with ctrl-D.]
> -----BEGIN CERTIFICATE-----
> MIICrzCCAZcCFBh0ETLcNAFCCEcjSrrd5U4/a6vuMA0GCSqGSIb3DQEBCwUAMBQx
> ...
> -----END CERTIFICATE-----
>
[ok][2019-06-10 19:59:36]

[edit]
admin@confd% set tls private-key key-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN RSA PRIVATE KEY-----
> Proc-Type: 4,ENCRYPTED
> DEK-Info: AES-128-CBC,6E816829A93AAD3E0C283A6C8550B255
> ...
> -----END RSA PRIVATE KEY-----
[ok][2019-06-10 20:00:27]

[edit]
admin@confd% set tls private-key passphrase
(<AES encrypted string>): ********
[ok][2019-06-10 20:00:39]

[edit]
admin@confd% set tls ca-certificates ca-cert-1 cert-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN CERTIFICATE-----
> MIIDCTCCAfGgAwIBAgIUbzrNvBdM7p2rxwDBaqF5xN1gfmEwDQYJKoZIhvcNAQEL
> ...
> -----END CERTIFICATE-----
[ok][2019-06-10 20:02:22]

[edit]
admin@confd% set tls ca-certificates ca-cert-2 cert-data
(<unknown>):
[Multiline mode, exit with ctrl-D.]
> -----BEGIN CERTIFICATE-----
> MIIDCTCCAfGgAwIBAgIUZ2GcDzHg44c2g7Q0Xlu3H8/4wnwwDQYJKoZIhvcNAQEL
> ...
> -----END CERTIFICATE-----
[ok][2019-06-10 20:03:07]

[edit]
admin@confd% commit
Commit complete.
[ok][2019-06-10 20:03:11]

[edit]
```

The SHA256 fingerprints of the public key certificate and the CA certificates can be accessed as operational data. The fingerprint is shown as a hex string. The first octet identifies what hashing algorithm is used, *04* is SHA256, and the following octets is the actual fingerprint.

Example 20. Show TLS certificate fingerprints

```
admin@io> show tls
tls certificate fingerprint 04:65:8a:9e:36:2c:a7:42:8d:93:50:af:97:08:ff:e6:1b:c5:43:a8:2c:b5:bf
NAME      FINGERPRINT
------------------------------------------------------------------------------------------------
```

```
cacert-1  04:00:5e:22:f8:4b:b7:3a:47:e7:23:11:80:03:d3:9a:74:8d:09:c0:fa:cc:15:2b:7f:81:1a:e6
cacert-2  04:2d:93:9b:37:21:d2:22:74:ad:d9:99:ae:76:b6:6a:f2:3b:e3:4e:07:32:f2:8b:f0:63:ad:21

[ok][2019-06-10 20:43:31]
```

When the database is populated NSO needs to be reloaded.

```
$ ncs --reload
```

After configuring NSO, populating the database, and reloading, the TLS transport is usable.

```
$ curl -kisu admin:admin https://localhost:8889
HTTP/1.1 302 Found
...
```

Package upload

The web server includes support for uploading packages to "/package-upload" using HTTP POST from the local host to the NSO host, making them *installable* there. It is disabled by default, but can be enabled in ncs.conf - see the manual pages of the section called “CONFIGURATION PARAMETERS” in *Manual Pages* .

By default only upload 1 file per request will be processed and any remaining file parts after that will result in an error and its content will be ignored. To allow multiple files in a request you can increase `/ncs-config/webui/package-upload/max-files`.

Example 21. Valid package example

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H "Cache-Control: no-cache" \
    -F "upload=@path/to/some-valid-package.tar.gz" \
    http://127.0.0.1:8080/package-upload

[
    {
        "result": {
            "filename": "some-valid-package.tar.gz"
        }
    }
]
```

Example 22. Invalid package example

```
curl \
    --cookie 'sessionid=sess12541119146799620192;' \
    -X POST \
    -H "Cache-Control: no-cache" \
    -F "upload=@path/to/some-invalid-package.tar.gz" \
    http://127.0.0.1:8080/package-upload

[
    {
        "error": {
            "filename": "some-invalid-package.tar.gz",
            "data": {
                "reason": "Invalid package contents"
            }
        }
    }
]
```

The AAA infrastructure can be used to restrict access to library functions using command rules:

```
<cmdrule xmlns="http://tail-f.com/yang/acm">
<name>deny-package-upload</name>
<context>webui</context>
<command>::webui:: package-upload</command>
<access-operations>exec</access-operations>
<action>deny</action>
</cmdrule>
```

Note how the command is prefixed with "::webui:: ". This tells the AAA engine to apply the command rule to webui API functions.

You can read more about command rules in "The AAA infrastructure" chapter in this User Guide.

CHAPTER 4

Single Sign-On

The Single Sign-On functionality enables users to login via HTTP based northbound APIs with a single sign-on authentication scheme, such as SAMLv2. Currently it is only supported for the JSON-RPC northbound interface.

Note For Single Sign-On to work, the Package Authentication needs to be enabled (see the section called “Package Authentication” in *Administration Guide*).

When enabled, the endpoint `/sso` is made public and handles single sign-on attempts.

An example configuration for the cisco-nso-saml2-auth Authentication Package is presented below. Note that `/ncs-config/aaa/auth-order` does not need to be set for Single Sign-On to work!

Example 23. Example `ncs.conf` to enable SAMLv2 Single Sign-On.

```
<aaa>
  <package-authentication>
    <enabled>true</enabled>
    <packages>
      <package>cisco-nso-saml2-auth</package>
    </packages>
  </package-authentication>
  <single-sign-on>
    <enabled>true</enabled>
  </single-sign-on>
</aaa>
```

A client attempting single sign-on authentication should request the `/sso` endpoint and then follow the continued authentication operation from there. For example, for cisco-nso-saml2-auth the client is redirected to an Identity Provider (IdP), which subsequently handles the authentication, and then redirects the client back to the `/sso` endpoint to validate the authentication and setup the session.