



## **NSO 4.4.2.3 Web UI**

**First Published:** May 17, 2010

**Last Modified:** September 1, 2017

### **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

THE SPECIFICATIONS AND INFORMATION REGARDING THE PRODUCTS IN THIS MANUAL ARE SUBJECT TO CHANGE WITHOUT NOTICE. ALL STATEMENTS, INFORMATION, AND RECOMMENDATIONS IN THIS MANUAL ARE BELIEVED TO BE ACCURATE BUT ARE PRESENTED WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. USERS MUST TAKE FULL RESPONSIBILITY FOR THEIR APPLICATION OF ANY PRODUCTS.

THE SOFTWARE LICENSE AND LIMITED WARRANTY FOR THE ACCOMPANYING PRODUCT ARE SET FORTH IN THE INFORMATION PACKET THAT SHIPPED WITH THE PRODUCT AND ARE INCORPORATED HEREIN BY THIS REFERENCE. IF YOU ARE UNABLE TO LOCATE THE SOFTWARE LICENSE OR LIMITED WARRANTY, CONTACT YOUR CISCO REPRESENTATIVE FOR A COPY.

The Cisco implementation of TCP header compression is an adaptation of a program developed by the University of California, Berkeley (UCB) as part of UCB's public domain version of the UNIX operating system. All rights reserved. Copyright © 1981, Regents of the University of California.

NOTWITHSTANDING ANY OTHER WARRANTY HEREIN, ALL DOCUMENT FILES AND SOFTWARE OF THESE SUPPLIERS ARE PROVIDED "AS IS" WITH ALL FAULTS. CISCO AND THE ABOVE-NAMED SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE.

IN NO EVENT SHALL CISCO OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOST PROFITS OR LOSS OR DAMAGE TO DATA ARISING OUT OF THE USE OR INABILITY TO USE THIS MANUAL, EVEN IF CISCO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Any Internet Protocol (IP) addresses and phone numbers used in this document are not intended to be actual addresses and phone numbers. Any examples, command display output, network topology diagrams, and other figures included in the document are shown for illustrative purposes only. Any use of actual IP addresses or phone numbers in illustrative content is unintentional and coincidental.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, go to this URL: <http://www.cisco.com/go/trademarks>. Third-party trademarks mentioned are the property of their respective owners. The use of the word partner does not imply a partnership relationship between Cisco and any other company. (1110R)

This product includes software developed by the NetBSD Foundation, Inc. and its contributors.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit <http://www.openssl.org/>.

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

U.S. Pat. No. 8,533,303 and 8,913,519

Copyright © 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017 Cisco Systems, Inc. All rights reserved.



## CONTENTS

---

### CHAPTER 1

<b>Web UI Development</b>	<b>1</b>
Introduction	1
Example of a common flow	2
Example of a JSON-RPC client	6
Example of a Comet client	9

---

### CHAPTER 2

<b>The JSON-RPC API</b>	<b>13</b>
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 start_subscription	24
Method unsubscribe	24

Methods - data	25
Method create	25
Method delete	25
Method exists	26
Method get_case	26
Method show_config	26
Methods - data - attrs	26
Method get_attrs	26
Method set_attrs	27
Methods - data - leafs	27
Method get_value	27
Method get_values	28
Method set_value	28
Methods - data - leafref	29
Method deref	29
Method get_leafref_values	29
Methods - data - lists	30
Method rename_list_entry	30
Method copy_list_entry	31
Method move_list_entry	31
Method append_list_entry	32
Method count_list_keys	32
Method get_list_keys	32
Methods - data - query	33
Method query	33
Method start_query	33
Method run_query	35
Method reset_query	36
Method stop_query	37
Methods - database	37
Method reset_candidate_db	37
Method lock_db	37
Method unlock_db	38
Method copy_running_to_startup_db	38
Methods - general	38

Method comet	38
Method get_system_setting	39
Method abort	40
Method eval_XPath	40
Methods - messages	41
Method send_message	41
Method subscribe_messages	41
Methods - rollbacks	42
Method get_rollbacks	42
Method get_rollback	42
Method install_rollback	42
Method load_rollback	43
Methods - schema	43
Method get_schema	43
Method hide_schema	47
Method unhide_schema	47
Method run_action	47
Methods - session	49
Method login	49
Method logout	50
Method kick_user	50
Methods - session data	51
Method get_session_data	51
Method put_session_data	51
Method erase_session_data	51
Methods - transaction	52
Method get_trans	52
Method new_trans	52
Method delete_trans	54
Method set_trans_comment	54
Method set_trans_label	54
Methods - transaction - changes	54
Method is_trans_modified	54
Method get_trans_changes	54
Method validate_trans	55

Method get_trans_conflicts	56
Method resolve_trans	56
Methods - transaction - commit changes	56
Method validate_commit	56
Method clear_validate_lock	57
Method commit	57
Method abort_commit	58
Method confirm_commit	58
Methods - transaction - webui	58
Method get_webui_trans	58
Method new_webui_trans	59
Methods - NSO specific	59
Method get_service_points	59
Method get_template_variables	59

---

## CHAPTER 3

### The web server 61

Introduction	61
Web server capabilities	61
CGI support	62



## 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

Cisco Network Service Orchestrator enabled by Tail-f (NSO) version 4.4.2.3 is an evolution of the Tail-f Network Control System (NCS). Tail-f was acquired by Cisco in 2014. The product has been enhanced and forms the base for Cisco NSO. Note that the terms 'ncs' and 'tail-f' are used extensively in file names, command-line command names, YANG models, application programming interfaces (API), etc. Throughout this document we will use NSO to mean the product, which consists of a number of modules and executable components. These executable components will be referred to by their command line name, e.g. **ncs**, **ncs-netsim**, **ncs\_cli**, etc. **ncs** is also used to refer to the executable, the running daemon.

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 northbound 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 \[http://www.jsonrpc.org/specification\]](http://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 \[https://jquery.com/\]](https://jquery.com/)) or from the command line (e.g. [curl \[https://github.com/bagder/curl\]](https://github.com/bagder/curl), [httpie \[https://github.com/jkbr/httpie\]](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 + '.');
  });
};

funcs.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.');
```

```

  });
};

funcs.commit = function(args /*{th}*/) {
  log('Committing 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('Committed.');
```

```

  });
};

funcs.subscribeChanges = function(args /*{path, handle}*/) {
  log('Subscribing 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([
  funcs.login,
  funcs.getSystemSetting,
  funcs.newReadTrans,
  function() {
    return funcs.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) {
            // Unsubscribe 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 unsubscribe from it, in order to release
        // 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 25](#)
- [Methods - data - attrs, page 26](#)
- [Methods - data - leafs, page 27](#)
- [Methods - data - leafref, page 29](#)
- [Methods - data - lists, page 30](#)
- [Methods - data - query, page 33](#)
- [Methods - database, page 37](#)
- [Methods - general, page 38](#)
- [Methods - messages, page 41](#)
- [Methods - rollbacks, page 42](#)
- [Methods - schema, page 43](#)
- [Methods - session, page 49](#)
- [Methods - session data, page 51](#)
- [Methods - transaction, page 52](#)
- [Methods - transaction - changes, page 54](#)
- [Methods - transaction - commit changes, page 56](#)
- [Methods - transaction - webui, page 58](#)
- [Methods - NSO specific, page 59](#)

## JSON-RPC

### Protocol overview

The JSON-RPC 2.0 Specification [<http://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 "subtract"):

```
[ { "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 set, 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
  }
})
```

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 over 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 *changes* method. See above.

## 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 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)

## Params

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

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

## Result

```
{}
```

## 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 a compact string representation of the configuration

### Params

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

The *path* param is a keypath to the configuration to be returned in a compact string format.

### Result

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

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

## Result

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

### 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>,
  "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.

### 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.

### Result

```

{}

```

### Errors (specific)

```

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

```

**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>,
 "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.

The *skip\_grouping* param is by default set to false and is only needed to be set to true if a set of sibling leafref leaves 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.

### 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%gt; }
```

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 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", 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 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 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">,
  "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 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 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 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 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" | "n

```

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 *xpath\_expr* param is the XPath expression to be evaluated.

### 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*.

#### 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").

## 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\_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> }
```

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.

## 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>,
  "is_action_input": <boolean>,
  "is_action_output": <boolean>,
  "is_cli_preformatted": <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>,
  "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 if "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 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 `run_action`

Invokes an action or rpc defined in a YANG module.

### Params

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

Actions are as specified in the 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.

**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>
```

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

## Errors (specific)

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

### Example 10. 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  \n
  bar false\n}\nhardwareClock 0000-00-00T04:00:00+00:00\n"}

```

# Methods - session

## Method login

Creates a user session and sets a browser cookie

### Params

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

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.

#### Example 11. 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 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 12. 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 13. 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>,
  "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\*) - Edit the candidate configuration without locking it.

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 *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 14. 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>}
```

## 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 *created*, *modified* or *value\_set*.

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

### Example 15. 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 all 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>}
```

### 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>}
```

The *timeout* param represents the confirmed commit timeout, i.e. set it to zero (0) to commit without timeout.

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:

- no-revision-drop - do not drop revision
- async-commit-queue - if some device is non-operational or has data waiting in the commit queue, the data in this transaction will be placed in the commit queue; a *commit\_queue\_id* will be found in the response
- no-networking - do not send data to the devices; this is a way to manipulate CDB in NSO without generating any southbound traffic
- dry-run=XXX - where XXX is the desired output format: xml, cli or native. Do not send data to the devices; this is a way to see the effects that would have taken place as a diff
- no-out-of-sync-check - continue with the transaction even if NSO detects that a device's configuration is out of sync
- 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
- sync-commit-queue - if some device is non-operational or has data waiting in the commit queue, the data in this transaction will be placed in the commit queue; that flag will cause the operation to not

return until the transaction has been sent to all devices, or a timeout occurs; a *commit\_queue\_id* will be found in the response if the timeout occurs

- `sync-commit-timeout=XXX` - where XXX is a timeout in seconds; only allowed after `sync-commit-queue`

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

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

The *commit\_queue\_id* is only returned if the method is run with the *async\_commit\_queue* flag, or with the *sync\_commit\_queue* flag and a timeout occurs.

## Errors (specific)

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

## 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\_service\_points

Extracts paths to all NSO services

### Result

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

### 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>
}
```





## CHAPTER 3

# The web server

---

- [Introduction, page 61](#)
- [Web server capabilities, page 61](#)
- [CGI support, page 62](#)

## 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 *NSO 4.4.2.3 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 *NSO 4.4.2.3 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\_INFO":
- "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

