# clixon

*Release 4.3*

Feb 23, 2020

# Table of contents


**1 Overview** **3**
- 1.1 System Architecture . . . 3
- 1.2 Platforms . . . 4
- 1.3 Standards . . . 4
- 1.4 How to get Clixon . . . 4
- 1.5 Support . . . 4
- 1.6 Bug reports . . . 4

**2 Installation** **5**
- 2.1 Ubuntu Linux . . . 5
- 2.2 FreeBSD . . . 6
- 2.3 Systemd . . . 6
- 2.4 Advanced install . . . 7

**3 Quick start** **9**
- 3.1 Content . . . 9
- 3.2 Compile and run . . . 9
- 3.3 Using the CLI . . . 10
- 3.4 Netconf . . . 10
- 3.5 Restconf . . . 10
- 3.6 Run a container . . . 11
- 3.7 Next steps . . . 12

**4 Standards** **13**
- 4.1 YANG . . . 13
- 4.2 XML and XPath . . . 14
- 4.3 NETCONF . . . 14
- 4.4 RESTCONF . . . 15
- 4.5 JSON . . . 15

**5 Configuration** **17**
- 5.1 Example . . . 17
- 5.2 Specification . . . 17
- 5.3 Finding the configuration . . . 18
- 5.4 Runtime modification . . . 18
- 5.5 Features . . . 18
- 5.6 Finding YANG files . . . 19

5.7 Default values . . . 20

**6 Backend** **21**
6.1 Command-line options . . . 22
6.2 Startup . . . 23
6.3 Socket . . . 23
6.4 Backend files . . . 24
6.5 Plugins . . . 24
6.6 Transactions . . . 25
6.7 Upgrade . . . 25
6.8 Privileges . . . 31

**7 Datastore** **33**
7.1 Datastore files . . . 33
7.2 Datastore format . . . 34
7.3 Module library support . . . 34
7.4 Datastore caching . . . 35

**8 CLI** **37**
8.1 Using the CLI . . . 37
8.2 CLI specs and plugins . . . 37
8.3 Modes . . . 38
8.4 History . . . 39

**9 Usecases** **41**
9.1 CLI read . . . 41
9.2 CLI write . . . 42
9.3 Commit . . . 43
9.4 RESTCONF RPC . . . 44

**10 Paths** **45**
10.1 XPath . . . 45
10.2 Instance-identifier . . . 46
10.3 Api-path . . . 46

**11 XML tree** **49**
11.1 Creating XML . . . 49
11.2 Searching in XML . . . 51
11.3 Internal representation . . . 54

**12 Advanced** **57**
12.1 CLI . . . 57

Clixon is a YANG-based configuration manager, with interactive CLI, NETCONF and RESTCONF interfaces, an embedded database and transaction mechanism.

CHAPTER 1

# Overview

Clixon is a management framework that can be used by networking devices and other computer systems. Clixon provides a datastore, CLI, NETCONF and RESTCONF interfaces all defined by YANG.

Clixon links:

- Source code at github.
- Project.
- Docs.

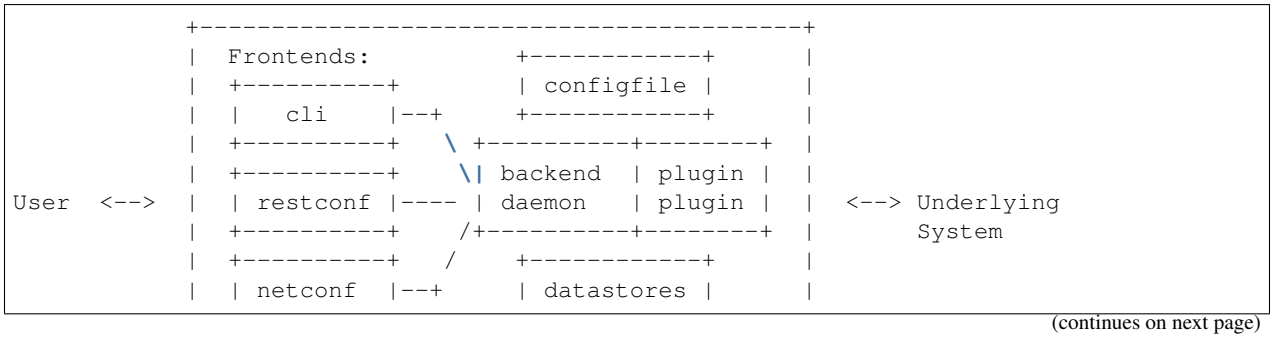
Most of the projects using Clixon are for networking devices. But Clixon can be used for other YANG-based systems as well due to a modular and pluggable architecture.

Clixon has a transaction mechanism that ensures configuration operations are atomic. It also features a generated interactive command-line interface using CLIgen.

The goal of Clixon is to provide a useful, production-grade, scalable and free YANG based configuration tool.

Clixon is open-source and dual licensed. Either Apache License, Version 2.0 or GNU General Public License Version 2.

## 1.1 System Architecture




(continues on next page)

(continued from previous page)

```
|  +----------+         +-------------+       |
+-----------------------------------------+
```

The core of the Clixon architecture is a backend daemon with configuration datastores and a set of internal clients: cli, restconf and netconf.

The clients provide frontend interfaces to users, including an interactive CLI, RESTCONF over HTTP, and XML NETCONF over SSH. Internally, the clients and backend communicate via NETCONF over a UNIX socket.

The backend manages a configuration datastore and implements a transaction mechanism for configuration operations (eg, create, read, update, delete) . The datastore supports candidate, running and startup configurations.

When you adapt Clixon to your system, you typically start with a set of YANG specifications that you want implemented. You then write backend plugins that interact with the underlying system. The plugins are written in C using the Clixon API and a set of plugin callbacks. The main callback is a transaction callback, where you specify how configuration changes are made to your system.

You can also design an interactive CLI using CLIgen, where you specify the CLI commands and write CLI plugins. You will have to write CLI rules, but Clixon can generate the configuration part of the CLI, including set, delete, show commands for a specific syntax.

## 1.2 Platforms

Clixon supports GNU/Linux, FreeBSD and Docker. MacOS may work. Linux platforms include 32 and 64 bits Ubuntu, Alpine, Raspian, etc.

## 1.3 Standards

Clixon supports standards including YANG, NETCONF, RESTCONF, XML and XPath. See *Standards* for more details.

## 1.4 How to get Clixon

Get the Clixon source code from Github

## 1.5 Support

For support issues use the Clixon slack channel

## 1.6 Bug reports

Report bugs via Github issues

CHAPTER 2

# Installation

Clixon runs on Linux, FreeBSD port and Mac/Apple. CPU architecures include x86_64, i686, ARM32.

You can also run Clixon in a docker container.

## 2.1 Ubuntu Linux

### 2.1.1 Installing dependencies

Install packages:

```
sudo apt-get update
sudo apt-get install flex bison fcgi-dev curl-dev
```

Install and build CLIgen:

```
git clone https://github.com/olofhagsand/cligen.git
cd cligen;
configure;
make;
make install
```

Add a clicon user and group, using useradd and usermod:

```
sudo useradd clicon
sudo usermod -a -G clicon <user>
sudo usermod -a -G clicon www-data
```

Or using adduser, with somewhat different syntax on different platforms.

### 2.1.2 Build from source

```
configure                  # Configure clixon to platform
make                       # Compile
sudo make install          # Install libs, binaries, and config-files
sudo make install-include # Install include files (for compiling)
```

## 2.2 FreeBSD

FreeBSD has ports for both cligen and clixon available. You can install them as binary packages, or you can build them in a ports source tree locally.

If you install using binary packages or build from the ports collection, the installation locations comply with FreeBSD standards and you have some assurance that the installed package is correct and functional.

The nginx setup for RESTCONF is altered - the system user www is used, and the restconf daemon is placed in /usr/local/sbin.

### 2.2.1 Binary package install

To install the pre-built binary package, use the FreeBSD pkg command:

```
% pkg install clixon
```

This will install clixon and all the dependencies needed.

### 2.2.2 Build from source

If you prefer you can also build clixon from the FreeBSD ports collection

Once you have the Ports Collection installed, you build clixon like this

```
% cd /usr/ports/devel/clixon
% make && make install
```

One issue with using the Ports Collection is that it may not install the latest version from GitHub. The port is generally updated soon after an official release, but there is still a lag between it and the master branch. The maintainer for the port tries to assure that the master branch will compile always, but no FreeBSD specific functional testing is done.

## 2.3 Systemd

Once installed, Clixon can be setup using systemd. The following shows an example with the backend and restconf daemons for the main example. Install them as /etc/systemd/system/example.service and /etc/systemd/system/example_retsconf.service, for example.

### 2.3.1 Systemd backend

The backend service is installed at /etc/systemd/system/example.service, for example. Note that in this example, the backend installation requires the restconf service, which is not necessary.

```
[Unit]
Description=Starts and stops a clixon example service on this system
Wants=example_restconf.service
[Service]
Type=forking
User=root
RestartSec=60
Restart=on-failure
ExecStart=/usr/local/sbin/clixon_backend -s running -f /usr/local/etc/example.xml
[Install]
WantedBy=multi-user.target
```

### 2.3.2 Systemd restconf

The Restconf service is installed at /etc/systemd/system/example_restconf.service, for example.

```
[Unit]
Description=Starts and stops an example clixon restconf service on this system
Wants=example.service
After=example.service
[Service]
Type=simple
User=www-data
WorkingDirectory=/www-data
Restart=on-failure
ExecStart=/www-data/clixon_restconf -f /usr/local/etc/example.xml
[Install]
WantedBy=multi-user.target
```

## 2.4 Advanced install

The Clixon *configure* script (generated by autoconf) includes several options apart from the standard ones.

**These include (standard options are omitted)**

**--enable-debug** Build with debug symbols, default: no

**--disable-optyangs** Include optional yang files in clixon install used for example and testing, default: no

**--enable-publish** Enable publish of notification streams using SSE and curl

**--with-cligen=dir** Use CLIGEN here

**--without-restconf** disable support for restconf

**--with-wwwuser=<user>** Set www user different from www-data

**--with-configfile=FILE** set default path to config file

**--with-libxml2** use gnome/libxml2 regex engine

**--with-yang-installdir=DIR** Install Clixon yang files here (default: ${prefix}/share/clixon)

**--with-opt-yang-installdir=DIR** Install optional yang files here (default: ${prefix}/share/clixon)

CHAPTER 3

# Quick start

This section describes how to run the Hello world example available in the soure code at: hello example.

Clixon is not a system in itself, it is a support system for an application. In this case, the "application" is hello world. The hello world application is very simple where the application semantics is completely described by a YANG specification and a CLI specification.

A more advanced application will have backend (and frontend) plugins to add application-specific semantics. This is not necessary in the hello world application.

## 3.1 Content

The hello example directory contains the following files:

***hello.xml*** The configuration file. See clixon-config.yang for the yang spec of hello.xml.

***clixon-hello@2019-04-17.yang*** The yang spec of the example.

***hello_cli.cli*** CLIgen specification.

***Makefile.in*** Example makefile where plugins are built and installed

***README.md*** This file

## 3.2 Compile and run

Before you start, go through the install instructions.

```
make && sudo make install
```

Start backend in the background:

```
sudo clixon_backend
```

Start cli:

```
clixon_cli
```

## 3.3 Using the CLI

The example CLI allows you to modify and view the data model using *set*, *delete* and *show* via generated code.

The following example shows how to add a very simple configuration *hello world* using the generated CLI. The config is added to the candidate database, shown, committed to running, and then deleted.

```
olof@vandal> clixon_cli
cli> set <?>
  hello
cli> set hello world
cli> show configuration
hello world;
cli> commit
cli> delete <?>
  all                   Delete whole candidate configuration
  hello
cli> delete hello
cli> show configuration
cli> commit
cli> quit
olof@vandal>
```

## 3.4 Netconf

Clixon also provides a Netconf interface. The following example starts a netconf client form the shell, adds the hello world config, commits it, and shows it:

```
olof@vandal> clixon_netconf -q
<rpc><edit-config><target><candidate/></target><config><hello xmlns="urn:example:hello
↪"><world/></hello></config></edit-config></rpc>]]>]]>
<rpc-reply><ok/></rpc-reply>]]>]]>
<rpc><commit/></rpc>]]>]]>
<rpc-reply><ok/></rpc-reply>]]>]]>
<rpc><get-config><source><running/></source></get-config></rpc>]]>]]>
<rpc-reply><data><hello xmlns="urn:example:hello"><world/></hello></data></rpc-reply>
↪]]>]]>
olof@vandal>
```

## 3.5 Restconf

Clixon also provides a Restconf interface. A reverse proxy needs to be configured.

For example, using nginx on an Ubuntu release: install, and edit config file */etc/nginx/sites-available/default*

```
server {
   ...
   location / {
      fastcgi_pass unix:/www-data/fastcgi_restconf.sock;
      include fastcgi_params;
   }
   # Enable this for restconf notification streams
   location /streams {
      fastcgi_pass unix:/www-data/fastcgi_restconf.sock;
      include fastcgi_params;
      proxy_http_version 1.1;
      proxy_set_header Connection "";
   }
}
```

Note that the second part is necessary only for notification streams.

Start nginx daemon

```
sudo /etc/init.d/nginx start
```

or using systemd:

```
sudo systemctl start nginx.service
```

Start the restconf daemon

```
sudo su -c "/www-data/clixon_restconf" -s /bin/sh www-data &
```

Start sending restconf commands (using Curl):

```
olof@vandal> curl -X POST http://localhost/restconf/data -d '{"clixon-hello:hello":{
↪"world":null}}'
olof@vandal> curl -X GET http://localhost/restconf/data
{
   "data": {
     "clixon-hello:hello": {
       "world": null
     }
   }
}
```

## 3.6 Run a container

You can run the hello example as a pre-built docker container, on a *x86_64* Linux. First, the container is started with the backend running:

```
docker run --rm --name hello -d clixon/clixon clixon_backend -Fs init
```

Then a CLI is started

```
$ sudo docker exec -it hello clixon_cli
cli> set ?
 hello
cli> set hello world
```

(continues on next page)

(continued from previous page)

```
cli> show configuration
hello world;
```

Or Netconf:

```
$ sudo docker exec -it clixon/clixon clixon_netconf
<rpc><get-config><source><candidate/></source></get-config></rpc>]]>]]>
<rpc-reply><data/></rpc-reply>]]>]]>
```

## 3.7 Next steps

The hello world example only has a Yang spec and a template CLI spec. For more advanced applications, customized backend, CLI, netconf and restconf code callbacks becomes necessary.

Further, you may want to add upgrade, RPC:s, state data, notification streams, authentication and authorization. The main example contains such capabilities.

CHAPTER 4

# Standards

## 4.1 YANG

YANG and XML is the heart of Clixon. Yang modules are used as a specification for handling XML configuration data. The YANG spec is used to generate an interactive CLI, netconf and restconf clients. It also specifies the format of the XML datastore.

The YANG standards that Clixon follows include:

- YANG 1.0 RFC 6020
- YANG 1.1 RFC 7950
- RFC 7895: YANG module library

However, the following YANG syntax modules are not implemented (reference to RFC7950 in parenthesis):

- deviation (7.20.3)
- action (7.15)
- augment in a uses sub-clause (7.17) (module-level augment is implemented)
- require-instance
- instance-identifier type (9.13)
- status (7.21.2)
- YIN (13)
- Yang extended XPath functions: re-match(), deref)(), derived-from(), derived-from-or-self(), enum-value(), bit-is-set() (10.2-10.6)
- Default values on leaf-lists (7.7.2)
- Lists without keys (non-config lists may lack keys)

### 4.1.1 Regular expressions

Clixon supports two regular expressions engines:

***Posix*** Posix is the default method, _translates_ XSD regexp:s to posix before matching with the standard Linux regex engine. This translation is not complete but can be considered "good-enough" for most yang use-cases. For reference, all standard Yang models have been tested.

***Libxml2*** Libxml2 uses the XSD regex engine. This is a complete XSD engine but you need to compile and link with libxml2 which may add overhead.

To use libxml2 in clixon you need enable libxml2 in both cligen and clixon:

```
./configure --with-libxml2 # both cligen and clixon
```

You then need to set the following configure option:

```
<CLICON_YANG_REGEXP>libxml2</CLICON_YANG_REGEXP>
```

## 4.2 XML and XPath

Clixon has its own implementation of XML and XPath. See more in the detailed API reference.

The XML-related standards include:

- XML 1.0. (DOCTYPE/ DTD not supported)
- Namespaces in XML 1.0
- XPath 1.0

The following XPath axes are supported:

- child,
- descendant,
- descendant_or_self,
- self
- parent

The following xpath axes are *not* supported: preceeding, preceeding_sibling, namespace, following_sibling, following, ancestor,ancestor_or_self, and attribute

## 4.3 NETCONF

Clixon implements the following NETCONF RFC:s:

- RFC 6241: NETCONF Configuration Protocol
- RFC 6242: Using the NETCONF Configuration Protocol over Secure Shell (SSH)
- RFC 6243: With-defaults Capability for NETCONF. Clixon implements "explicit" default handling, but does not implement the RFC.
- RFC 5277: NETCONF Event Notifications
- RFC 8341: Network Configuration Access Control Model

The following RFC6241 capabilities/features are hardcoded in Clixon:

- :candidate (RFC6241 8.3)
- :validate (RFC6241 8.6)
- :xpath (RFC6241 8.9)
- :notification (RFC5277)

The following features are optional and can be enabled by setting CLICON_FEATURE:

- :startup (RFC6241 8.7)

Clixon does *not* support the following NETCONF features:

- :url capability
- copy-config source config
- edit-config testopts
- edit-config erropts
- edit-config config-text
- edit-config operation

Further, in *get-config* filter expressions, the RFC6241 XPath Capability is preferred over default subtrees. This has two reasons:

1. XPath has better performance since the underlying system uses xpath, and subtree filtering is done after the complete tree is retreived.
2. Subtree filtering does not support namespaces yet.

## 4.4 RESTCONF

Clixon Restconf is a daemon based on FastCGI C-API. Instructions are available to run with NGINX. The implementation is based on RFC 8040: RESTCONF Protocol.

The following features of RFC8040 are supported:

- OPTIONS, HEAD, GET, POST, PUT, DELETE, PATCH
- stream notifications (Sec 6)
- query parameters: “insert”, “point”, “content”, “depth”, “start-time” and “stop-time”.
- Monitoring (Sec 9)

The following features are not implemented:

- ETag/Last-Modified
- Query parameters: “fields”, “filter”, “with-defaults”

## 4.5 JSON

Clixon implements JSON according to ECMA JSON Data Interchange Syntax and RFC 7951 JSON Encoding of Data Modeled with YANG.

CHAPTER 5

# Configuration

The clixon configuration file is an XML file modelled by YANG. By default, it is installed in */usr/local/etc/clixon.xml*. The YANG specification for the configuration is clixon config.yang. All Clixon processes (backend, cli, netconf, restconf) use the same config file, although some configuration options are not valid for all processes.

Please consult the YANG spec directly if you want detailed description of config options.

## 5.1 Example

The following is the configuration file of the *hello world* example:

```
<clixon-config xmlns="http://clicon.org/config">
  <CLICON_FEATURE>*:*</CLICON_FEATURE>
  <CLICON_CONFIGFILE>/usr/local/etc/example.xml</CLICON_CONFIGFILE>
  <CLICON_YANG_DIR>/usr/local/share/clixon</CLICON_YANG_DIR>
  <CLICON_YANG_MODULE_MAIN>clixon-hello</CLICON_YANG_MODULE_MAIN>
  <CLICON_CLI_MODE>hello</CLICON_CLI_MODE>
  <CLICON_CLISPEC_DIR>/usr/local/lib/hello/clispec</CLICON_CLISPEC_DIR>
  <CLICON_SOCK>/usr/local/var/hello.sock</CLICON_SOCK>
  <CLICON_BACKEND_PIDFILE>/usr/local/var/hello.pidfile</CLICON_BACKEND_PIDFILE>
  <CLICON_XMLDB_DIR>/usr/local/var/hello</CLICON_XMLDB_DIR>
  <CLICON_STARTUP_MODE>init</CLICON_STARTUP_MODE>
  <CLICON_MODULE_LIBRARY_RFC7895>false</CLICON_MODULE_LIBRARY_RFC7895>
</clixon-config>
```

## 5.2 Specification

Some options (of approximately 50) described below of clixon-config.yang are the following (descriptions are skipped):

```
container clixon-config {
    leaf CLICON_CONFIGFILE{
        type string;
    }
    leaf CLICON_YANG_MAIN_DIR {
        type string;
    }
    leaf-list CLICON_FEATURE {
       type string;
    }
    leaf CLICON_YANG_REGEXP {
        type regexp_mode;
        default posix;
    }
```

The option *CLICON_CONFIGFILE* is special, it must be available before the configuration file is found (see *Finding the configuration*), which means that the value in the file is a no-op.

## 5.3 Finding the configuration

There are several ways to change where Clixon finds it config file (FILE), in priority order:

1. Start a clixon program with the *-f <FILE>* option. For example:

   ```
   clixon_backend -f FILE
   ```

2. At install time, Use the *–with-configfile=FILE* option to configure:

   ```
   ./configure -f FILE
   ```

3. At install time: *./configure –with-sysconfig=<dir>* when configuring. Then FILE is *<dir>/clixon.xml*
4. At install time: *./configure –sysconfig=<dir>* when configuring. Then FILE is *<dir>/etc/clixon.xml*
5. If none of the above: FILE is */usr/local/etc/clixon.xml*

Note that the configure file itself may have the option *CLICON_CONFIGFILE* but due to bootstrapping reasons, its value is meaningless but can be useful for documentation purposes:

```
<CLICON_CONFIGFILE>/usr/local/etc/clixon.xml</CLICON_CONFIGFILE>
```

## 5.4 Runtime modification

You can modify clixon options at runtime by using the *-o* option to modify the configuration specified in the configuration file. For example, add *usr/local/share/ietf* to the list of directories where yang files are searched for:

```
clixon_cli -o CLICON_YANG_DIR=/usr/local/share/ietf
```

## 5.5 Features

*CLICON_FEATURE* is a list of values, describing how Clixon supports feature. Clixon have three hardcoded features:

- ietf-netconf:candidate (RFC6241 8.3)
- ietf-netconf:validate (RFC6241 8.6)
- ietf-netconf:xpath (RFC6241 8.9)

Supplying the *-o* option will add a value to the list (not replace existing). For example, if *-o CLICON_FEATURE=ietf-netconf:startup* is given at startup, the following options would be present in the configuration:

```
<CLICON_FEATURE>ietf-netconf:startup</CLICON_FEATURE>
<CLICON_FEATURE>*:*</CLICON_FEATURE>
```

(Which does not really make sense since **:** enables all features anayway.)

## 5.6 Finding YANG files

The example have two options for finding Yang files:

```
<CLICON_YANG_DIR>/usr/local/share/clixon</CLICON_YANG_DIR>
<CLICON_YANG_MODULE_MAIN>clixon-hello</CLICON_YANG_MODULE_MAIN>
```

which means that Yang files are searched for in */usr/local/share/clixon* and that the module *clixon-hello* should be loaded. Note:

- *clixon-hello.yang* must be present in */usr/local/share/clixon*
- Clixon itself may load several YANG files as part of the system startup, such as *clixon-config.yang*. These must all reside in the list of *CLICON_YANG_DIR*:s.
- When a Yang file is loaded, it may contain references to other Yang files (eg using *import* and *include*). They must also be found in the list of *CLICON_YANG_DIR*:s.

The following configuration file options control the loading of Yang files:

***CLICON_YANG_DIR*** A list of directories (yang dir path) where Clixon searches for module and submodules.

***CLICON_YANG_MAIN_FILE*** Load a specific Yang module given by a file.

***CLICON_YANG_MODULE_MAIN*** Specifies a single module to load. The module is searched for in the yang dir path.

***CLICON_YANG_MODULE_REVISION*** Specifies a revision to the main module.

***CLICON_YANG_MAIN_DIR*** Load all yang modules in this directory.

Note that the special *YANG_INSTALLDIR* autoconf configure option, by default */usr/local/share/clixon* should be included in the yang dir path for Clixon system files to be found.

You can combine the options, however, if there are different variants of the same module, more specific options override less specific. The precedence of the options are as follows:

- *CLICON_YANG_MAIN_FILE*
- *CLICON_YANG_MODULE_MAIN*
- *CLICON_YANG_MAIN_DIR*

Note that using *CLICON_YANG_MAIN_DIR* Clixon may find several files containing the same Yang module. Clixon will prefer the one without a revision date if such a file exists. If no file has a revision date, Clixon will prefer the newest.

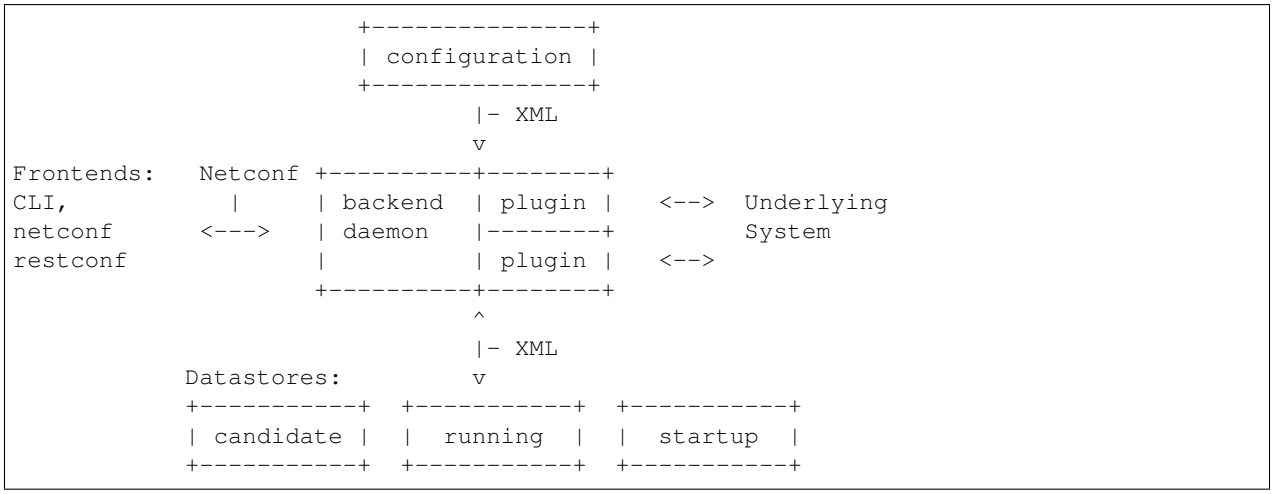
## 5.7 Default values

*CLICON_YANG_REGEXP* which is not present in the *hello world* is an example of a configuration option with a default value of *posix*:

```
<CLICON_YANG_REGEXP>posix</CLICON_YANG_REGEXP>
```

# CHAPTER 6

# Backend




The backend daemon is the central Clixon component. It consists of a main module and a number of dynamically loaded plugins. The backend has four APIs:

***configuration*** An XML file read at startup, possibly amended with *-o* options.

***Internal interface*** A NETCONF socket to frontend clients. This is by default a UNIX domain socket but can also be an IPv4 or IPv6 TCP socket.

***Datastores*** XML files storing configuration. The three main datastores are *candidate*, *running* and *startup*. A user edits the candidate datastore, commits the changes to running which triggers callbacks in the plugins.

***Application*** Backend plugins configure the underlying system with application-specific APIs. These API:s depend on how the underlying system is configured, examples include configuration files or a socket, for example.

Note that a user typically does not access the datastores directly, it is possible to read, but write operations should not be done, since the backend daemon uses a datastore cache.

## 6.1 Command-line options

**The backend have the following command-line options:**

| | |
|---|---|
| **-h** | Help |
| **-D <level>** | Debug level |
| **-f <file>** | CLICON config file |
| **-l <option>** | Log on (s)yslog, std(e)rr, std(o)ut or (f)ile. Syslog is default. If foreground, then syslog and stderr is default. Filename is given after -f: -lf<file>. |
| **-d <dir>** | Specify backend plugin directory (default: none) |
| **-p <dir>** | Yang directory path (see CLICON_YANG_DIR) |
| **-b <dir>** | Specify XMLDB database directory |
| **-F** | Run in foreground, do not run as daemon |
| **-z** | Kill other config daemon and exit |
| **-a <family>** | Internal backend socket family: UNIX\|IPv4\|IPv6 |
| **-u <path\|addr>** | Internal socket domain path or IP addr (see -a)(default: /usl/var/hello.sock) |
| **-P <file>** | PID filename (default: /usr/local/var/hello.pidfile) |
| **-1** | Run once and then quit (dont wait for events) |
| **-s <mode>** | Specify backend startup mode: none\|startup\|running\|init) |
| **-c <file>** | Load extra xml configuration, but don't commit. |
| **-g <group>** | Client membership required to this group (default: clicon) |
| **-U <user>** | Run backend daemon as this user AND drop privileges permanently |
| **-y <file>** | Load yang spec file (override yang main module) |
| **-o <option=value>** | Give configuration option overriding config file (see clixon-config.yang) |

### 6.1.1 Logging and debugging

At debug, the backend can be run in the foreground and with debug flags:

```
clixon_backend -FD 1
```

Logging is done on syslog. Alternatively, logging can be made on a file using the *-l* option:

```
clixon_backend -lf<file>
```

When run in foreground, logging is by default done on both syslog and stderr.

In a debugging mode, it can be useful to run in *once-only* mode, where the backend quits directly after starting up, instead of waiting for events:

```
clixon_backend -F1D 1
```

## 6.2 Startup

There are four different backend startup modes selected by the *-s* option. The difference is in how the running state is handled, ie what state the machine is in when you start the daemon and how loading the configuration affects it:

***none*** Do not touch running state. Typically after crash when running state and db are synched.

***init*** Initialize running state. Start with a completely clean running state.

***running*** Commit running db configuration into running state. Typically after reboot if a persistent running db exists.

***startup*** Commit startup configuration into running state. After reboot when no persistent running db exists.

You use the *-s* to select startup mode:

```
clixon_backend -s running
```

You may also add a default method in the configuration file:

```
<clixon-config xmlns="http://clicon.org/config">
  ...
  <CLICON_STARTUP_MODE>init</CLICON_STARTUP_MODE
</clixon-config>
```

When loading the startup/tmp configuration, the following actions are performed by the system:

- Check syntax errors,
- Upgrade callbacks.
- Validation of the XML against the current Yang models
- If errors are detected, enter *failsafe* mode.

## 6.3 Socket

```
Frontends:   socket  +----------+
CLI,           |     | backend  |
netconf      <--->   | daemon   |
restconf             |          |
                     +----------+
```

The Clixon backend creates a socket that the frontends can connect to. Communication is made over this socket using internal Netconf. The following config options are related to the internal socket:

**CLICON_SOCK_FAMILY** Address family for communicating with clixon_backend. One of: UNIX, IPv4, or IPv6. Can also be set with *-a* command-line option. Default is *UNIX* which denotes a UNIX socket.

**CLICON_SOCK** If family above is AF_UNIX: Unix socket for communicating with clixon_backend. If family is AF_INET: IPv4 address";

**CLICON_SOCK_PORT** Inet socket port for communicating with clixon_backend (only IPv4|IPv6). Default is port *4535*.

**CLICON_SOCK_GROUP** Group membership to access clixon_backend unix socket. Default is *clicon*.

## 6.4 Backend files

A couple of config options control files related to the backend, as follows:

**CLICON_BACKEND_DIR** Location of backend .so plugins. Load all *.so* plugins in this dir as backend plugins

**CLICON_BACKEND_REGEXP** Regexp of matching backend plugins in CLICON_BACKEND_DIR. default: **.so*

**CLICON_BACKEND_PIDFILE** Process-id file of backend daemon

## 6.5 Plugins

Backend plugins are the "glue" that binds the Clixon system to the underlying system. The backend invokes *callbacks* in the plugins when events occur.

Plugins are written in C as dynamically loaded modules (*.so* files). At startup, the backend daemon looks in the directory pointed to by the config option *CLICON_BACKEND_DIR*, and loads all files with *.so* suffixes from that dir in alphabetical order.

For example, to load all backend plugins from: */usr/local/lib/example/backend*:

```
<CLICON_BACKEND_DIR>/usr/local/lib/example/backend</CLICON_BACKEND_DIR>
```

You can filter which plugins to load by specifying a regular expression. For example, the following will only load backend plugins starting with "example":

```
<CLICON_BACKEND_REGEXP>^example*.so$</CLICON_BACKEND_REGEXP>
```

A plugin must have a init function called *clixon_plugin_init*. If this function does not exist, the backend will fail.

The backend calls *clixon_plugin_init* and expects it to return an API struct defining all callbacks. The init function may return *NULL* in which case the backend logs this and continues.

Once the plugin is loaded, it awaits callbacks from the backend.

### 6.5.1 Callbacks

The following callbacks are defined for backend plugins:

**init** Clixon plugin init function, called immediately after plugin is loaded into the backend. The name of the function must be called *clixon_plugin_init*. It returns a struct with the name of the plugin, and all other callback names.

**start** Called when application is "started", (almost) all initialization is complete and daemon is in the background. If daemon privileges are dropped (see *dropping privileges*) this callback is called *before* privileges are dropped.

**exit** Called just before plugin is unloaded

**extension** Called at parsing of yang modules containing an extension statement. A plugin may identify the extension by its name, and perform actions on the yang statement, such as transforming the yang in-memory. A callback is made for every statement, which means that several calls per extension can be made.

**reset** Reset system status

**upgrade** General-purpose upgrade called once when loading the startup datastore

**trans_begin, trans_validate, trans_complete, trans_commit, trans_revert, trans_end, trans_abort** Transaction callbacks which are invoked for two reasons: validation requests or commits. These callbacks are further described in *transactions* section.

## 6.6 Transactions

Clixon follows NETCONF in its validate and commit semantics. Using the CLI or another frontend, you edit the *candidate* configuration, which is first *validated* for consistency and then *committed* to the *running* configuration.

A clixon developer writes commit functions to incrementaly upgrade a system state based on configuration changes. Writing commit callbacks is the core functionality of a clixon system.

The netconf validation and commit operation is implemented in Clixon by a transaction mechanism, which ensures that user-written plugin callbacks are invoked atomically and revert on error. If you have two plugins, for example, a transaction sequence looks like the following:

```
Backend   Plugin1    Plugin2
|           |           |
+---------->+---------->+ begin
|           |           |
+---------->+---------->+ validate
|           |           |
+---------->+---------->+ commit
|           |           |
+---------->+---------->+ end
```

If an error occurs in the commit call of plugin2, for example, the transaction is aborted and the commit reverted:

```
Backend   Plugin1    Plugin2
|           |           |
+---------->+---------->+ begin
|           |           |
+---------->+---------->+ validate
|           |           |
+---------->+---->X    + commit error
|           |           |
+---------->+           + revert
|           |           |
+---------->+---------->+ abort
```

## 6.7 Upgrade

Clixon has several variant of update callbacks:

- General-purpose datastore upgrade.
- Module-specific manual upgrade
- Experimental automatic module upgrade

This section describes how a user can write upgrade callbacks for data modeled by outdated Yang models. The scenario is a Clixon system with a set of current yang models that loads a datastore with old or even obsolete data.

### 6.7.1 General-purpose

A plugin registers a *ca_datastore_upgrade* function which gets called once on startup. This upgrade should be seen as a generic wrapper function to basic repair or upgrade of existing datastores. The module-specific upgrade callbacks are more fine-grained.

The general-purpose upgrade callback is usable if module-state is not available, or actions such as the following need to be done in the whole datastore:

- Remove or rename nodes
- Rename namespaces
- Add nodes

A recommended method, as shown in the example, is to make a pattern matching using XPath and then perform actions on the resulting nodes.

Example:

```
static clixon_plugin_api api = {
   ...
   .ca_datastore_repair=example_upgrade
};

/*! General-purpose datastore upgrade callback called once on startup
 * @param[in] h    Clicon handle
 * @param[in] db   Name of datastore, eg "running", "startup" or "tmp"
 * @param[in] xt   XML tree. Upgrade this "in place"
 * @param[in] msd  Info on datastore module-state, if any
 */
int
example_upgrade(clicon_handle h,
                char         *db,
                cxobj        *xt,
                modstate_diff_t *msd)
{
    cxobj   **xvec = NULL;   /* vector of result nodes */
    size_t    xlen;
    cvec     *nsc = NULL;    /* Canonical namespace */
    int       i;

    /* Skip other than startup datastore */
    if (strcmp(db, "startup") != 0)
       return 0;
    /* Skip if there is proper module-state in datastore */
    if (msd->md_status)
       return 0;
    /* Get canonical namespaces for using "normalized" prefixes */
    if (xml_nsctx_yangspec(yspec, &nsc) < 0)
       err;
    /* Get all xml nodes matching path */
    if (xpath_vec(xt, nsc, "/a:x/a:y", &xvec, &xlen) < 0)
       err;
    /* Iterate throughnodes and remove them */
    for (i=0; i<xlen; i++){
       if (xml_purge(xvec[i]) < 0)
          err;
    }
}
```

The example above first checks whether it is the *startup* datastore and that it does not contain module-state. It then matches all nodes matching the XPath */a:x/a:y* using canonical prefixes, which are then deleted.

### 6.7.2 Module-state

Clixon can store Yang module-state information according to YANG Module library in the datastores. Including yang module-state in the datastores is enabled by the following entry in the Clixon configuration:

```
<CLICON_XMLDB_MODSTATE>true</CLICON_XMLDB_MODSTATE>
```

If the datastore does not contain module-state info, no module-specific upgrades can be made, only the general-purpose upgrade is available.

A backend does not perform detection of mismatching XML/Yang if:

1. The datastore was saved in a pre-3.10 system
2. *CLICON_XMLDB_MODSTATE* was not enabled when saving the file
3. The backend configuration does not have *CLICON_XMLDB_MODSTATE* enabled.

Note that the module-state detection is independent of the other steps of the startup operation: syntax errors, validation checks, failsafe mode, etc, are still made, even though module-state detection does not occur.

Note also that a backend with *CLICON_XMLDB_MODSTATE* disabled will silently ignore the module state.

Example of a (simplified) datastore with Yang module-state:

```
<config>
  <a1 xmlns="urn:example:a">some text</a1>
  <modules-state xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <module-set-id>42</module-set-id>
    <module>
      <name>A</name>
      <revision>2019-01-01</revision>
      <namespace>urn:example:a</namespace>
    </module>
  </modules-state>
</config>
```

Note that the module-state is not available to the user, the backend datastore handler strips the module-state info. It is only shown in the datastore itself.

### 6.7.3 Module-specific upgrade

Module-specific upgrade is only available if *module-state* is enabled.

If the module-state of the startup configuration does not match the module-state of the backend daemon, a set of module-specific upgrade callbacks are made. This allows a user to program upgrade funtions in the backend plugins to automatically upgrade the XML to the current version.

A user registers upgrade callbacks based on module and revision ranges. A user can register many callbacks, or choose wildcards. When an upgrade occurs, the callbacks will be called if they match the module and revision ranges registered.

Different strategies can be used for upgrade functions. One coarse-grained method is to register a single callback to handle all modules and all revisions.

A fine-grained method is to register a separate _stepwise_ upgrade callback per module and revision range that will be called in a series.

### 6.7.4 Registering a callback

A user registers upgrade callbacks in the backend *clixon_plugin_init()* function. The signature of upgrade callback is as follows:

```
upgrade_callback_register(h, cb, namespace, from, revision, arg);
```

where:

- *h* is the Clicon handle,
- *cb* is the name of the callback function,
- *namespace* defines a Yang module. NULL denotes all modules. Note that module *name* is not used (XML uses namespace, whereas JSON uses name, XML is more common).
- *from* is a revision date indicated an optional start date of the upgrade. This allows for defining a partial upgrade. It can also be *0* to denote any version.
- *revision* is the revision date "to" where the upgrade is made. It is either the same revision as the Clixon system module, or an older version. In the latter case, you can provide another upgrade callback to the most recent revision.
- *arg* is a user defined argument which can be passed to the callback.

One example of registering a "catch-all" upgrade:

```
upgrade_callback_register(h, xml_changelog_upgrade, NULL, 0, 0, NULL);
```

Another example are fine-grained stepwise upgrades of a single module [upgrade example](#example-upgrade):

```
upgrade_callback_register(h, upgrade_2016, "urn:example:interfaces",
                          20140508, 20160101, NULL);
upgrade_callback_register(h, upgrade_2018, "urn:example:interfaces",
                          20160101, 20180220, NULL);

   20140508       20160101       20180220
------+--------------+--------------+-------->
      upgrade_2016   upgrade_2018
```

In the latter case, the first callback upgrades from revision 2014-05-08 to 2016-01-01; while the second makes upgrades from 2016-01-01 to 2018-02-20. These are run in series.

### 6.7.5 Upgrade callback

When Clixon loads a startup datastore with outdated modules, the matching upgrade callbacks will be called.

Note the following:

- Upgrade callbacks _will_ _not_ be called for data that is up-to-date with the current system
- Upgrade callbacks _will_ _not_ be called if there is no module-state in the datastore, or if module-state support is disabled.
- Upgrade callbacks _will_ be called if the datastore contains a version of a module that is older than the module loaded in Clixon.
- Upgrade callbacks _will_ also be called if the datastore contains a version of a module that is not present in Clixon - an obsolete module.

Re-using the previous stepwise example, if a datastore is loaded based on revision 20140508 by a system supporting revision 2018-02-20, the following two callbacks are made:

```
upgrade_2016(h, <xml>, "urn:example:interfaces", 20140508, 20180220, NULL, cbret);
upgrade_2018(h, <xml>, "urn:example:interfaces", 20140508, 20180220, NULL, cbret);
```

Note that the example shown is a template for an upgrade function. It gets the nodes of an yang module given by *namespace* and the (outdated) *from* revision, and iterates through them.

If no action is made by the upgrade calback, and thus the XML is not upgraded, the next step is XML/Yang validation.

An out-dated XML may still pass validation and the system will go up in normal state.

However, if the validation fails, the backend will try to enter the failsafe mode so that the user may perform manual upgarding of the configuration.

### 6.7.6 Example upgrade

The example and shows the code for upgrading of an interface module. The example is inspired by the ietf-interfaces module that made a subset of the upgrades shown in the examples.

The code is split in two steps. The *upgrade_2016* callback does the following transforms:

- Move /if:interfaces-state/if:interface/if:admin-status to /if:interfaces/if:interface/
- Move /if:interfaces-state/if:interface/if:statistics to if:interfaces/if:interface/
- Rename /interfaces/interface/description to /interfaces/interface/descr

**The *upgrade_2018* callback does the following transforms:**

- Delete /if:interfaces-state
- Wrap /interfaces/interface/descr to /interfaces/interface/docs/descr
- Change type /interfaces/interface/statistics/in-octets to decimal64 and divide all values with 1000

Please consult the *upgrade_2016* and *upgrade_2018* functions in [the example](../example/example_backend.c) and [test](../test/test_upgrade_interfaces.sh) for more details.

### 6.7.7 Extra XML

If the Yang validation succeeds and the startup configuration has been committed to the running database, a user may add “extra” XML.

There are two ways to add extra XML to running database after start. Note that this XML is “merged” into running, not “committed”.

The first way is via a file. Assume you want to add this xml:

```
<config>
  <x xmlns="urn:example:clixon">extra</x>
</config>
```

You add this via the -c option:

```
clixon_backend ... -c extra.xml
```

The second way is by programming the plugin_reset() in the backend plugin. The example code contains an example on how to do this (see plugin_reset() in example_backend.c).

The extra-xml feature is not available if startup mode is *none*. It will also not occur in failsafe mode.

### 6.7.8 Failsafe mode

If the startup fails, the backend looks for a *failsafe* configuration in *CLICON_XMLDB_DIR/failsafe_db*. If such a config is not found, the backend terminates. In this mode, running and startup mode should be unchanged.

If the failsafe is found, the failsafe config is loaded and committed into the running db.

If the startup mode was *startup*, the *startup* database will contain syntax errors or invalidated XML.

If the startup mode was *running*, the the *tmp* database will contain syntax errors or invalidated XML.

### 6.7.9 Repair

If the system is in failsafe mode (or fails to start), a user can repair a broken configuration and then restart the backend. This can be done out-of-band by editing the startup db and then restarting clixon.

In some circumstances, it is also possible to repair the startup configuration on-line without restarting the backend. This section shows how to repair a startup datastore on-line.

However, on-line repair _cannot_ be made in the following circumstances:

- The broken configuration contains syntactic errors - the system cannot parse the XML.
- The startup mode is *running*. In this case, the broken config is in the *tmp* datastore that is not a recognized Netconf datastore, and has to be accessed out-of-band.
- Netconf must be used. Restconf cannot separately access the different datastores.

First, copy the (broken) startup config to candidate. This is necessary since you cannot make *edit-config* calls to the startup db:

```
<rpc>
  <copy-config>
    <source><startup/></source>
    <target><candidate/></target>
  </copy-config>
</rpc>
```

You can now edit the XML in candidate. However, there are some restrictions on the edit commands. For example, you cannot access invalid XML (eg that does not have a corresponding module) via the edit-config operation. For example, assume *x* is obsolete syntax, then this is _not_ accepted:

```
<rpc>
  <edit-config>
    <target><candidate/></target>
    <config>
      <x xmlns="example" operation='delete'/>
    </config>
  </edit-config>
</rpc>
```

Instead, assuming *y* is a valid syntax, the following operation is allowed since *x* is not explicitly accessed:

```
<rpc>
  <edit-config>
    <target><candidate/></target>
    <config operation='replace'>
      <y xmlns="example"/>
    </config>
  </edit-config>
</rpc>
```

Finally, the candidate is validate and committed:

```
<rpc>
  <commit/>
</rpc>
```

The example shown in this Section is also available as a regression [test script](../test/test_upgrade_repair.sh).

## 6.8 Privileges

The backend process itself does not really require any specific access, but it may be an important topic for an application using clixon when the plugins are designed. A plugin may need to access privileged system resources (such as configure files).

The backend itself is usually started as root: *sudo clixon_backend -s init*, which means that the plugins also run as root (being part of the same process).

The backend can also be started as a non-root user. However, you may need to set some config options to allow user write access, for example as follows(there may be others):

```
<CLICON_SOCK>/tmp/example.sock</CLICON_SOCK>
<CLICON_BACKEND_PIDFILE>/tmp/mytest/example.pid</CLICON_BACKEND_PIDFILE>
<CLICON_XMLDB_DIR>/tmp/mytest</CLICON_XMLDB_DIR>
```

### 6.8.1 Dropping privileges

You may want to start the backend as root and then drop privileges to a non-root user which is a common technique to limit exposure of exploits.

This can be done either by command line-options: *sudo clicon_backend -s init -U clicon* or (more generally) using configure options:

```
<CLICON_BACKEND_USER>clicon</CLICON_BACKEND_USER>
<CLICON_BACKEND_PRIVILEGES>drop_perm</CLICON_BACKEND_PRIVILEGES>
```

This will initialize resources as root and then *permanently* drop uid:s to the unprivileged user (*clicon* in the example abobe). It will also change ownership of several files to the user, including datastores and the clicon socket (if the socket is unix domain).

Note that the unprivileged user must exist on the system, see *Installation*.

### 6.8.2 Drop privileges temporary

If you drop privileges permanently, you need to access all privileged resources initially before the drop. For a plugin designer, this means that you need to access privileges system resources in the *plugin_init* or *plugin_start* callbacks.

The transaction callbacks, for example, will be run in unprivileged mode.

An alternative is to drop privileges temporary and the be able to raise privileges when needed:

```
<CLICON_BACKEND_USER>clicon</CLICON_BACKEND_USER>
<CLICON_BACKEND_PRIVILEGES>drop_temp</CLICON_BACKEND_PRIVILEGES>
```

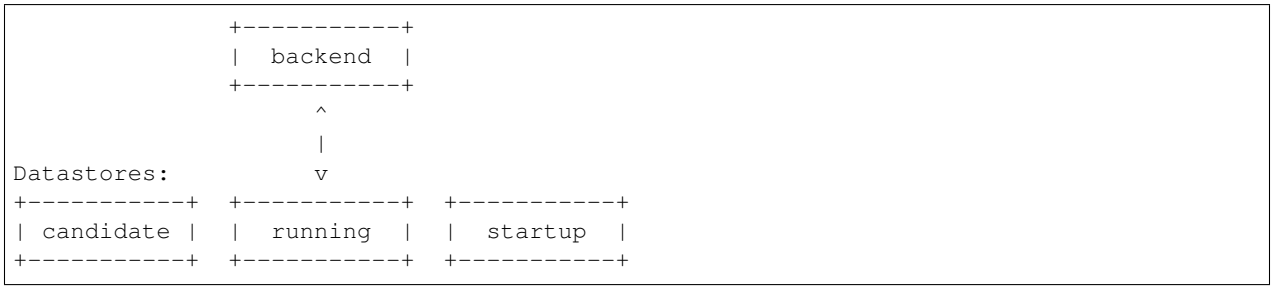
In this mode, a plugin callback (eg commit), can temporarily raise the privileges when accessing system resources, and the lower them when done.

An example C-code for raising privileges in a plugin is as follows:

```
uid_t euid = geteuid();
restore_priv();
... make high privilege stuff...
drop_priv_temp(euid);
```

CHAPTER 7

# Datastore




Clixon configuration datastores follow the Netconf model (from RFC 6241: NETCONF Configuration Protocol):

***Candidate*** A configuration datastore that can be manipulated without impacting the device's current configuration and that can be committed to the running configuration datastore.

***Running*** A configuration datastore holding the complete configuration currently active on the device.

***Startup*** The configuration datastore holding the configuration loaded by the device when it boots. Only present on devices that separate the startup configuration datastore from the running configuration datastore.

Note that there may appear other datastores, Clixon is not limited to the three datastores above. For example, a *tmp* datastore appears in several cases as an intermediate datastore.

## 7.1 Datastore files

The mandatory *CLICON_XMLDB_DIR* option determines where the datastores are placed. Example:

```
<CLICON_XMLDB_DIR>/usr/local/var/example</CLICON_XMLDB_DIR>
```

The permission of the datastores files is accessible to the user that starts the backend only. Typically this is *root*, but if the backend is started as a non-privileged user, or if privileges are dropped (see *Backend*) this may be another user, such as in the following example where *clicon* is used:

```
sh> ls -l /usr/local/var/example
-rwx------ 1 clicon clicon   0 sep 15 17:02 candidate_db
-rwx------ 1 clicon clicon   0 sep 15 17:02 running_db
-rwx------ 1 clicon clicon   0 sep 14 18:12 startup_db
```

Note that a user typically does not access the datastores directly, it is possible to read, but write operations should not be done, since the backend daemon may use a datastore cache, see *Datastore caching*.

## 7.2 Datastore format

By default, the datastore files use pretty-printed XML, with the top-symbol *config*. The following is an example of a valid datastore:

```
<config>
  <hello xmlns="urn:example:hello">
    <world/>
  </hello>
</config>
```

The format of the datastores can be changed using the following options:

***CLICON_XMLDB_FORMAT*** Datastore format. *xml* is the only fully supported alternative. *json* and *tree* are experimental (the latter is a random-access binary format).

***CLICON_XMLDB_PRETTY*** XMLDB datastore pretty print. The default value is *true*, which inserts spaces and line-feeds making the XML/JSON human readable. If false, the XML/JSON is more compact.

Note that the format settings applies to all datastores.

## 7.3 Module library support

Clixon can store Yang module-state information according to RFC 7895: YANG module library in the datastores. With module state, you know which Yang version the XML belongs to, and is useful when upgrading, for example.

Including yang module-state in the datastores is enabled by the following entry in the Clixon configuration:

```
<CLICON_XMLDB_MODSTATE>true</CLICON_XMLDB_MODSTATE>
```

A backend performs detection of mismatching XML/Yang if *CLICON_XMLDB_MODSTATE* was not enabled when saving the file so that it contains module-state information; and the backend configuration has *CLICON_XMLDB_MODSTATE* enabled.

Example of a (simplified) datastore with Yang module-state:

```
<config>
  <modules-state xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-library">
    <module-set-id>42</module-set-id>
    <module>
      <name>A</name>
      <revision>2019-01-01</revision>
      <namespace>urn:example:a</namespace>
    </module>
  </modules-state>
  <a1 xmlns="urn:example:a">some text</a1>
</config>
```

## 7.4 Datastore caching

Clixon datastore cache behaviour is controlled by the *CLICON_DATASTORE_CACHE* and can have the following values:

***nocache*** No cache, always read and write directly with datastore file.

***cache*** Use in-memory write-through cache. Make copies of the XML when accessing internally by callbacks and plugins. This is the default.

***cache-zerocopy*** Use in-memory write-through cache and do not copy when doing callbacks. This is the fastest but opens up for callbacks changing the cache. That is, plugin callbacks may not edit the XML in any way.

# CHAPTER 8

# CLI

The CLI uses https://github.com/olofhagsand/cligen is a central part of Clixon. CLIgen can stand on its own but is fully integrated into Clixon. This section describes the Clixon integration, for details on CLI syntax, etc, please consult the tutorial.

## 8.1 Using the CLI

Once the backend is started, the easiest way to use Clixon is via the CLI.

The following example shows how to add an interface in candidate, validate and commit it to running, then look at it (as xml) and finally delete it:

```
  clixon_cli -f /usr/local/etc/example.xml
  cli> set interfaces interface eth9 ?
    description               enabled                   ipv4
    ipv6                      link-up-down-trap-enable  type
  cli> set interfaces interface eth9 type ex:eth
  cli> validate
  cli> commit
  cli> show configuration xml
  <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
      <name>eth9</name>
      <type>ex:eth</type>
      <enabled>true</enabled>
    </interface>
  </interfaces>
cli> delete interfaces interface eth9
```

## 8.2 CLI specs and plugins

When defining a CLI frontend, there are two kinds of CLI specification files:

- *Clispecs*: CLI specification files (clispecs) written in CLIgen syntax.
- *Plugins*: Dynamic loadable plugin files loaded by *clixon_cli* at startup. Callbacks from clispec files are resolved and need to exist as symbols either in the CLixon libs or in the plugin file.

The following example shows examples of both files taken from the main example. First, a clispec file containing two commands:

```
show("Show state") config("Show configuration"), cli_show_config("candidate", "text",
↪"/");
example("Callback example") <var:int32>("any number"), mycallback("myarg");
```

In the CLI, these will generate CLI commands such as:

```
show config
example 23
```

The effect of typing the commands above will be of calling the callbacks: *cli_show_config* and *mycallback*. Both these functions must exist as C-functions. In fact, *cli_show_config* is a library function avaliable in the Clixon libs, while *mycallback* is defined in the main example CLI plugin.

In this way, a designer writes cli command specifications which invokes C-callbacks. If there are no appropriate callbacks the designer must write a new callback function.

The following config options are related to clispec and plugin files:

**CLICON_CLI_DIR** Directory containing frontend cli loadable plugins. Load all *.so* plugins in this directory as CLI object plugins.

**CLICON_CLISPEC_DIR** Directory containing frontend cligen spec files. Load all *.cli* files in this directory as CLI specification files.

**CLICON_CLISPEC_FILE** Specific frontend cligen spec file as alternative or complement to *CLICON_CLISPEC_DIR*. Also available as *-c* in clixon_cli.

## 8.3 Modes

The CLI can have different *modes* which is controlled by a config option and some internal clispec variables. The config option is:

**CLICON_CLI_MODE** Startup CLI mode. This should match a CLICON_MODE variable setting in one of the clispec files. Default is "base".

Inside the clispec files *CLICON_MODE* is used to specify to which modes the syntax in a specific file defines. For example, if you have major modes *configure* and *operation* you can have a file with commands for only that mode, or files with commands in both, (or in all).

First, lets add a single command in the configure mode:

```
CLICON_MODE="configure";
show configure;
```

Then add syntax to both modes:

```
CLICON_MODE="operation:configure";
show("Show") files("files");
```

Finally, add a command to all modes:

```
CLICON_MODE="*";
show("Show") all("all");
```

Note that CLI command trees are merged so that show commands in other files are shown together. Thus, for example, using the clispecs above the two modes will be three commands in total for the *configure* mode:

```
> clixon_cli -m configure
cli> show <TAB>
  all     routing      files
```

but only two commands in the *operation* mode:

```
> clixon_cli -m operation
cli> show <TAB>
  all      files
```

## 8.4 History

Clixon CLI supports persistent command history. There are two CLI history related configuration options:

**CLICON_CLI_HIST_FILE** The file containing the history, default value is: *~/.clixon_cli_history*

**CLICON_CLI_HIST_SIZE** Max number of history line, default value is 300.

**The design is similar to bash history but is simpler in some respects:**

- The CLI loads/saves its complete history to a file on entry and exit, respectively
- The size (number of lines) of the file is the same as the history in memory
- Only the latest session dumping its history will survive (bash merges multiple session history).

Further, tilde-expansion is supported and if history files are not found or lack appropriate access will not cause an exit but will be logged at debug level

### 8.4.1 Sub-trees

You use sub-trees using the tree operator @. Every mode gets assigned a tree which can be referenced as @*name*. This tree can be either on the top-level or as a sub-tree. For example, create a specific sub-tree that is used as sub-trees in other modes:

```
CLICON_MODE="subtree";
subcommand{
  a, a();
  b, b();
}
```

then access that subtree from other modes:

```
CLICON_MODE="configure";
main @subtree;
other @subtree,c();
```

The configure mode will now use the same subtree in two different commands. Additionally, in the *other* command, the callbacks will be overwritten by *c*. That is, if *other a*, or *other b* is called, callback function *c* will be invoked.

### 8.4.2 Generated config syntax

A special kind of sub-tree is the system-generated syntax tree. The config options for generated config tree is:

**CLICON_CLI_GENMODEL** If set, generate CLI specification for CLI completion of loaded Yang modules. This CLI tree can be accessed in CLI spec files using the tree reference syntax (eg @datamodel). Default 1.

**CLICON_CLI_MODEL_TREENAME** If set, CLI specs can reference the model syntax using this reference. Example: set @datamodel, cli_set();

**CLICON_CLI_GENMODEL_COMPLETION** Generate code for CLI completion of existing db symbols

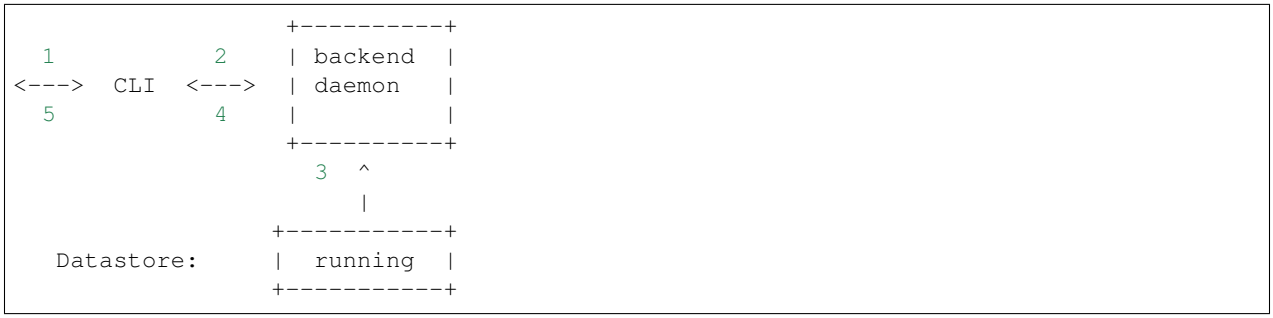
**CLICON_CLI_GENMODEL_TYPE** How to generate and show CLI syntax: VARS|ALL

CHAPTER 9

# Usecases

This section contains usescases which illustrate the flow of data from a user via Clixon frontends, backend to the underlying system and back.

## 9.1 CLI read




The first usecase illustrates how a retrieval of a configured value from the system is made.

1. The user makes a *show config* call using the hello world example(see *Quick start*). In the following examples uses *text* as modifier, and filters on *hello* top-level symbol:

```
cli> show configuration text hello
hello world;
```

2. The CLI string *show configuration text hello* is translated to internal NETCONF and sent to the backend:

```
<rpc username="myuser" xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source><candidate/></source>
    <nc:filter nc:type="xpath" nc:select="hello" xmlns="urn:example:hello"/>
  </get-config>
</rpc>
```

3. The backend receives the internal Netconf message, reads the *running* datastore and filters the output according to the XPath expression.

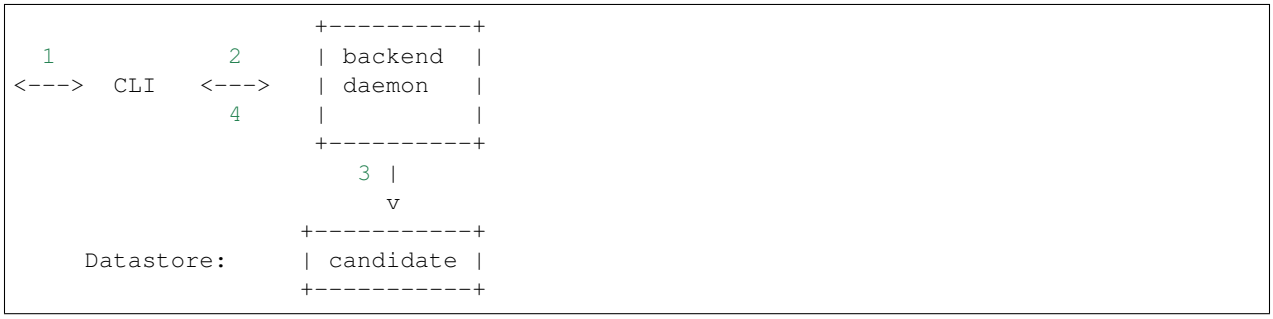
4. The backend returns the filtered output to the client:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <hello xmlns="urn:example:hello">
      <world/>
    </hello>
  </data>
</rpc-reply>
```

5. The CLI client translates the netconf to "text" output: *hello world;*

The user can also retrieve state data. Instead of reading from the running datastore, the backend reads state data either from a plugin, or from itself (if backend internal).

## 9.2 CLI write




The figure illustrates the way messages flow through the system. The numbers illustrate the enumeration below.

When setting a config value, the candidate datastore is modified and the commited to running which triggers a plugin commit transaction:

1. CLI example command:

```
cli> set hello world
cli>
```

2. Internal netconf containing a "replace" operation:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:nc=
↪"urn:ietf:params:xml:ns:netconf:base:1.0" username="clicon">
  <edit-config>
    <target><candidate/></target>
    <default-operation>none</default-operation>
    <config>
      <hello xmlns="urn:example:hello">
        <world nc:operation="replace"/>
      </hello>
    </config>
  </edit-config>
</rpc>
```

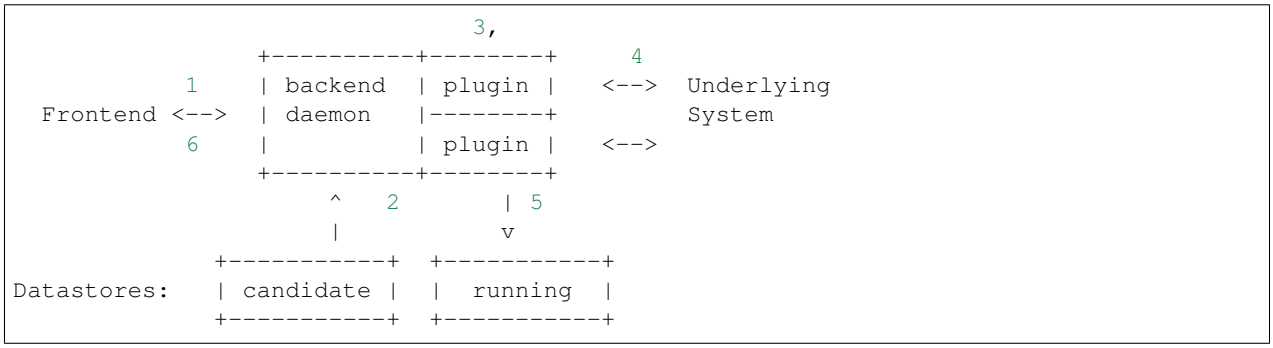
3. The backend modifies the *candidate* datastore. If there was no previous content it will look like the following after the edit:

```
<config>
  <hello xmlns="urn:example:hello">
    <world/>
  </hello>
</config>
```

4. The backend will reply with an OK:

```
<rpc-reply>
  <ok/>
</rpc-reply
```

## 9.3 Commit




After one, or several, edits, the user can commit the changes to running which triggers commit callbacks that will actually change the underlying system. Often, commits are made at once after every edit (such as RESTCONF operations). In that case, the edit described in the previous sections and commit are made in series by the client.

1. The client sends the commit message (frontend is not specified in this usecase):

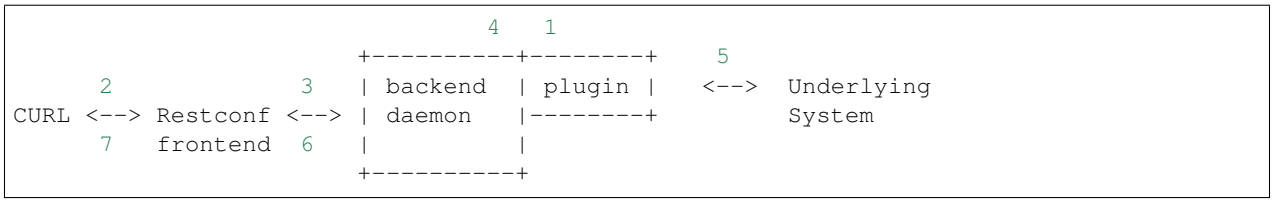
```
<rpc username="olof">
  <commit/>
</rpc>
```

2. When the backend receives the commit message, it computes the differences between candidate and running datastores, creates a transaction datastructure and initites a transaction.
3. Each plugin in turn gets callbacks to validate the transaction. The plugins verifies that the proposed changes to the system is sound. If not, the commit fails.
4. Each plugin in turn gets callbacks to commit the transaction to the underlying system. In this step, the application-dependent API:s are used to push the changes made.
5. If all validation and callbacks succeed, running is replaced with current

6. An OK is returned to the user.

```
<rpc-reply>
  <ok/>
</rpc-reply
```

## 9.4 RESTCONF RPC




A plugin can register an application-dependent RPC, and a client can then access it.

1. A plugin registers *example-rpc*:

```
rpc_callback_register(h, example_rpc, NULL, "urn:example:clixon", "example");
```

2. A user makes an RPC call, in this case RESTCONF:

```
curl -is -X POST -H "Content-Type: application/yang-data+json" -d '{"clixon-
↪example:input":{"x":0}}' http://localhost/restconf/operations/clixon-example:example
```

3. The restconf client receives the HTTP POST message (via a reverse proxy such as nginx) and translates the JSON to internal NETCONF:

```
<rpc username="none">
  <example xmlns="urn:example:clixon">
    <x>0</x>
  </example>
</rpc>
```

4. The backend receives the Netconf message and calls the registered callback *example_rpc()* in the plugin.
5. The plugin processes the rpc, for example by accessing state in the underlying system

6. The plugin returns a reply which is returned to the restonf client (for example):

```
<rpc-reply>
  <x xmlns="urn:example:clixon">0</x>
  <y xmlns="urn:example:clixon">42</y>
</rpc-reply>
```

7. The restconf client translates the Netconf message to JSON and returns to the client (via a reverse proxy):

```
{
  "clixon-example:output":{
     "x":"0",
     "y":"42"
  }
}
```

# CHAPTER 10

# Paths

Clixon uses paths to navigate in trees. Clixon uses the following three methods:

- *XML Path Language* defined in XPath 1.0 as part of XML and used in NETCONF.
- *Instance-identifier* defined in RFC 7950: The YANG 1.1 Data Modeling Language, a subset of XPath and used in NACM,.
- *Api-path* defined and used in RFC 8040: RESTCONF Protocol

All three use similar notations to find a path in tree, such as *a/b*, but differ in some ways.

## 10.1 XPath

Example of XPath in a NETCONF *get-config* RPC using the XPath capability:

```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
   <get-config>
      <source>
         <candidate/>
      </source>
      <filter type="xpath" select="/interfaces/interface[name='eth0']/description" />
   </get-config>
</rpc>
```

XPath is a powerful language for addressing parts of an XML document, including types and expressions. The following is a valid but complex XPath:

```
/assembly[name="robot_4"]//shape/name[containts(text(),'bolt')]/surface/roughness
```

Clixon uses XPaths extensively due to their expressive power. However, it is recommended to use instance-identifiers instead if you want optimized access.

### 10.1.1 Namespaces

XPath uses XML names, requiring an *XML namespace context* using the *xmlns* attribute to bind namespaces and prefixes. An XML namespace context can specify both:

- A default namespace for unprefixed names (*/x/*), defined by for example: *xmlns="urn:example:default"*.
- An explicit namespace for prefixed names prefix (*/ex:x/*), defined by for example: *xmlns:ex="urn:example:example"*.

Further, XML prefixes are *not inherited*, each symbol must be prefixed with a prefix or default. That is, */ex:x/y* is not the same as */ex:x/ex:y*, unless *ex* is also default.

Example: Assume an XML namespace context:

```
<a xmlns="urn:example:default" xmlns:ex="urn:example:example">
```

with an associated XPath:

```
/x/ex:y/ex:z[ex:i='w']`,
```

then symbol *x* belongs to "urn:example:default" and symbols *y*, *z* and *i* belong to "urn:example:example".

## 10.2 Instance-identifier

Instance-id:s are defined in YANG for some minor usage but appears in for example NACM and provides a useful subset of XPath. The subset is as follows (see Section 9.13 in YANG 1.1):

- Child paths using slashes: `/ex:system/ex:services`
- List entries for one or several keys: `/ex:system[ex:ip='192.0.2.1'][ex:port='80']`
- Leaf-list entries for one key: `/ex:system/ex:cipher[.='blowfish-cbc']`
- Position in lists: `/ex:stats/ex:port[3]`

Example of instance-id in NACM:

```
<path xmlns:acme="http://example.com/ns/itf">
      /acme:interfaces/acme:interface[acme:name='dummy']
</path>
```

Namespaces in instance-identifiers are the same as in XPaths.

## 10.3 Api-path

RESTCONF defines api-paths as a YANG-based path language. Keys are implicit which make path expressions more concise, but they are also less powerful

Example of Api-path in a restconf GET request:

```
curl -s -X GET http://localhost/restconf/data/ietf-interfaces:interfaces/
↪interface=eth0/description
```

Clixon uses Api-paths internally in some cases when accessing xml keys, but more commonly translates Api-paths to XPaths.

Api-path is in comparison to XPaths limited to pure path expressions such as, for example:

```
a/b=3,4/c
```

which corresponds to the XPath: *a[i=3][j=4]/c*. Note that you cannot express any other index variables than the YANG list keys.

### 10.3.1 Namespaces

In contrast to XPath, Api-path namespaces are defined implicitly by a YANG context using *module-names* as prefixes. The namespace is defined in the Yang module by the *namespace* keyword. Api-paths must have a Yang definition whereas XPaths can be compeletely defined in XML.

A prefix/module-name is *inherited*, such that a child inherits the prefix of a parent, and there are no defaults. For example, */moda:x/y* is the same as */moda:a/moda:y*.

Further, an Api-path uses a shorthand for defining list indexes. For example, */modx:z=w* denotes the element in a list of *z*:s whose key is the value *w*. This assumes that *z* is a Yang list (or leaf-list) and the index value is known.

Example: Assume two YANG modules *moda* and *modx* with namespaces "urn:example:default" and "urn:example:example" respectively, with the following Api-path (equivalent to the XPath example above):

```
/moda:x/modx:y/z=w
```

where, as above, *x* belongs to "urn:example:default" and *y*, and *z* belong to "urn:example:example".

CHAPTER 11

# XML tree

Clixon represents its internal data in an in "in-memory" tree representation. In the C API, this data structure is called *cxobj* and is used to represent config and state data. Typically, a cxobj is parsed from or printed to XML or JSON, but is really a generic representation of a tree.

## 11.1 Creating XML

### 11.1.1 Creating XML from a string

A simple way to create an cxobj is to parse it from a string:

```
cxobj *xt = NULL;
if (xml_parse_va(&xt, yt, "<x xmlns='urn:example:a'><k1>a</k2></x>") < 0)
   err;
```

where

- *xt* is a top-level cxobj containing the XML tree.
- *yt* is the yang spec corrresponding to *xt* (or NULL)
- The third parameter is a "printf" like format string followed by a variable argument list

If printed with for example: *xml_print(stdout,xt)* the tree looks as follows:

```
<top>
   <x xmlns="urn:example:a">
      <k1>a</k2>
   </x>
</top>
```

Note that a top-level node (*top*) is always created to encapsulate all trees parsed and that the default namespace in this example is "urn:example:a".

### 11.1.2 XML and YANG

If a yang specification (*yt*) is supplied to the parse function, the XML is bound to YANG. YANG binding is necessary for many Clixon functions including validation, optimized search, etc. But it is not mandatory.

For the XML in the example above, the YANG module could look something like:

```
module mod_a{
  prefix a;
  namespace "urn:example:a";
  list x{
    leaf k1{
      type string;
    }
  }
}
```

### 11.1.3 Creating JSON from a string

You can create an XML tree from JSON as well (note quotes are not escaped for clarity):

```
cxobj *xt = NUL;L
cxobj *xerr = NULL;

if ((ret = json_parse_str(&xt, yt, "{"mod_a:x":{"k1":"a"}}", &xerr)) < 0)
   err;
```

yielding the same xt tree as in *Creating XML from a string*.

There are some differences in comparison with the XML parse function:

- There is no variable argument format string
- Namespace prefixes in JSON use YANG module names, making the JSON format dependent on a correct YANG specification. This is similar to prefix handling in Api-path described in *Paths*.
- Error return is *-1* and OK is *1*
- If the return value of the function is *0*, the JSON is not validated correctly witrh respect to YANG, and an error message is returned in *xerr*.

### 11.1.4 Creating XML programmatically

You may also manually create a tree by *xml_new()*, *xml_addsub()* and other functions, this is more efficient than parsing but more work to program.

**Note:** expand on this.

### 11.1.5 Config data

To create a copy of configuration data, a user retrieve a copy from the datastore to get a cxobj handle. Read-only operations may then be done on the in-memory tree.

The following example code gets a copy of the whole *running* datastore to cxobj *xt*:

```
cxobj *xt = NULL;
if (xmldb_get(h, "running", NULL, NULL, &xt) < 0)
   err;
```

**Note:** In the case of config data, in-memory trees are read-only *caches* of the datastore and can normally not be written back to the datastore. Changes to the config datastire should be made via the backend netconf API, eg using *edit-config*.

## 11.2 Searching in XML

Clixon search indexes are either *implicitly* created from the YANG specification, or *explicitly* created using the API.

From YANG it is only *list* and *leaf-list* that are candidates for optimized lookup, direct *leaf* and *container* lookup is fast either way.

*Binary* search is used by search indexes and works by ordering list items alphabetically (or numerically), and then dividing the search interval in two equal parts depending on if the requested item is larger than, or less than, the middle of the interval.

Binary search complexity is *O(log N)*, whereas linear search is is *O(n)*. For example, a search in a vector of one million children will take up to *20* lookups, whereas linear search takes up to *1.000.000* lookups.

Therefore, if you have a large number of children and you need to make searches, it is important that you use indexes, either implicit, or explicit.

### 11.2.1 Implicit indexes

Implicit YANG-based search indexes are based on *list* and *leaf-lists*. For any list with keys *k1,...kn*, a set of indexes are created and an optimized search can be made using the keys in the order they are defined.

For example, assume the following YANG (this YANG is reused in later examples):

```
module mod_a{
  prefix a;
  namespace "urn:example:a";
  list x{
    key "k1 k2";
    leaf k1{
      type string;
    }
    leaf k2{
      type string;
    }
    leaf-list y{
      type string;
    }
    leaf z{
      type string;
    }
    ...
```

Assume also an example XML tree as follows:

```
<top xmlns="urn:example:a">
  <x>
    <k1>a</k1>
    <k2>a</k2>
    <y>cc</y>
    <y>dd</y>
    <z>foo</a>
  </x>
  <x>
    <k1>a</k1>
    <k2>b</k2>
    <y>cc</y>
    <y>dd</y>
    <z>bar</a>
  </x>
  <x>
    <k1>b</k1>
    ...
</top>
```

Then there will be two implicit search indexes created for all XML nodes *x* so that they can be accessed with *O(log N)* with e.g.:

- XPath or Instance-id: *x[k1="a"][k2="b"]*.
- Api-path: *x=a,b*.

If other search variables are used, such as: *x[z="foo"]* the time complexity will be *O(n)* since there is no explicit index for *z*. The same applies to using key variables in another order than they appear in the YANG specification, eg: *x[k2="b"][k1="a"]*.

A search index is also generated for leaf-lists, using *x* as the base node, the following searches are optimized:

- XPath or Instance-id: *y[.="bb"]*.
- Api-path: *y=bb*.

In the following cases, implicit indexes are *not* created:

- No YANG definition of the XML children exists. There are several use-cases. For example that YANG is not used or the tree is part of YANG *ANYXML*.
- The list represents *state* data
- The list is *ordered-by user* instead of the default YANG *ordered-by system*.

In those cases where implicit YANG indexes cannot be used, explicit indexes can be created for fast access.

### 11.2.2 Explicit indexes[1]

You can register explicit indexes using the function *clixon_index_register()*.

**Note:** *This section is not completed*

[1] Is planned for Clixon 4.4

### 11.2.3 Direct children

The basic C API for searching direct children of a cxobj is the *xml_find_index()* API.

An example call is as follows:

```
cxobj  **xvec = NULL;
size_t   xlen = 0;
cvec    *cvk = NULL; vector of index keys
... Populate cvk with key/values eg k1=a k2:b
if (clixon_xml_find_index(xp, yp, name, cvk, &xvec, &xlen) < 0)
   err;
/* Loop over found children*/
for (i = 0; i < xlen; i++) {
    x = xvec[i];
    ...
}
```

where

| | |
|---|---|
| *xp* | is an XML parent |
| *yp* | is the YANG specification of xp |
| *name* | is the name of the wanted children |
| *cvk* | is a vector of index name and value pairs |
| *xvec* | is a result vector of XML nodes. |

For example, using the previous XML tree and if *name=x* and *cvk* contains the single pair: *k1=a*, then *xvec* will contain both *x* entries after calling the function:

```
0: <x><k1>a</k1><k2>a</k2><y>cc</y><y>dd</y><z>foo</a></x>
1: <x><k1>a</k1><k2>b</k2><y>cc</y><y>dd</y><z>bar</a></x>
```

and the search was done using *O(logN)*.

### 11.2.4 Paths

If deeper searches are needed, i.e., not just to direct children, Clixon *Paths* can be used to make a search request. There are three path variants, each with its own pros and cons:

- XPath is most expressive, but only supports *O(logN)* search for YANG *list* entries (not leaf-lists), and adds overhead in terms of memory and cycles.
- Api-path is least expressive since it can only express YANG *list* and *leaf-list* key search.
- Instance-identifier can express all optimized searches as well as non-key searches. This is the recommended option.

Assume the same YANG as in the previous example, a path to find *y* entries with a specific value could be:

- XPath or instance-id: */a:x[a:k1="a"][a:k2="b"]/a:y[.="bb"]*
- Api-path: */mod_a:x=a,b/y=bb*

which results in the following result:

```
0: <y>bb</y>
```

An example call using instance-id:s is as follows:

```
cxobj **xvec = NULL;
size_t  xlen;
if (clixon_xml_find_instance_id(xt, yt, &xvec, &xlen,
       "/a:x[a:k1=\"a\"][k2=\"b\"]/a:y[.=\"bb\""") < 0)
   goto err;
for (i=0; i<xlen; i++){
   x = xvec[i];
      ...
}
```

The example shows the usage of implicit key indexes which makes this work in *O(logN)*, with the same exception rules as for direct children state in *Implicit indexes*.

An example call using api-path:s instead is as follows:

```
cxobj **xvec = NULL;
size_t  xlen;
if (clixon_xml_find_api_path(xt, yt, &xvec, &xlen, "/mod_a:x=a,b/y=bb") < 0)
   goto err;
for (i=0; i<xlen; i++){
   x = xvec[i];
      ...
}
```

The corresponding API for XPath is *xpath_vec()*.

### 11.2.5 Multiple keys

Optimized *O(logN)* lookup works with multiple key YANG *lists* but not for explicit indexes. Further, less significant keys can be omitted which may result multiple result nodes.

For example, the following lookups can be made using *O(logN)* using implicit indexes:

```
x[k1="a"][k2="b"]/y[.="cc"]
x[k1="a"]/y[.="cc"]
x[k1="a"][k2="b"]
```

The following lookups are made with *O(N)*:

```
x[k2="b"][k1="a"]
x[k1="a"][z="foo"]
```

## 11.3 Internal representation

A cxobj has several components, which are all accessible via the API. For example:

| | |
|---|---|
| name | Name of node |
| *prefix* | Optional prefix denoting a localname according to XML namespaces |
| *type* | A node is either an element, attribute or body text |
| *value* | Attributes and bodies may have values. |
| *children* | Elements may have a set of XML children |
| *spec* | A pointer to a YANG specification of this XML node |

The most basic way to traverse an cxobj tree is to linearly iterate over all children from a parent element node.

```
cxobj *x = NULL;
while ((x = xml_child_each(xt, x, CX_ELMNT)) != NULL) {
  ...
}
```

where *CX_ELMNT* selects element children (no attributes or body text).

However, it is recommended to use the *Searching in XML* for more efficient searching.

### 11.3.1 Footnotes

# CHAPTER 12

# Advanced

(Work in progress, these are sections that do not fit into the rest of the document)

## 12.1 CLI

### 12.1.1 Differences to CLIgen

Clixon adds some features and structure to CLIgen which include:

- A plugin framework for both textual CLI specifications(.cli) and object files (.so)
- Object files contains compiled C functions referenced by callbacks in the CLI specification. For example, in the cli spec command: *a,fn()*, *fn* must exist in the object file as a C function.
- The CLIgen *treename* syntax does not work.
- A CLI specification file is enhanced with the following CLIgen variables:
  - *CLICON_MODE*: A colon-separated list of CLIgen *modes*. The CLI spec in the file are added to _all_ modes specified in the list. You can also use wildcards * and *?*.
  - *CLICON_PROMPT*: A string describing the CLI prompt using a very simple format with: *%H*, *%U* and *%T*.
  - *CLICON_PLUGIN*: the name of the object file containing callbacks in this file.
- Clixon generates a command syntax from the Yang specification that can be referenced as *@datamodel*. This is useful if you do not want to hand-craft CLI syntax for configuration syntax.

Example of *@datamodel* syntax:

```
set    @datamodel, cli_set();
merge  @datamodel, cli_merge();
create @datamodel, cli_create();
show   @datamodel, cli_show_auto("running", "xml");
```

The commands (eg *cli_set*) will be called with the first argument an api-path to the referenced object.

### 12.1.2 How to deal with large specs

CLIgen is designed to handle large specifications in runtime, but it may be difficult to handle large specifications from a design perspective.

Here are some techniques and hints on how to reduce the complexity of large CLI specs:

#### Sub-modes

The *CLICON_MODE* is used to specify in which modes the syntax in a specific file should be added. For example, if you have major modes *configure* and *operation* you can have a file with commands for only that mode, or files with commands in both, (or in all).

First, lets add a basic set in each:

```
CLICON_MODE="configure";
show configure;
```

First, lets add a basic set in each:

```
CLICON_MODE="configure";
show configure;
```

Note that CLI command trees are *merged* so that show commands in other files are shown together. Thus, for example:

```
CLICON_MODE="operation:files";
show("Show") files("files");
```

will result in both commands in the operation mode:

```
> clixon_cli -m operation
cli> show <TAB>
  routing      files
```

but

```
> clixon_cli -m operation
cli> show <TAB>
  routing      files
```

#### Sub-trees

You can also use sub-trees and the the tree operator @. Every mode gets assigned a tree which can be referenced as @*name*. This tree can be either on the top-level or as a sub-tree. For example, create a specific sub-tree that is used as sub-trees in other modes:

```
CLICON_MODE="subtree";
subcommand{
  a, a();
  b, b();
}
```

then access that subtree from other modes:

```
CLICON_MODE="configure";
main @subtree;
other @subtree,c();
```

The configure mode will now use the same subtree in two different commands. Additionally, in the *other* command, the callbacks will be overwritten by *c*. That is, if *other a*, or *other b* is called, callback function *c* will be invoked.

### C-preprocessor

You can also add the C preprocessor as a first step. You can then define macros, include files, etc. Here is an example of a Makefile using cpp:

```
C_CPP     = clispec_example1.cpp clispec_example2.cpp
C_CLI     = $(C_CPP:.cpp=.cli
CLIS      = $(C_CLI)
all:      $(CLIS)
%.cli : %.cpp
     $(CPP) -P -x assembler-with-cpp $(INCLUDES) -o $@ $<
```