CBSD Part 1—**Production**

BY OLEG GINZBURG

In 2012, I worked as an IT Systems Administrator for Nevosoft, a small game developer that had the entire server infrastructure developed on the base of the FreeBSD OS. At that time, no one knew about Kubernetes and Docker, but by virtue of the FreeBSD Jail, the company's servers benefitted by having all the components separated and each service used an independent Jail container. Nowadays, FreeBSD boasts a dozen (or even more) container orchestration programs, although in 2012, there was not a wide choice. ezjail was available, but it was a solution for a Standalone Server. Our installation had thirty to forty physical servers, each of which launched from ten to twenty containers. Except for the primitive create actions, deleting, launching, and cloning containers, the company's system administrators needed more advanced capabilities, such as container management on remote servers, container migration from one server to another, and the ability to save a container as a portable image. Those results were achieved by creating simple shell scripts. In 2013, however, the necessity of applying the proprietary software on the servers caused the company to begin the migration process to Linux. Because by this time, all created shell scripts were on a par with ezjail in the sense of their capabilities, and in certain instances, they added unique features (for example, there was initially the TUI—text-based user interface), a decision was made to combine the scripts collection and publish them in the FreeBSD Ports Tree under the same title. This was the beginning of the CBSD Project.

Comparison with Other Management Systems

Today, FreeBSD supports at least thirty titles of utilities for managing containers and virtual machines. There is bhyve on the FreeBSD platform and from there the list grows steadily longer. 2022 marks CBSD's decennary—it is kept current and continues to expand. Thus far, this is one of the oldest virtual environment management systems on the FreeBSD platform. The virtual environments are not only intended to mean containerizing based on jail, but also support for virtual machines based on bhyve, XEN and QEMU / NVMM hypervisors.

The development was based on the following concepts and philosophy:

- Focus both on single-mode installations and on those consisting of multiple hosts;
- Have the opportunity to integrate with other solutions;
- Be open for changes and think in big-picture terms: the bigger idea, the larger the potential; whereas small ones usually do not lend themselves to growth;
- Remain simple and flexible in-use.

"Simplicity is the ultimate sophistication" – Leonardo da Vinci

A wide range of tool kits have sprung up around CBSD: web interface, message broker service for delivering tasks to distributed CBSD nodes, API service, and thin client for working with it.

Finally, CBSD is a native FreeBSD product, not a Linux solution porting attempt.

Learn more about the project objective: https://www.bsdstore.ru/en/cbsd_goals_ssi.html

Learn more about the project development: https://www.bsdstore.ru/en/cbsd_history_ssi.html

Child Projects

CBSD is not only a product for the end user, but also one of the elements involved in complex solutions building that can save a lot of person-hours by delegating functions for creating and managing CBSD virtual environments. As follows, there are various self-contained projects and distributions for demonstration and gaining insights into the re-use of CBSD work:

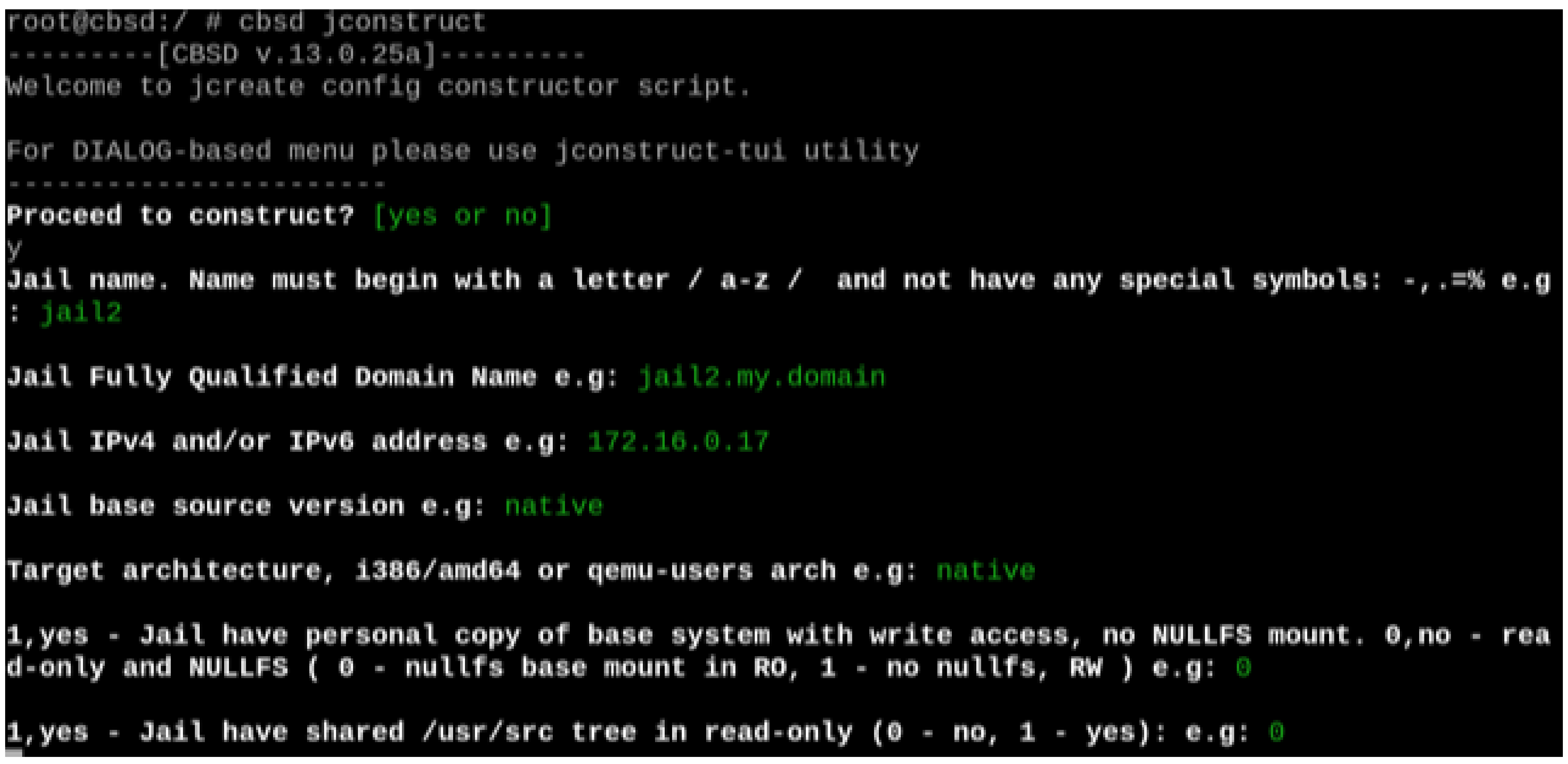
- **Reggae** (developed by Goran Mekić) uses CBSD for DevOps tasks automation;
- The distribution kit https://k8s-bhyve.convectix.com/ (**k8s-bhyve**) shows the ability to quickly (from a few seconds to 1 minute) launch Kubernetes Clusters set up on a bhyve hypervisors;
- The distribution kit https://clonos.convectix.com/ has the ability to construct a WEB/UI interface over CBSD for creating containers and virtual machines bhyve;
- The distribution kit https://myb.convectix.com/ (**MyBee**) enables work with cloud images through CBSD without using the UI and CLI: it is possible to get virtual machines having sent one request to the CBSD API using an HTTP request (for example, through the use of the curl utility) or using the nubectl thin client (https://github.com/bitcoin-software/nubectl).

CBSD and Jail Container: Practical Application

Let's get better acquainted with available CBSD jail operation methods. The site has a description of the initial CBSD installation and customization process, and we have assumed that you already have the run-time version. There are various ways to set up containers:

Option 1: in the command line dialog format. This method does not require learning a variety of possible jail parameters because the script will recall them:

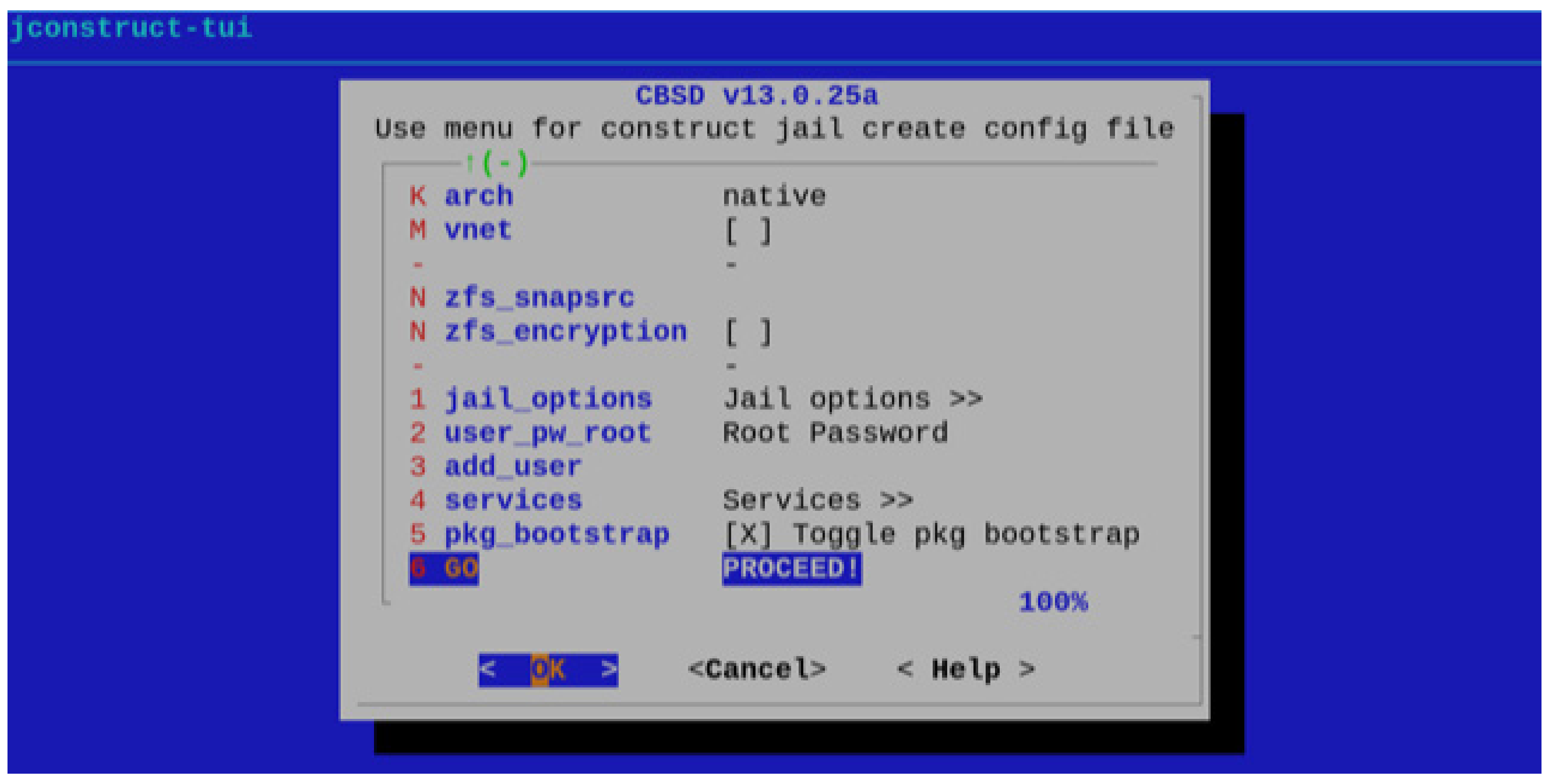
```
cbsd jconstruct
```



```
root@cbsd:/ # cbsd jconstruct
---------[CBSD v.13.0.25a]---------
Welcome to jcreate config constructor script.

For DIALOG-based menu please use jconstruct-tui utility
-----------------------
Proceed to construct? [yes or no]
y
Jail name. Name must begin with a letter / a-z /  and not have any special symbols: -,.=% e.g
: jail2

Jail Fully Qualified Domain Name e.g: jail2.my.domain

Jail IPv4 and/or IPv6 address e.g: 172.16.0.17

Jail base source version e.g: native

Target architecture, i386/amd64 or qemu-users arch e.g: native

1,yes - Jail have personal copy of base system with write access, no NULLFS mount. 0,no - rea
d-only and NULLFS ( 0 - nullfs base mount in RO, 1 - no nullfs, RW ) e.g: 0

1,yes - Jail have shared /usr/src tree in read-only (0 - no, 1 - yes): e.g: 0
```

Option 2: in dialogue format through TUI. This option is also appropriate for beginners, as—in line with the first option—knowledge of possible arguments and commands is not required:

```
cbsd jconstruct-tui
```



```
jconstruct-tui
                    CBSD v13.0.25a
    Use menu for construct jail create config file
     ↑(-)
     K arch              native
     M vnet              [ ]
     -                   -
     N zfs_snapsrc
     N zfs_encryption    [ ]
     -                   -
     1 jail_options      Jail options >>
     2 user_pw_root      Root Password
     3 add_user
     4 services          Services >>
     5 pkg_bootstrap     [X] Toggle pkg bootstrap
     6 GO                PROCEED!
                                      100%

          <  OK  >     <Cancel>    < Help >
```

The output of both dialogs is the text configuration file generation with the collection of parameters for the jcreate script, which you can call directly from the TUI interface or command line.

Option 3: Using a configuration file or command-line arguments.

Compared to the previous methods, this one is complicated in that it requires knowledge of the configurable parameters' names, but it is adequate for automation of environments development. You can synthesize a configuration file template as follows:

```
jname="myjail1"
ver=13.0
baserw=0
pkglist="misc/mc shells/bash"
astart=0
ip4_addr="DHCP"
...
```

Launch: `cbsd jcreate jconf=<path_to_file>`

Also, it is possible to specify the options as arguments without the configuration file:

```
cbsd jcreate jname=myjail1 ver=13.0 baserw=0 pkglist="misc/mc
shells/bash" astart=0 ip4_addr="DHCP"
```

The configuration file and arguments can be combined:

```
cbsd jcreate jconf=<path_to_file> ip4_addr="10.0.0.2" runasap=1
```

Option 4: Vagrant-like style: CBSDfile.

This is another notation for describing CBSD virtual environments allowing both: describe container settings and operate customization while creating (for example, copy files to the container file system and configure the service). Notwithstanding the fact that in one CBSD file you can describe an infinite number of environments, create and delete them by calling `cbsd up` and `cbsd destroy`, this form of notation is user friendly for a certain directory structures having one directory describing only one environment, for example: https://github.com/cbsd/cbsdfile-recipes/tree/master/jail

Also, the distinctive capability of environments created through the CBSD file format is not with local environments and through the CBSD API.

Jail Templates

We will not dwell upon the typical containers working operations, since they are described on the site. We'll go over some innovations of the CBSD project aimed to easily reference in jail—in particular, working with container templates. Templates are methods of the container description and configuration. As for the container, it can be exported to a portable image. A container can contain a standard runtime environment, but in most cases, they are used for services/application isolation and distribution through an image. There are advantages and disadvantages of this. Some of the advantages of the container approach are the speed of service deployment, no effect on the basic system environment, and the capability to commit the versions of dependencies with which your application is fully operational.

Referring to disadvantages of this approach, there is the issue of security, especially when using a container or image without a self-contained build script (template). This makes an operational software update to a container difficult (for example, to fix 0-day vulnerabilities) and it makes backdoors more likely to occur that are intentionally or accidentally left by image collectors. Considering the complexity of installing certain software, it is not unusual to find a container with a configured service hand-emplaced or with a lost assembly instruction.

Another disadvantage is the complexity of configuring services to the containers. Some images maintain the capability to configure a limited number of parameters through the environment variables, though not always sufficient. One of the examples is dynamic configurations, in the case of a need to add and configure multiple virtual hosts (vhost) for the WEB server and setup several databases in the DBMS. For such tasks, there is a specific software segment called configuration management. The best known products in this category are Ansible, Chef, Puppet, Rex, SaltStack. However, it is general practice that they are all optimized for work in classic environments. CBSD can combine the full power of configuration managers and a container-based approach to get containers with management services. There are two kinds of templates used in CBSD:

- static (classic), basically including install software only. An example of such a template for CBSD is the sambashare container.

Similar templates can be found in other projects: for instance, the nginx template through the example of Fockerfile (focker project):

```
base: freebsd-latest

steps:
  - run:
      - ASSUME_ALWAYS_YES=yes IGNORE_OSVERSION=yes pkg install nginx
```

Or here's an example of the RabbitMQ template for the BastilleBSD project, where the **CMD** file content is:

```
service rabbitmq restart
```

PKG file content:

```
rabbitmq
python27
```

These templates are of limited usefulness, as this is an alternate form of writing two lines in shell:

```
pkg install -y rabbitmq python27
service rabbitmq restart
```

There are more complex templates that cover the copies of the service configuration files and, in some cases, those followed by accompanied processing through complex structures from sed/awk while container configuration.

As a general principle:

a) This is an irreversible operation and designed for one-time operation: there's no way to roll back or change the configuration without re-creating the container;

b) such templates are highly demanding for their maintenance: even after a minor service update in the container, the source configuration file can be substantially changed;

c) there is no guarantee that the service will be operable, or, in the case of configurator error, that it will roll back to the previous run-time version;

d) it's a difficult line to draw between sed/awk operating and switching from it to use config management programs.

Here is the solution to the problem and the main CBSD project message: do not reinvent the wheel, but re-use someone else's work whenever you can. Many developers maintain configuration management modules, so we recommend that for reasons of saving time you use their work for delegating configuration with utilities that are native for this. With this objective in view, the forms script has been included in CBSD since 2016. The CBSD Forms functions are to get and save user parameters in the YAML format in the form appropriate for the configuration module—a peculiar kind of middleware. **Puppet** was chosen as the configuration management system, but any other framework can be used.

Preparing CBSD for Using Templates

To put things into perspective, let's create a jail1 container and apply the redis service template to it.

As it stands, it is expected that CBSD is installed and set up correctly: you can run a container and the cbsd dhcpd command gives a client the addresses that have access to the Internet (for example through NAT)--this is a necessary criterion for pkg operation and packages installation in the container.

1) Git must be installed for the CBSD plug-in installation from the public repository https://github.com/cbsd. Install git (~220 MB) or its light version - git-lite (~40 MB), if you have not yet done so:

```
pkg install -y git-lite
```

2) Create the system container cbsdpuppet1 from the profile included in the base distribution of CBSD. This is done once on each new CBSD host:

```
cbsd jcreate jname=cbsdpuppet1 jprofile=cbsdpuppet
```

The cbsdpuppet1 container should not be run, as it fulfils one function—it contains the puppet7 installed package used by CBSD to configure containers.

3) Install the CBSD puppet module—this is optional functionality and is not included in the basic distribution. The module contains the puppet modules checked by the CBSD project. This is done once on each new CBSD host:

```
cbsd module mode=install puppet
```

4) Install the module for the redis service configuration. This is done once on each new CBSD host:

```
cbsd module mode=install forms-redis
```

5) Create a container with any arbitrary name where we are about to get the redis service, for example: jail1:

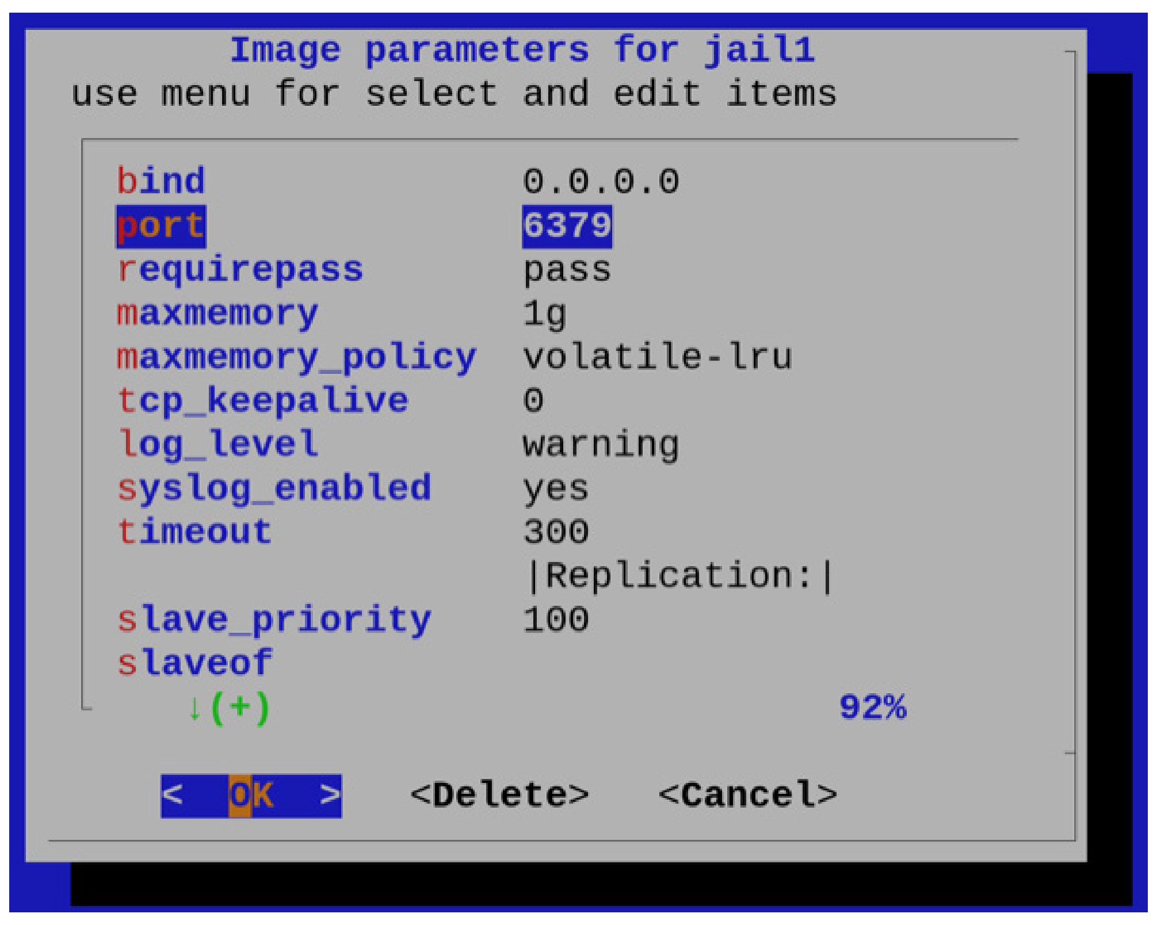
```
cbsd jcreate jname=jail1 runasap=1
```

(Note: the ***runasap=1*** *argument means the container will be run immediately).*

6) And the last step: pick up the redis service template to our jail1 container:

```
cbsd forms module=redis jname=jail1
```

There will be the familiar TUI interface with the parameters of the Redis module which can be configured at your discretion:



```
        Image parameters for jail1
  use menu for select and edit items

    bind                 0.0.0.0
    port                 6379
    requirepass          pass
    maxmemory            1g
    maxmemory_policy     volatile-lru
    tcp_keepalive        0
    log_level            warning
    syslog_enabled       yes
    timeout              300
                         |Replication:|
    slave_priority       100
    slaveof
       ↓(+)                          92%

     <  OK  >   <Delete>   <Cancel>
```

Let's retain the default arguments and pick them by the [COMMIT] operation. It may take a while before the script will finish running (depending on the Internet connection speed, since the module installs the redis server from the official repository `pkg.FreeBSD.org`) so you'll have to double check that the service in the container is installed and operating:

```
~ # cbsd jexec jname=jail1 sockstat -4l
USER     COMMAND    PID   FD PROTO  LOCAL ADDRESS         FOREIGN ADDRESS
redis    redis-serv 8910  7  tcp4   172.16.0.13:6379      *:*
```

By re-using the cbsd forms call, you can reconfigure the service at any time. Moreover, the forms hold the previous values and always output current data during initialization. For example, let's change an array of parameters:

- set the port to **7777**;
- set up a password to connect to redis server;
- set the maxmemory parameter to **4g**;
- set the maxmemory_policy parameter to **noeviction**.

Check on the state after the new parameter's application:

```
~ # cbsd jexec jname=jail1 sockstat -4l
USER     COMMAND    PID   FD PROTO  LOCAL ADDRESS         FOREIGN ADDRESS
redis    redis-serv 12587 7  tcp4   172.16.0.13:7777      *:*

~ # cbsd jexec jname=jail1 grep '^maxmemory' /usr/local/etc/redis.conf
maxmemory 4g
maxmemory-policy noeviction
```

Most Puppet modules commit to incorrect data validation, the configuration file validation, the service state. Therefore, the chance of getting a non-serviceable service due to invalid parameter input is considerably less than with statical templates.

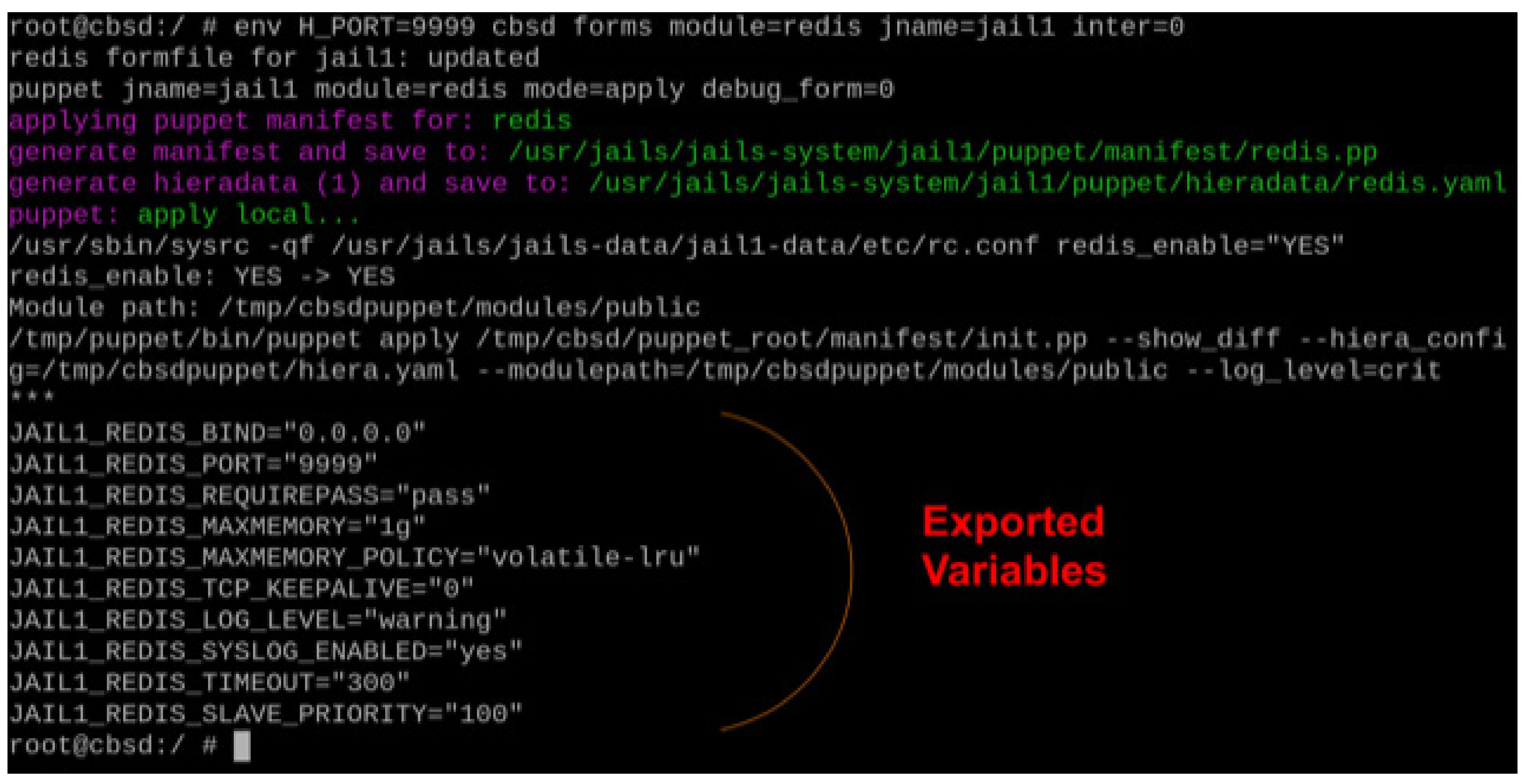
Now that we've gotten to know the TUI interface, let's turn to automation. The cbsd forms permit acceptance of the parameters for a template through environment variables dispensing with interactive dialogs. In order to see what variables a particular template accepts, use the **vars** argument with indication of the desired module:

```
~ # cbsd forms module=redis vars
H_BIND H_PORT H_REQUIREPASS H_MAXMEMORY H_MAXMEMORY_POLICY
H_TCP_KEEPALIVE H_LOG_LEVEL H_SYSLOG_ENABLED H_TIMEOUT H_SLAVE_PRIORITY
H_SLAVEOF
```

Forms add the **H_** prefix to the names of regular parameter to cut the likelihood of potential conflicts with global variables of the system. Let's reconfigure the redis port the third time (set it to the value of **9999**), but not in an interactive mode (inter=0):

```
env H_PORT=9999 cbsd forms module=redis jname=jail1 inter=0
```

Give attention to the output of values in applying the template:



```
root@cbsd:/ # env H_PORT=9999 cbsd forms module=redis jname=jail1 inter=0
redis formfile for jail1: updated
puppet jname=jail1 module=redis mode=apply debug_form=0
applying puppet manifest for: redis
generate manifest and save to: /usr/jails/jails-system/jail1/puppet/manifest/redis.pp
generate hieradata (1) and save to: /usr/jails/jails-system/jail1/puppet/hieradata/redis.yaml
puppet: apply local...
/usr/sbin/sysrc -qf /usr/jails/jails-data/jail1-data/etc/rc.conf redis_enable="YES"
redis_enable: YES -> YES
Module path: /tmp/cbsdpuppet/modules/public
/tmp/puppet/bin/puppet apply /tmp/cbsd/puppet_root/manifest/init.pp --show_diff --hiera_confi
g=/tmp/cbsdpuppet/hiera.yaml --modulepath=/tmp/cbsdpuppet/modules/public --log_level=crit
***
JAIL1_REDIS_BIND="0.0.0.0"
JAIL1_REDIS_PORT="9999"
JAIL1_REDIS_REQUIREPASS="pass"
JAIL1_REDIS_MAXMEMORY="1g"
JAIL1_REDIS_MAXMEMORY_POLICY="volatile-lru"
JAIL1_REDIS_TCP_KEEPALIVE="0"
JAIL1_REDIS_LOG_LEVEL="warning"
JAIL1_REDIS_SYSLOG_ENABLED="yes"
JAIL1_REDIS_TIMEOUT="300"
JAIL1_REDIS_SLAVE_PRIORITY="100"
root@cbsd:/ #
```

This is an informational output of the exported CBSD variables as a result of the module's work. The parameterization format can be set by mask through the configuration file `forms_export_vars.conf` in the directory `~cbsd/etc`.

Look into its default value:

https://github.com/cbsd/cbsd/blob/v13.0.18/etc/defaults/forms_export_vars.conf

These values can be automatically exported to a file or various Service Discovery services such as Consul. In this case, your cluster gets these values automatically, which can be used to build a **SOA** (Service Oriented Architecture), but that's a theme for another time.

Along with redis, here are other templates for services configuration: repositories named modules-forms-XXXX. For example, try to use these templates:

- modules-forms-memcached
- modules-forms-mysql
- modules-forms-grafana
- modules-forms-rabbitmq
- modules-forms-postgresql
- modules-forms-elasticsearch

Note: regardless of the conventional 1 service-1 container concept, in regard to the CBSD forms you can apply multiple templates to the same environment, having gotten all services in it at once.

How It Works

The Puppet modules (or any other similar system) contain the parameters that in 99% of cases have the value parameter form and are usually parameterized in YAML format. For user convenience, CBSD offers a TUI interface for working with basic forms, where, in a similar manner to HTML forms, the following elements can exist:

- radio button (the boolean type's values are true/false, yes/no);
- checkbox elements;
- dropdown menu of known certain values;
- the custom inputbox for relative user input;

To hold parameters in a universal form, CBSD forms uses the SQLite3 database—a describing form for the parameters input which stores the input value. The recording (writing) format is universal and serves for both TUI dialogs autogeneration (CBSD forms) and WEB/HTML forms (ClonOS), as the following examples clearly show. Let's create a SQLite3 database using the format required by CBSD forms:

```
sqlite3 /tmp/myforms.sqlite <<EOF
BEGIN TRANSACTION;
CREATE TABLE forms (  idx INTEGER PRIMARY KEY AUTOINCREMENT, mytable
VARCHAR(255) DEFAULT '', group_id INTEGER DEFAULT 1, order_id INTEGER
DEFAULT 1, param TEXT DEFAULT NULL, desc TEXT DEFAULT NULL, def TEXT
DEFAULT NULL, cur TEXT DEFAULT NULL, new TEXT DEFAULT NULL, mandatory
INTEGER DEFAULT 0, attr TEXT DEFAULT NULL, xattr TEXT DEFAULT NULL, type
VARCHAR(255) DEFAULT 'inputbox', link VARCHAR(255) DEFAULT '', groupname
VARCHAR(128) DEFAULT ''  );

INSERT INTO forms (
mytable,group_id,order_id,param,desc,def,cur,new,mandatory,attr,type,link,
groupname ) VALUES ( "forms", 1,0,"-","BSDMag poll: favorite BSD stuff
on",'-','','',1, "maxlen=128", "delimer", "", "" );
INSERT INTO forms (
mytable,group_id,order_id,param,desc,def,cur,new,mandatory,attr,type,link,
groupname ) VALUES ( "forms", 1,1,"FreeBSD","enter favorite FreeBSD tools,
e.g: sysrc",'bsdinstall','','',1, "maxlen=60", "inputbox", "", "" );
INSERT INTO forms (
mytable,group_id,order_id,param,desc,def,cur,new,mandatory,attr,type,link,
groupname ) VALUES ( "forms", 1,2,"DragonFlyBSD","enter favorite DFLY
tools. e.g: checkpoint",'checkpoint','','',1, "maxlen=60", "inputbox",
 "", "" );

CREATE TABLE system (  helpername TEXT DEFAULT NULL, helperdesc TEXT
DEFAULT NULL, version INTEGER DEFAULT 0, packages INTEGER DEFAULT 0,
have_restart INTEGER DEFAULT 0, longdesc TEXT DEFAULT NULL, title TEXT
DEFAULT ''  );
INSERT INTO system VALUES('bsdmag',NULL,201607,'','','','');
COMMIT;
EOF
```

And let's talk more about some of these lines. For us, the most important table columns are:

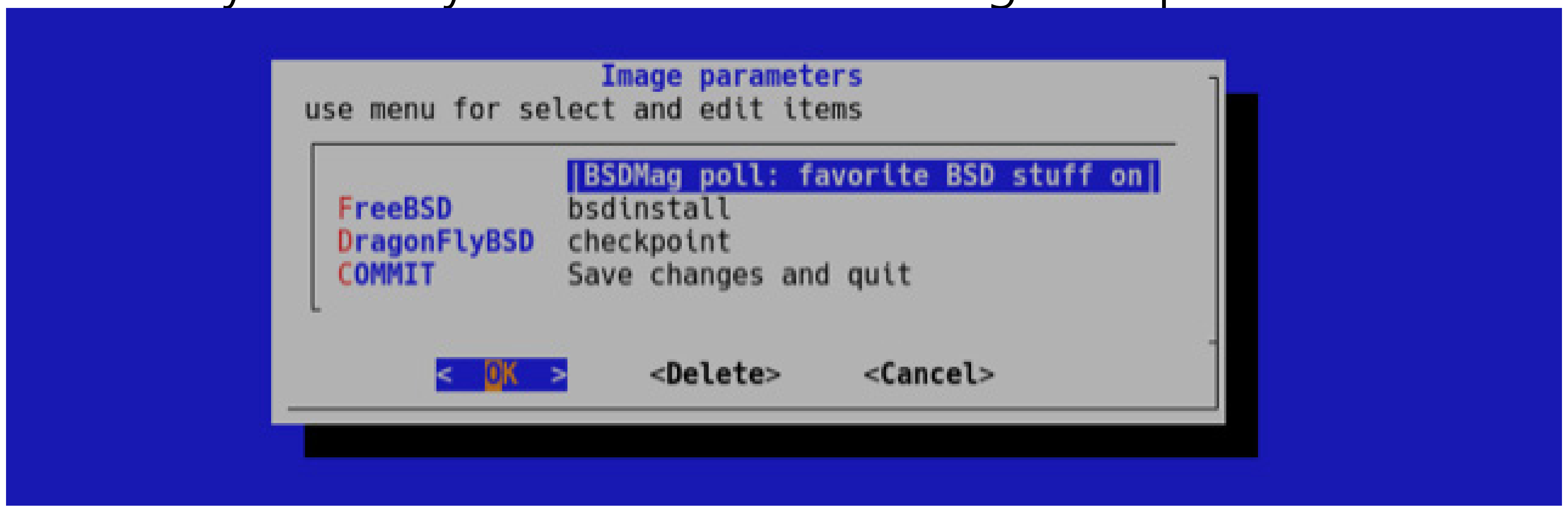
* **param** – the parameter-name directly, the value of which we want to get;
* **desc** – its relevant description, where appropriate;
* **def** – the default value;
* **new** – new value inputted by the user;
* **type** – type of a field: inputbox, delimiter, password, group_add, group_del, radio, checkbox.

Two parameters can be added in the next two lines, and the values we want to get are: FreeBSD and DragonFlyBSD, each having its own default value and both have the inputbox field format, so users are free to enter an arbitrary string value.

Run the cbsd forms script on this form:

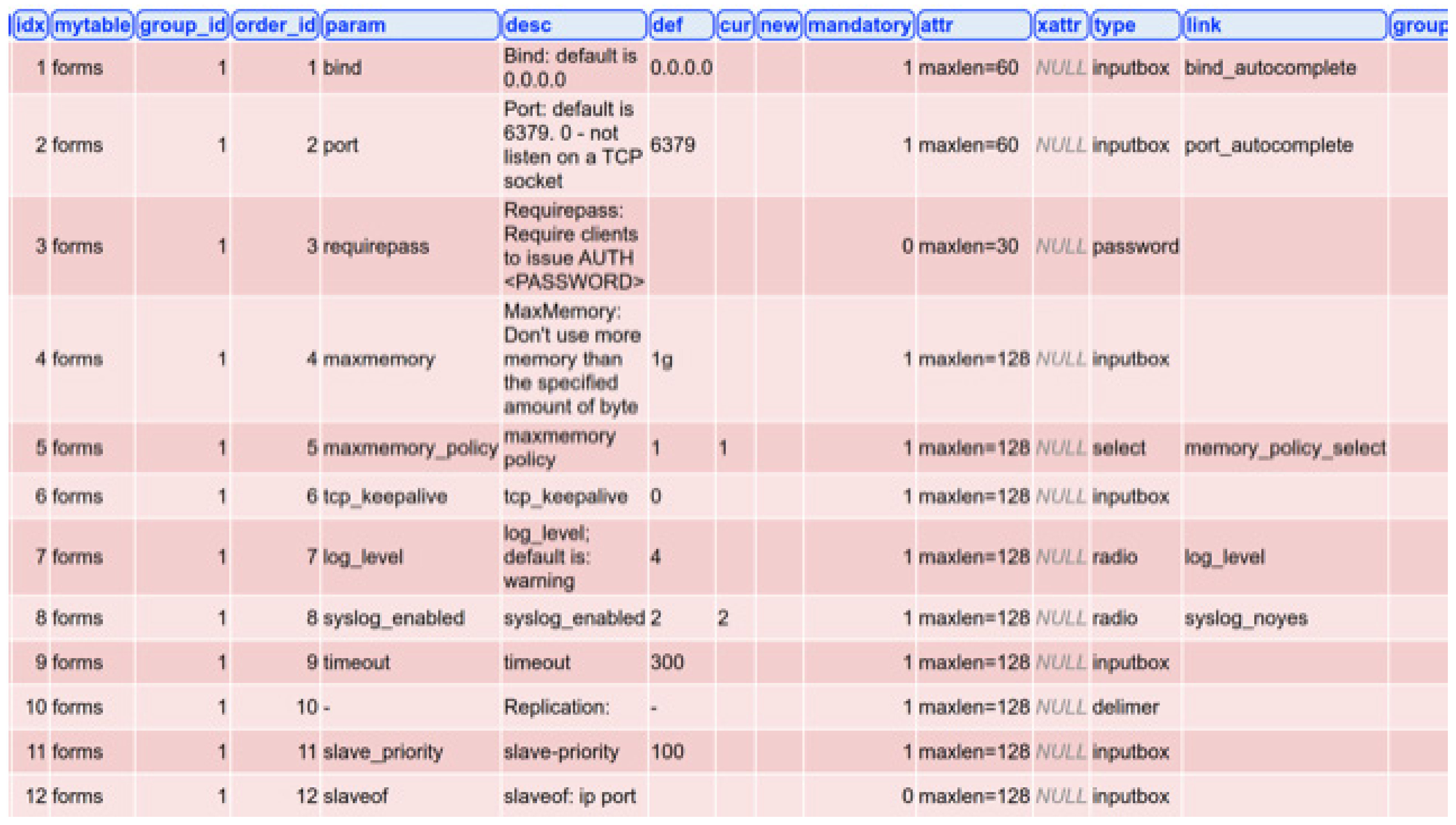
```
cbsd forms formfile=/tmp/myforms.sqlite
```

The output will be a dynamically built form for working with parameters:



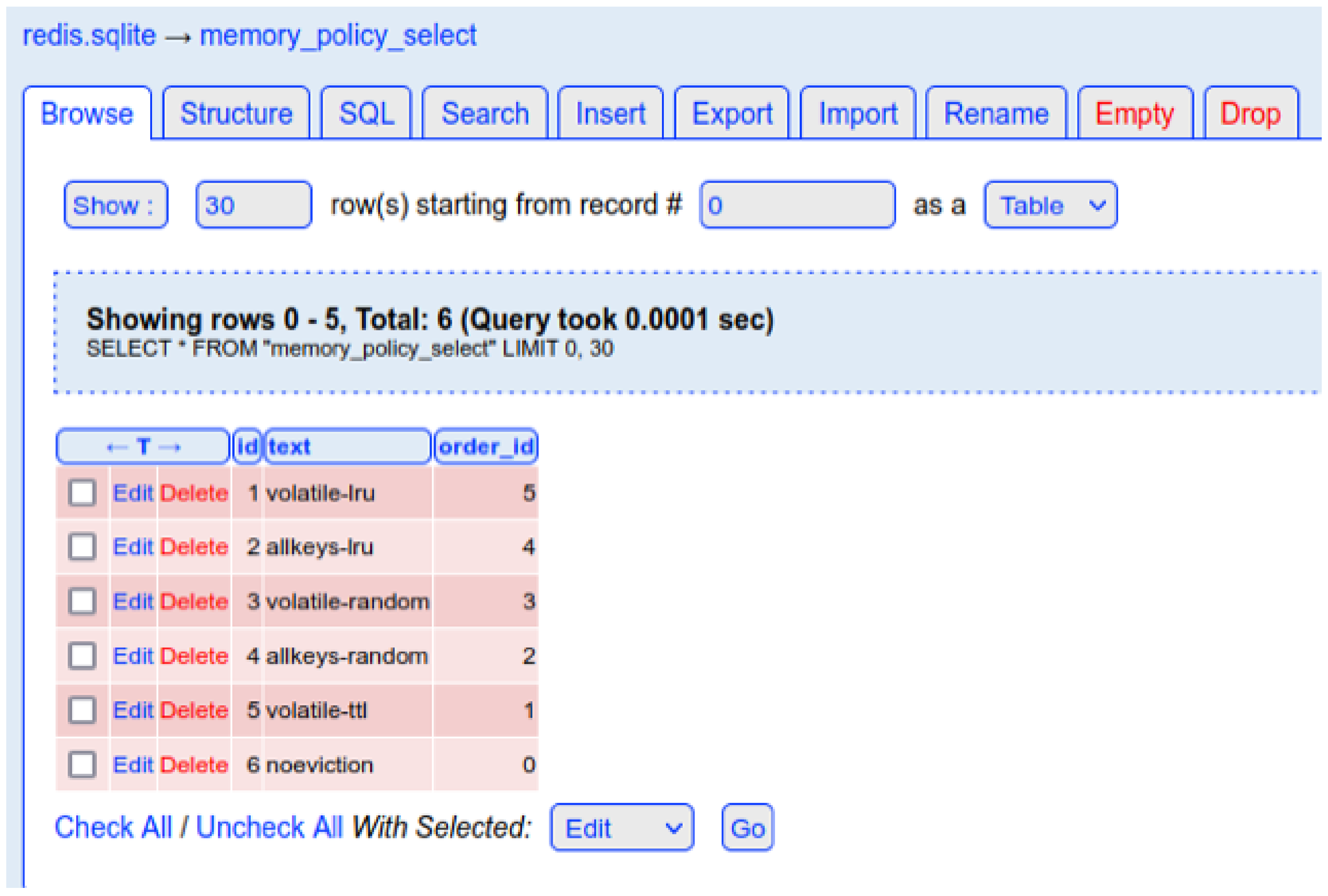
```
                     Image parameters
 use menu for select and edit items

                      |BSDMag poll: favorite BSD stuff on|
   FreeBSD            bsdinstall
   DragonFlyBSD       checkpoint
   COMMIT             Save changes and quit

              <  OK  >      <Delete>      <Cancel>
```

The form structure for the redis service is more complex as it contains the field elements of boolean and dropdown types. Let us look at the table:



idx	mytable	group_id	order_id	param	desc	def	cur	new	mandatory	attr	xattr	type	link	group
1	forms	1	1	bind	Bind: default is 0.0.0.0	0.0.0.0			1	maxlen=60	NULL	inputbox	bind_autocomplete	
2	forms	1	2	port	Port: default is 6379. 0 - not listen on a TCP socket	6379			1	maxlen=60	NULL	inputbox	port_autocomplete	
3	forms	1	3	requirepass	Requirepass: Require clients to issue AUTH <PASSWORD>				0	maxlen=30	NULL	password		
4	forms	1	4	maxmemory	MaxMemory: Don't use more memory than the specified amount of byte	1g			1	maxlen=128	NULL	inputbox		
5	forms	1	5	maxmemory_policy	maxmemory policy	1	1		1	maxlen=128	NULL	select	memory_policy_select	
6	forms	1	6	tcp_keepalive	tcp_keepalive	0			1	maxlen=128	NULL	inputbox		
7	forms	1	7	log_level	log_level; default is: warning	4			1	maxlen=128	NULL	radio	log_level	
8	forms	1	8	syslog_enabled	syslog_enabled	2	2		1	maxlen=128	NULL	radio	syslog_noyes	
9	forms	1	9	timeout	timeout	300			1	maxlen=128	NULL	inputbox		
10	forms	1	10	-	Replication:	-			1	maxlen=128	NULL	delimer		
11	forms	1	11	slave_priority	slave-priority	100			1	maxlen=128	NULL	inputbox		
12	forms	1	12	slaveof	slaveof: ip port				0	maxlen=128	NULL	inputbox		

We take as our example the table memory_policy_select, the `memory_policy parameter` variation table of the select type:



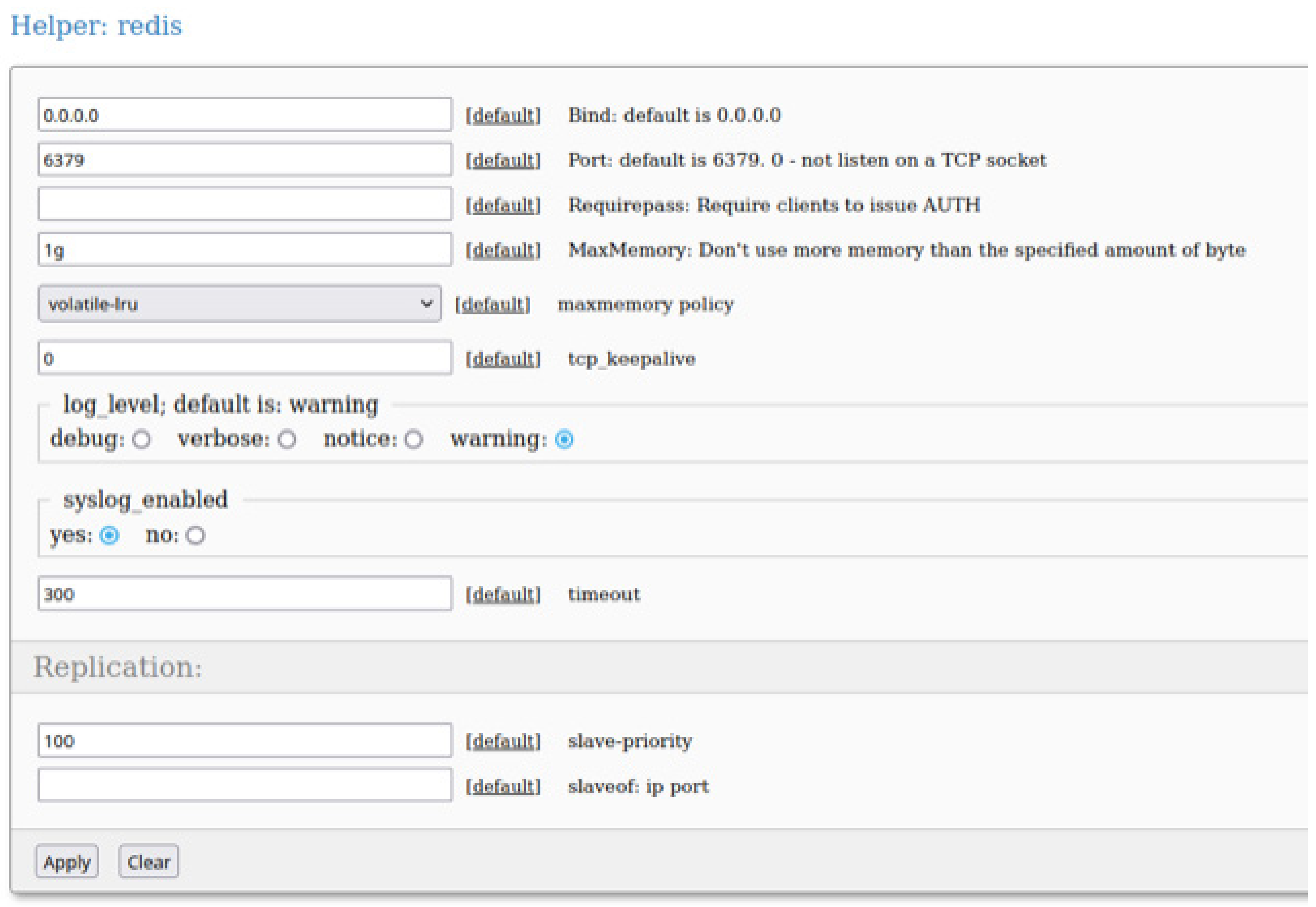
redis.sqlite → memory_policy_select

Show : 30 row(s) starting from record # 0 as a Table

Showing rows 0 - 5, Total: 6 (Query took 0.0001 sec)
SELECT * FROM "memory_policy_select" LIMIT 0, 30

← T →	id	text	order_id
Edit Delete	1	volatile-lru	5
Edit Delete	2	allkeys-lru	4
Edit Delete	3	volatile-random	3
Edit Delete	4	allkeys-random	2
Edit Delete	5	volatile-ttl	1
Edit Delete	6	noeviction	0

In the WEB interface, the auto-generated form (the screenshot was taken from ClonOS) looks like this:



Helper: redis

0.0.0.0 [default] Bind: default is 0.0.0.0
6379 [default] Port: default is 6379. 0 - not listen on a TCP socket
[default] Requirepass: Require clients to issue AUTH
1g [default] MaxMemory: Don't use more memory than the specified amount of byte
volatile-lru [default] maxmemory policy
0 [default] tcp_keepalive

log_level; default is: warning
debug: ○ verbose: ○ notice: ○ [✓ warning: ●]

syslog_enabled
[✓ yes: ●] no: ○

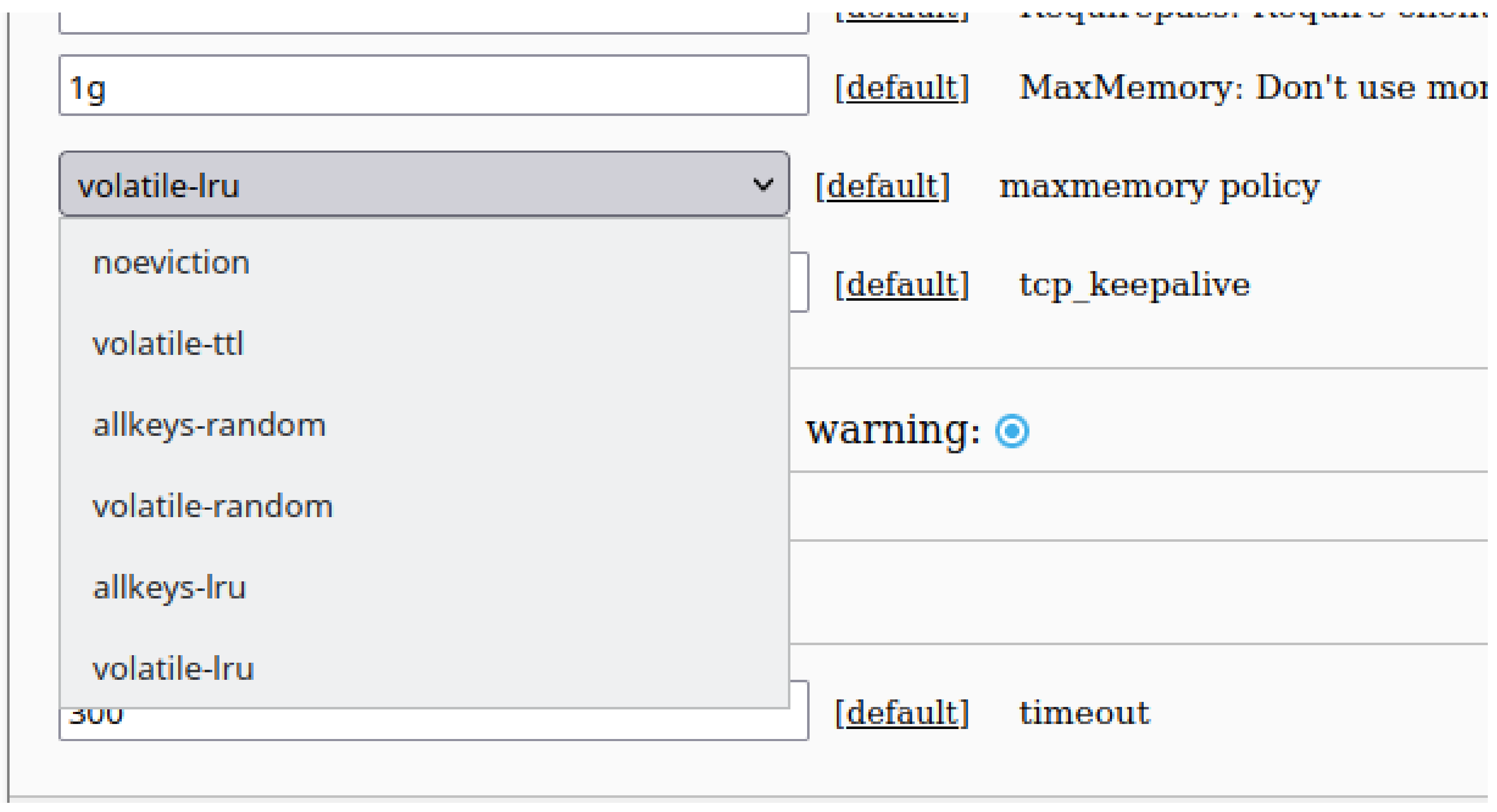
300 [default] timeout

Replication:

100 [default] slave-priority
[default] slaveof: ip port

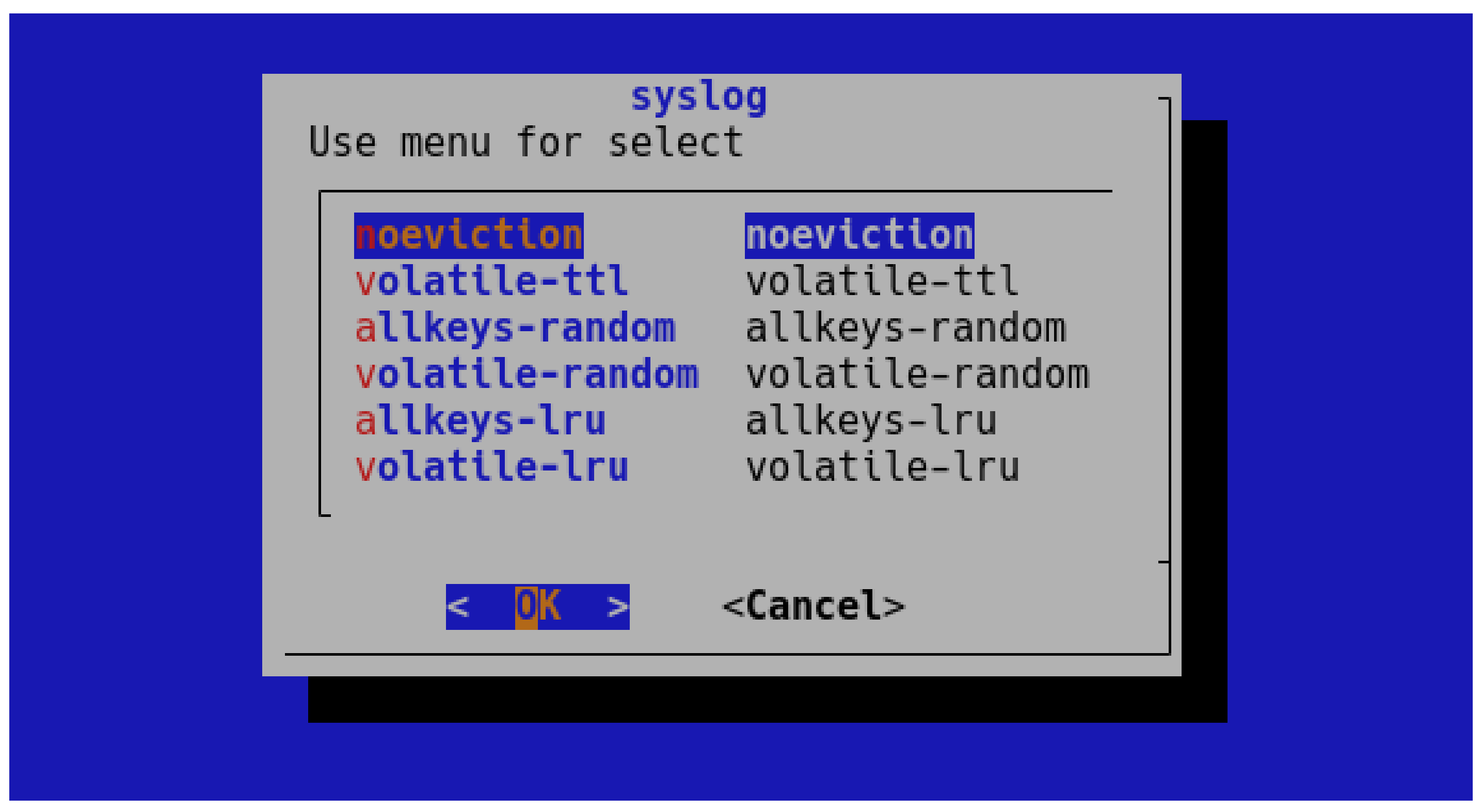
Apply Clear

Where the maxmemory policy parameters are in a specific dropdown element:



1g [default] MaxMemory: Don't use mo
volatile-lru [default] maxmemory policy
noeviction
volatile-ttl
allkeys-random
volatile-random
allkeys-lru
volatile-lru
[default] tcp_keepalive
warning: ●
300 [default] timeout

The TUI interface offers a similar choice:



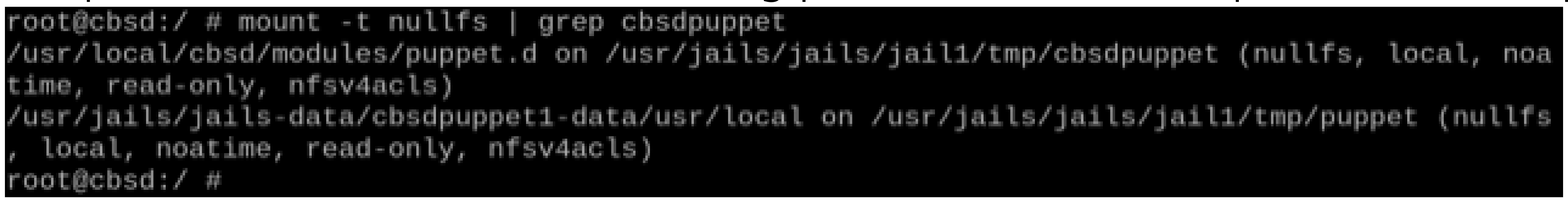
```
              syslog
Use menu for select

  noeviction       noeviction
  volatile-ttl     volatile-ttl
  allkeys-random   allkeys-random
  volatile-random  volatile-random
  allkeys-lru      allkeys-lru
  volatile-lru     volatile-lru

         <  OK  >      <Cancel>
```

Note: all CBSD TUI dialogs can be described by similar SQL-based forms which are currently used only for work with templates.

After getting and processing the parameters, after the cbsd forms, the cbsd puppet module intervenes in proceedings. Here's what happens next:

- the overlay file system mounting (having used nullfs) of the puppet7 package files from the cbsdpuppet1 system container to the `/tmp/XXX` directory to the destination container (jail1);
- YAML file presentation with parameters for the module to the destination container (jail1);
- fetching the puppet apply instruction to the configurable container (jail1) to apply the puppet manifest from `the /tmp/XXX directory`.

For example, this is what the nullfs mounting points look like when parameters are applied:



```
root@cbsd:/ # mount -t nullfs | grep cbsdpuppet
/usr/local/cbsd/modules/puppet.d on /usr/jails/jails/jail1/tmp/cbsdpuppet (nullfs, local, noa
time, read-only, nfsv4acls)
/usr/jails/jails-data/cbsdpuppet1-data/usr/local on /usr/jails/jails/jail1/tmp/puppet (nullfs
, local, noatime, read-only, nfsv4acls)
root@cbsd:/ #
```

Upon completion and reconfiguration of the service, temporary nullfs file systems are dismounted as superfluous. Therefore, if we look at the installed software in the jail1 container from the example above, there would not be the puppet7 package or the files it depends on.

```
~ # cbsd jexec jname=jail1 pkg info
pkg-1.17.4                     Package manager
redis-6.2.6                    Persistent key-value database
```

Stating the matter another way, with the use of the cbsdpuppet1 container, there is no need to install puppet7 and the necessary dependencies in each container--the configuration application is not dependent upon the presence (or absence) of any system software in a finite container.

Epilogue

- cbsd forms is one of the most powerful capabilities of the CBSD framework when working with jail-based containers, forming a bridge between CBSD and configuration managers. Accordingly, instead of the sed/awk scripts support and static templates, the CBSD project contributes FreeBSD support to the appropriate Puppet modules, if it is out of support: if a module can operate in a FreeBSD environment, you can use it via cbsd forms transparently. There are benefits for particular CBSD users and all the Puppet users on FreeBSD in general. If you use another configuration system, you can set the call of other systems by using the cbsd forms script.

The CBSD project sustains oscillation and distribution of ready-to-use images based on existing templates, which you can use through the CBSD repository using cbsd repo and cbsd images. See inline documentation and examples:

```
~ # cbsd repo --help
~ # cbsd images --help
```

In the next of the series of articles, we'll look at the virtual machine management capabilities of CBSD.

OLEG GINZBURG lives in Russia and works as a DevOps engineer in the X5 Retail group. He has been fond of computer science since the ZX Spectrum and is a Unix fan and FreeBSD enthusiast. He is a FreeBSD promoter and member of several projects: CBSD, ClonOS, MyB, K8S-bhyve, and AdvanceBSD group.