



# Univention Developer Reference

*Release 5.2-5*

May 11, 2026

The source of this document is licensed under GNU Affero General Public License v3.0 only.

# CONTENTS

<b>1</b>	<b>Foreword</b>	<b>1</b>
<b>2</b>	<b>Packaging software</b>	<b>3</b>
2.1	Introduction	3
2.2	Preparations	3
2.3	Example: Re-building an UCS package	3
2.3.1	Checking out and building a UCS package	4
2.4	Example: Creating a new UCS package	4
2.5	Setup repository	8
2.6	Building packages through the openSUSE Build Service	9
<b>3</b>	<b>Univention Config Registry</b>	<b>11</b>
3.1	Using UCR	11
3.1.1	Using UCR from shell	11
3.1.2	Using UCR from Python	12
3.2	Configuration files	14
3.2.1	debian/package.univention-config-registry	15
3.2.2	debian/package.univention-config-registry-variables	18
3.2.3	debian/package.univention-config-registry-categories	20
3.2.4	debian/package.univention-service	20
3.3	UCR Template files <i>conffiles/path/to/file</i>	21
3.4	Build integration	22
3.5	Examples	23
3.5.1	Minimal File example	23
3.5.2	Multifile example	24
3.5.3	Services	25
<b>4</b>	<b>Domain join</b>	<b>29</b>
4.1	Join scripts	29
4.2	Join status	29
4.3	Running join scripts	29
4.4	Writing join scripts	30
4.4.1	Basic join script example	30
4.4.2	Join script exit codes	32
4.4.3	Join script libraries	32
4.5	Writing unjoin scripts	37
<b>5</b>	<b>Lightweight Directory Access Protocol (LDAP) in UCS</b>	<b>39</b>
5.1	Packaging LDAP Schema Extensions	39
5.2	Packaging LDAP ACL Extensions	40
5.3	LDAP secrets	41
5.3.1	Password change	42
<b>6</b>	<b>Univention Directory Listener</b>	<b>43</b>
6.1	Structure of Listener Modules	43

6.1.1	Handle LDAP objects . . . . .	45
6.1.2	Initialize and clean . . . . .	45
6.1.3	Suspend and resume . . . . .	46
6.2	High-level Listener modules API . . . . .	47
6.3	Low-level Listener module . . . . .	52
6.4	Listener tasks and examples . . . . .	55
6.4.1	Listener API example . . . . .	55
6.4.2	Basic example . . . . .	56
6.4.3	Rename and move . . . . .	57
6.4.4	Full example with packaging . . . . .	58
6.4.5	A little bit more object oriented . . . . .	62
6.5	Technical Details . . . . .	65
6.5.1	User-ID and Credentials . . . . .	65
6.5.2	Internal Cache . . . . .	65
6.5.3	Internal working . . . . .	66
6.5.4	LDAP Schema handling . . . . .	68
6.5.5	Python 3 migration . . . . .	69
<b>7</b>	<b>Univention Directory Manager (UDM)</b>	<b>71</b>
7.1	UDM modules . . . . .	72
7.1.1	Overview . . . . .	72
7.1.2	Structure of a module . . . . .	72
7.1.3	Example module . . . . .	77
7.2	UDM syntax . . . . .	83
7.2.1	UDM syntax override . . . . .	83
7.2.2	UDM LDAP search . . . . .	84
7.3	Package extended attributes . . . . .	87
7.3.1	Selection lists . . . . .	91
7.3.2	Known issues . . . . .	93
7.3.3	Extended options . . . . .	93
7.3.4	Extended attribute hooks . . . . .	94
7.4	Package UDM hooks . . . . .	98
7.5	Package UDM extension modules . . . . .	98
7.6	Package UDM syntax extension . . . . .	99
7.7	UDM HTTP REST API . . . . .	100
7.7.1	API clients . . . . .	100
<b>8</b>	<b>Univention Management Console (UMC)</b>	<b>103</b>
8.1	Architecture . . . . .	103
8.2	Protocol HTTP for UMC . . . . .	103
8.3	UMC files . . . . .	104
8.3.1	<code>debian/package.umc-modules</code> . . . . .	104
8.3.2	UMC module declaration file . . . . .	105
8.4	Local system module . . . . .	105
8.4.1	Python API . . . . .	106
8.4.2	UMC module API (Python and JavaScript) . . . . .	106
8.4.3	Packaging . . . . .	110
8.5	Domain LDAP module . . . . .	112
8.6	Disabling a module . . . . .	113
<b>9</b>	<b>Web services</b>	<b>115</b>
9.1	Extending the overview page . . . . .	115
<b>10</b>	<b>App Center</b>	<b>117</b>
<b>11</b>	<b>Integration of external repositories</b>	<b>119</b>
11.1	Integrate with Univention Management Console . . . . .	119
11.2	Integrate with Univention Configuration Registry . . . . .	120

<b>12 Translate UCS</b>	<b>123</b>
12.1 Translating a single Debian package	123
12.1.1 Setup of <code>univention-110n-build</code>	123
12.1.2 UCS package translation workflow	124
12.2 Create a translation package for UCS	126
12.2.1 Install needed tools	126
12.2.2 Obtain a current checkout of the UCS Git repository	127
12.2.3 Create translation package	127
12.2.4 Edit translation files	127
12.2.5 Update the translation package	127
12.2.6 Build the translation package	127
12.3 Editing translation files	128
12.3.1 Editing translation entries	128
12.3.2 Update meta information	129
<b>13 Univention Updater</b>	<b>131</b>
13.1 Separate repositories	131
13.2 Updater scripts	131
13.2.1 Digital signature	132
13.3 Release update walk-through	132
<b>14 Miscellaneous</b>	<b>133</b>
14.1 Databases	133
14.1.1 PostgreSQL	133
14.1.2 MariaDB	133
14.2 UCS lint	133
14.3 Function libraries	134
14.3.1 <code>shell-univention-lib</code>	134
14.3.2 <code>python-univention-lib</code>	135
14.4 Login access control	136
14.5 Network packet filter	136
14.5.1 Filter rules by Univention Configuration Registry	136
14.5.2 Local filter rules through <code>iptables</code> commands	137
14.5.3 Testing Univention Firewall settings	137
14.6 Active Directory Connection custom mappings	137
<b>15 Appendix</b>	<b>139</b>
15.1 Bug reporting	139
15.2 Debian packaging	139
15.2.1 Prerequisites and preparation	139
15.2.2 <code>dh_make</code>	140
15.2.3 Debian control files	141
15.2.4 Building	149
15.2.5 Further reading	149
15.3 Bibliography	149
<b>Bibliography</b>	<b>151</b>
<b>Python Module Index</b>	<b>153</b>
<b>Index</b>	<b>155</b>



## FOREWORD

This developer guide provides information to extend Univention Corporate Server. It is targeted at third party vendors who intend to provide applications for the Univention App Center and for power users who wish to deploy locally built or modified software.

Feedback is very welcome. Please either file a bug (see [Bug reporting](#) (page 139)) or send an email to [feedback@univention.de](mailto:feedback@univention.de).



## PACKAGING SOFTWARE

This chapter describes how software for UCS is packaged. For more details on packaging software in the Debian format, see *Debian packaging* (page 139).

### 2.1 Introduction

UCS is based on the Debian distribution, that uses the deb format to package software. The program `dpkg` is used for handling a set of packages. On installation packages are unpacked and configured, while on un-installation packages are de-configured and the files belonging to the packages are removed from the system.

On top of that the `apt`-tools provide a software repository, which allows software to be downloaded from central file servers.

Package files provide an index of all packages contained in the repository, which is used to resolve dependencies between packages. While `dpkg` works on a set of packages given on the command line, `apt-get` builds that set of packages and their dependencies before invoking `dpkg` on this set. `apt-get` is a command line tool, which is fully described in its manual page `apt-get(8)`<sup>1</sup>. A more modern version with a text based user interface is `aptitude`, while `synaptic` provides a graphical frontend.

On UCS systems the administrator is not supposed to use these tools directly. Instead all software maintenance can be done through the UMC, which maps the requests to invocations of the commands given above.

### 2.2 Preparations

This chapter describes some simple examples using existing packages. For downloading and building them, some packages must be installed on the system used as a development system:

- `git` is used to checkout the source files belonging to the packages.
- `build-essential` must be installed for building the package.
- `devscripts` provides some useful tools for maintaining packages.

This can be achieved by running the following command:

```
$ sudo apt-get install git build-essential devscripts
```

### 2.3 Example: Re-building an UCS package

Source code: [UCS source: doc/developer-reference/packaging/testdeb/](https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/packaging/testdeb/)<sup>2</sup>

<sup>1</sup> <https://manpages.debian.org/bookworm/apt/apt-get.8.en.html>

<sup>2</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/packaging/testdeb/>

## 2.3.1 Checking out and building a UCS package

1. Create the top level working directory

```
$ mkdir work
$ cd work/
```

2. Either fetch the latest source code from the GIT version control system or download the source code of the currently packaged version.

- Checkout example package from GIT version control system:

```
$ git clone https://github.com/univention/univention-corporate-server.git
$ cd univention-corporate-server/base/univention-ssh
```

- Fetch the source code from the Univention Repository server:

- a. Enable the source repository once:

```
$ sudo ucr set repository/online/sources=yes
$ sudo apt-get update
```

- b. Fetch source code:

```
$ apt-get source univention-ssh
$ cd univention-ssh-*/
```

3. Increment the version number of package to define a newer package:

```
$ debchange --local work 'Private package rebuild'
```

4. Install the required build dependencies

```
$ sudo apt-get build-dep .
```

5. Build the binary package

```
$ dpkg-buildpackage -uc -us -b -rfakeroot
```

6. Locally install the new binary package

```
$ sudo apt-get install ../univention-ssh_*_*.deb
```

## 2.4 Example: Creating a new UCS package

The following example provides a walk-through for packaging a Python script called `testdeb.py`. It creates a file `testdeb-DATE-time` in the `/tmp/` directory.

A directory needs to be created for each source package, which hosts all other files and sub-directories.

```
$ mkdir testdeb-0.1
$ cd testdeb-0.1
```

The file `testdeb.py`, which is the program to be installed, will be put into that directory.

```
#!/usr/bin/python3
"""
Example for creating UCS packages.
"""
```

(continues on next page)

(continued from previous page)

```

from datetime import datetime

if __name__ == "__main__":
    now = datetime.now()
    filename = "/tmp/testdeb-{:y%m%d%H%M}".format(now)
    with open(filename, "a") as tmpfile:
        pass

```

In addition to the files to be installed, some metadata needs to be created in the `debian/` sub-directory. This directory contains several files, which are needed to build a Debian package. The files and their format will be described in the following sections.

To create an initial `debian/` directory with all template files, invoke the `dh_make(1)`<sup>3</sup> command provided by the package `dh-make`:

```
$ dh_make --native --single --email user@example.com
```

Here several options are given to create the files for a source package, which contains all files in one archive and only creates one binary package at the end of the build process. More details are given in *dh\_make* (page 140).

The program will output the following information:

```

Maintainer name   : John Doe
Email-Address     : user@example.com
Date              : Thu, 28 Feb 2013 08:11:30 +0100
Package Name     : testdeb
Version          : 0.1
License          : blank
Type of Package  : Single
Hit <enter> to confirm:

```

The package name `testdeb` and version `0.1` were determined from the name of the directory `testdeb-0.1`, the maintainer name and address were gathered from the UNIX account information.

After pressing the `Enter` key some warning message will be shown:

```

Currently there is no top level Makefile. This may require additional
tuning. Done. Please edit the files in the debian/ subdirectory now.
You should also check that the testdeb Makefiles install into $DESTDIR
and not in / .

```

Since this example is created from scratch, the missing `Makefile` is normal and this warning can be ignored. Instead of writing a `Makefile` to install the single executable, `dh_install` will be used later to install the file.

Since the command completed successfully, several files were created in the `debian/` directory. Most of them are template files, which are unused in this example. To improve understandability they are deleted:

```

$ rm debian/*.ex debian/*.EX
$ rm debian/README* debian/doc

```

The remaining files are required and control the build process of all binary packages. Most of them don't need to be modified for this example, but others must be completed using an editor.

#### **debian/control**

The file contains general information about the source and binary packages. It needs to be modified to include a description and contain the right build dependencies:

<sup>3</sup> [https://manpages.debian.org/bookworm/dh-make/dh\\_make.1.en.html](https://manpages.debian.org/bookworm/dh-make/dh_make.1.en.html)

```
Source: testdeb
Section: univention
Priority: optional
Maintainer: John Doe <user@example.com>
Build-Depends:
    debhelper-compat (= 13),
Standards-Version: 4.3.0.3

Package: testdeb
Architecture: all
Depends: ${misc:Depends}
Description: An example package for the developer guide
 This purpose of this package is to describe the structure of a Debian
 packages. It also documents
 .
 * the structure of a Debian/Univention package
 * installation process.
 * content of packages
 * format and function of control files
 .
For more information about UCS, refer to:
https://www.univention.de/
```

### **debian/rules**

This file has a **Makefile** syntax and controls the package build process. Because there is no special handling needed in this example, the default file can be used unmodified.

```
#!/usr/bin/make -f
%:
    dh $@
```

#### **Note**

Tabulators must be used for indentation in this file.

### **debian/testdeb.install**

To compensate the missing Makefile, `dh_install(1)`<sup>4</sup> is used to install the executable. `dh_install` is indirectly called by `dh` from the `debian/rules` file. To install the program into `/usr/bin/`, the file needs to be created manually containing the following single line:

```
testdeb.py usr/bin/
```

#### **Note**

The path is not absolute, but relative.

### **debian/testdeb.postinst**

Since for this example the program should be invoked automatically during package installation, this file needs to be created. In addition to just invoking the program shipped with the package itself, it also shows how Univention Configuration Registry Variables can be set. For more information, see *Using UCR from shell* (page 11).

---

<sup>4</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_install.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_install.1.en.html)

```
#!/bin/sh
set -e

case "$1" in
configure)
    # invoke sample program
    testdeb.py
    # Set UCR variable if previously unset
    ucr set repository/online/server?https://updates.software-univention.de/
    # Force UCR variable on upgrade from previous package only
    if dpkg --compare-versions "$2" lt-nl 0.1-2
    then
        ucr set timeserver1=time.fu-berlin.de
    fi
    ;;
abort-upgrade|abort-remove|abort-deconfigure)
    ;;
*)
    echo "postinst called with unknown argument \"$1\"" >&2
    exit 1
    ;;
esac

#DEBHELPER#

exit 0
```

**debian/changelog**

The file is used to keep track of changes done to the packaging. For this example the file should look like this:

```
testdeb (0.1-1) unstable; urgency=low

 * Initial Release.

-- John Doe <user@example.com> Mon, 21 Mar 2013 13:46:39 +0100
```

**debian/copyright**

This file is used to collect copyright related information. It is critical for Debian only, which need this information to guarantee that the package is freely re-distributable. For this example the file remains unchanged.

The `copyright` and `changelog` file are installed to the `/usr/share/doc/testdeb/` directory on the target system.

**debian/source/format**

This file control some internal aspects of the package build process. It can be ignored for the moment and are further described in *Debian control files* (page 141).

Now the package is ready and can be built by invoking the following command:

```
$ dpkg-buildpackage -us -uc
```

The command should then produce the following output:

```
dpkg-buildpackage: info: source package testdeb
dpkg-buildpackage: info: source version 0.1-1
dpkg-buildpackage: info: source distribution unstable
dpkg-buildpackage: info: source changed by John Doe <user@example.com>
dpkg-buildpackage: info: host architecture amd64
dpkg-source --before-build .
```

(continues on next page)

(continued from previous page)

```
debian/rules clean
dh clean
  dh_clean
  dpkg-source -b .
dpkg-source: info: using source format '1.0'
dpkg-source: warning: source directory 'testdeb' is not <sourcepackage>-
↳<upstreamversion> 'testdeb-0.1'
dpkg-source: info: building testdeb in testdeb_0.1-1.tar.gz
dpkg-source: info: building testdeb in testdeb_0.1-1.dsc
  debian/rules build
dh build
  dh_update_autotools_config
  dh_autoreconf
  create-stamp debian/debhelper-build-stamp
  debian/rules binary
dh binary
  dh_testroot
  dh_prep
  dh_install
  dh_installdocs
  dh_installchangelogs
  dh_perl
  dh_link
  dh_strip_nondeterminism
  dh_compress
  dh_fixperms
  dh_missing
  dh_installdeb
  dh_gencontrol
  dh_md5sums
  dh_builddeb
dpkg-deb: building package 'testdeb' in '../testdeb_0.1-1_all.deb'.
dpkg-genbuildinfo
dpkg-genchanges >../testdeb_0.1-1_amd64.changes
dpkg-genchanges: info: including full source code in upload
dpkg-source --after-build .
dpkg-buildpackage: info: full upload; Debian-native package (full source is_
↳included)
```

The binary package file `testdeb_0.1-1_all.deb` is stored in the parent directory. When it is installed manually using `dpkg -i ../testdeb_0.1-2_all.deb` as root, the Python script is installed as `/usr/bin/testdeb.py`. It is automatically invoked by the `postint` script, so a file named `/tmp/testdeb-date-time` has been created, too.

Congratulations! You've successfully built your first own Debian package.

## 2.5 Setup repository

Until now the binary package is only available locally. For installation you must manually copy it to each host and manually install it using `dpkg -i`.

If the package requires additional dependencies, the installation process cancels, because `dpkg` doesn't download dependencies, but `apt` does. To support automatic installation and dependency resolution, you must copy the package to an `apt` repository, that's available through HTTP.

The following example creates a repository under `/var/www/repository/`. All UCS systems with `apache2` installed export this directory by default. For compatibility reasons with the UCS `Updater`, you need to create several subdirectories inside this directory.

The following commands create a repository for UCS 5.0 with the component name `testcomp`:

```
$ WWW_BASE="/var/www/repository/5.2/maintained/component"
$ TESTCOMP="testcomp/all"
$ install -m755 -d "$WWW_BASE/$TESTCOMP"
$ install -m644 -t "$WWW_BASE/$TESTCOMP" *.deb
$ ( cd "$WWW_BASE"
  rm -f "$TESTCOMP/Packages"*
  apt-ftparchive packages "$TESTCOMP" > "Packages"
  gzip -9 < "Packages" > "$TESTCOMP/Packages.gz"
  mv "Packages" "$TESTCOMP/Packages" )
```

You can then include this repository on any UCS system by appending the following line to `/etc/apt/sources.list`, assuming that the FQDN of the host with the repository is `repository.example.com`.

```
deb [trusted=yes] http://repository.example.com/repository/5.2/maintained/
↪component testcomp/all/
```

### Important

The directory, from where you run the `apt-ftparchive` command, must match the first string given in the `sources.list` file after the `deb` prefix. The URL together with the suffix `testcomp/all/` not only specifies the location of the `Packages` file, but the package manager also uses it as the base URL for all packages listed in the `Packages` file.

Instead of editing the `sources.list` file directly, you can include the repository as a component and configure it by setting several UCR variables. You can also configure UCR variables through UDM policies, which simplifies the task of installing packages from such a repository on many hosts. For the repository before you need to set the following variables:

- `repository/online/component/NAME` (page 120)
- `repository/online/component/NAME/server` (page 120)

```
$ ucr set \
  repository/online/component/testcomp=enabled \
  repository/online/component/testcomp/server=https://repository.example.com/
↪repository
```

## 2.6 Building packages through the openSUSE Build Service

The openSUSE Build Service (OBS) is a framework to generate packages for a wide range of distributions. Additional information can be found at [OpenSUSE Build Service](https://build.opensuse.org/)<sup>5</sup>.

If OBS is already used to build packages for other distributions, it can also be used for Univention Corporate Server builds. The build target for UCS 4.4 is called *Univention UCS 4.4*. Note that OBS doesn't handle the integration steps described in later chapters, for example the use of Univention Configuration Registry templates.

<sup>5</sup> <https://build.opensuse.org/>



## UNIVENTION CONFIG REGISTRY

The Univention Configuration Registry (UCR) is a local mechanism, which is used on all UCS system roles to consistently configure all services and applications. It consists of a database, where the currently configured values are stored, and a mechanism to trigger certain actions, when values are changed. This is mostly used to create configuration files from templates by filling in the configured values. In addition to using simple place holders its also possible to use Python code for more advanced templates or to call external programs when values are changed. UCR values can also be configured through an UDM policy in Univention directory service (LDAP), which allows values to be set consistently for multiple hosts of a domain.

### 3.1 Using UCR

Univention Configuration Registry provides two interfaces, which allows easy access from shell scripts and Python programs.

#### 3.1.1 Using UCR from shell

**univention-config-registry** (and its alias **ucr**) can be invoked directly from shell. The most commonly used functions are:

**ucr set [ key=value | key?value ] ...**

Set Univention Configuration Registry Variable *key* to the given *value*. Using **=** forces an assignment, while **?** only sets the value if the variable is unset.

Listing 3.1: Use of **ucr set**

```
$ ucr set print/papersize?a4 \
variable/name=value
```

**ucr get key**

Return the current value of the Univention Configuration Registry Variable *key*.

Listing 3.2: Use of **ucr get**

```
case "$(ucr get system/role)" in
    domaincontroller_*)
        echo "Running on a UCS Directory Node"
        ;;
esac
```

For variables containing boolean values the shell-library-function `is_ucr_true key` from `/usr/share/univention-lib/ucr.sh` should be used. It returns 0 (success) for the values 1, yes, on, true, enable, enabled, 1 for the negated values 0, no, off, false, disable, disabled. For all other values it returns a value of 2 to indicate inappropriate usage.

Listing 3.3: Use of `is_ucr_true`

```
. /usr/share/univention-lib/ucr.sh
if is_ucr_true update/secure_apt
then
    echo "The signature check for UCS packages is enabled."
fi
```

**ucr unset key ...**

Unset the Univention Configuration Registry Variable `key`.

Listing 3.4: Use of `ucr unset`

```
$ ucr unset print/papersize variable/name
```

**ucr shell [ key ...]**

Export some or all Univention Configuration Registry Variables in a shell compatible manner as environment variables. All shell-incompatible characters in variable names are substituted by underscores (`_`).

Listing 3.5: Use of command: `ucr shell`

```
eval "$(ucr shell)"
case "$server_role" in
    domaincontroller_*)
        echo "Running on a UCS Domain Controller serving $ldap_base"
        ;;
esac
```

It is often easier to export all variables once and then reference the values through shell variables.

**Warning**

Be careful with shell quoting, since several Univention Configuration Registry Variables contain shell meta characters. Use `eval "$(ucr shell)"`.

**Note**

`ucr` is installed as `/usr/sbin/ucr`, which is not on the search path `$PATH` of normal users. Changing variables requires root access to `/etc/univention/base.conf`, but reading works for normal users too, if `/usr/sbin/ucr` is invoked directly.

### 3.1.2 Using UCR from Python

UCR also provides a Python binding, which can be used from any Python program. An instance of `univention.config_registry.ConfigRegistry` needs to be created first. After loading the current database state with `load()` the values can be accessed by using the instance like a Python dictionary:

Listing 3.6: Reading a Univention Configuration Registry variable in Python

```
from univention.config_registry import ConfigRegistry
ucr = ConfigRegistry()
ucr.load()
print(ucr['variable/name'])
print(ucr.get('variable/name', '<not set>'))
```

Since UCS 5.0 several new APIs are provided to simplify reading UCR settings:

#### **ucr**

This is a lazy-loaded shared instance, which only allows reading values. It is implemented as a singleton, so all modules using it share the same instance (per process). It can be refreshed by invoking `load()`.

Listing 3.7: Reading a Univention Configuration Registry variable in Python

```
from univention.config_registry import ucr
print(ucr["ldap/base"])
```

#### **ucr\_live**

In contrast to `ucr` this shared singleton instance automatically reloads the settings. This is done on each access, but only happens if the files on disk actually changed.

Listing 3.8: Reading a Univention Configuration Registry variable in Python

```
from univention.config_registry import ucr_live
print(ucr_live["version/erratalevel"])
```

Repeated reads of the same key may return different values due to the live character. Reading multiple keys in sequence is not atomic as other processes might update UCR in between. Reading many keys is slower due to the extra check for updated files. To mitigate this a frozen view (a read-only snapshot with auto reload disabled) is created when this instance is used as a Python context manager:

Listing 3.9: Reading a Univention Configuration Registry variable in Python

```
from univention.config_registry import ucr_live
with ucr_live as view:
    for key, value in view.items():
        print(key, value)
```

#### **ucr\_factory**

This function can be used to create a new private instance. All values are already loaded.

Listing 3.10: Reading a Univention Configuration Registry variable in Python

```
from univention.config_registry import ucr_factory
ucr = ucr_factory()
print(ucr["version/erratalevel"])
```

For variables containing boolean values the methods `is_true()` and `is_false()` should be used. The former returns `True` for the values `1`, `yes`, `on`, `true`, `enable`, `enabled`, while the later one returns `True` for the negated values `0`, `no`, `off`, `false`, `disable`, `disabled`. Both methods accept an optional argument `default`, which is returned as-is when the variable is not set.

Listing 3.11: Reading boolean Univention Configuration Registry variables in Python

```
if ucr.is_true('update/secure_apt'):
    print("package signature check is explicitly enabled")
if ucr.is_true('update/secure_apt', True):
    print("package signature check is enabled")
if ucr.is_false('update/secure_apt'):
    print("package signature check is explicitly disabled")
if ucr.is_false('update/secure_apt', True):
```

(continues on next page)

(continued from previous page)

```
print("package signature check is disabled")
```

Modifying variables requires a different approach. The function `ucr_update()` should be used to set and unset variables.

Listing 3.12: Changing Univention Configuration Registry variables in Python

```
from univention.config_registry.frontend import ucr_update
ucr_update(ucr, {
    'foo': 'bar',
    'baz': '42',
    'bar': None,
})
```

The function `ucr_update()` requires an instance of `ConfigRegistry` (returned by `ucr_factory()`) as its first argument. The method is guaranteed to be atomic and internally uses file locking to prevent race conditions.

The second argument must be a Python dictionary mapping UCR variable names to their new value. The value must be either a string or `None`, which is used to unset the variable.

As an alternative the old functions `handler_set()` and `handler_unset()` can still be used to set and unset variables. Both functions expect an array of strings with the same syntax as used with the command line tool `ucr`. As the functions `handler_set()` and `handler_unset()` don't automatically update any instance of `ConfigRegistry`, the method `load()` has to be called manually afterwards to reflect the updated values.

Listing 3.13: Setting and unsetting Univention Configuration Registry variables in Python

```
from univention.config_registry import handler_set, handler_unset
handler_set(['foo=bar', 'baz?42'])
handler_unset(['foo', 'bar'])
```

Listing 3.14: Getting integer values from Univention Configuration Registry variables in Python

```
from univention.config_registry import ucr
print(ucr.get_int("key"))
print(ucr.get_int("key", 10))
```

## 3.2 Configuration files

Packages can use the UCR functionality to create customized configuration files themselves. UCR diverts files shipped by Debian packages and replaces them by generated files. If variables are changed, the affected files are committed, which regenerated their content. This diversion is persistent and even outlives updates, so they are not overwritten by configuration files of new packages.

For this, packages need to ship additional files:

### **`conffiles/path/to/file`**

This template file is used to create the target file. There exist two variants:

1. A *single file template* consists of only a single file, from which the target file is created.
2. A *multi file template* can consist of multiple file fragments, which are concatenated to form the target file.

For more information, see *UCR Template files `conffiles/path/to/file`* (page 21).

### **`debian/package.univention-config-registry`**

This mandatory information file describes the each template file. It specifies the type of the template and lists

the UCR variable names, which shall trigger the regeneration of the target file.

For more information, see *debian/package.univention-config-registry* (page 15).

#### **debian/package.univention-config-registry-variables**

This optional file can add descriptions to UCR variables, which should describe the use of the variable, its default and allowed values.

For more information, see *debian/package.univention-config-registry-variables* (page 18).

#### **debian/package.univention-config-registry-categories**

This optional file can add additional categories to group UCR variables.

For more information, see *debian/package.univention-config-registry-categories* (page 20).

#### **debian/package.univention-service**

This optional file is used to define long running services.

For more information, see *debian/package.univention-config-registry-categories* (page 20).

In addition to these files, code needs to be inserted into the package maintainer scripts (see *debian/preinst*, *debian/prerm*, *debian/postinst*, *debian/postrm* (page 148)), which registers and un-registers these files. This is done by calling **univention-install-config-registry** from *debian/rules* during the package build binary phase. The command is part of the **univention-config-dev** package, which needs to be added as a Build-Depends build dependency of the source package in *debian/control*.

### 3.2.1 *debian/package.univention-config-registry*

This file describes all template files in the package. The file is processed and copied by **univention-install-config-registry** into */etc/univention/templates/info/* when the package is built.

It can consist of multiple sections, where sections are separated by one blank line. Each section consists of multiple key-value-pairs separated by a colon followed by one blank. A typical entry has the following structure:

```
Type: <type>
[Multifile|File]: <filename>>
[Subfile: <fragment-filename>]
Variables: <variable1>
...
```

Type specifies the type of the template, which the following sections describe in more detail.

#### **File**

A single file template is specified as type *file*. It defines a template, were the target file is created from only a single source file. A typical entry hat the following structure:

```
Type: file
File: <filename>
Variables: <variable1>
User: <owner>
Group: <group>
Mode: <file-mode>
Preinst: <module>
Postinst: <module>
...
```

The following keys can be used:

#### **File (required)**

Specifies both the target and source file name, which are identical. The source file containing the template must be put below the *conffiles/* directory. The file can contain any textual content and is processed as described in *UCR Template files conffiles/path/to/file* (page 21).

The template file is installed to `/etc/univention/templates/files/`.

#### Variables (optional)

This key can be given multiple times and specifies the name of UCR variables, which trigger the file commit process. This is normally only required for templates using `@!@` Python code regions. Variables used in `@%@` sections do not need to be listed explicitly, since `ucr` extracts them automatically.

The variable name is actually a Python regular expression, which can be used to match, for example, all variable names starting with a common prefix.

#### User (optional); Group (optional); Mode (optional)

These specify the symbolic name of the user, group and octal file permissions for the created target file. If no values are explicitly provided, then `root:root` is used by default and the file mode is inherited from the source template.

#### Preinst (optional); Postinst (optional)

These specify the name of a Python module located in `/etc/univention/templates/modules/`, which is called before and after the target file is re-created. The module must implement the following two functions:

```
def preinst(
    config_registry: ConfigRegistry,
    changes: dict[str, tuple[str | None, str | None]],
) -> None:
    pass
def postinst(
    config_registry: ConfigRegistry,
    changes: dict[str, tuple[str | None, str | None]],
) -> None:
    pass
```

Each function receives two arguments: The first argument `config_registry` is a reference to an instance of `ConfigRegistry`. The second argument `changes` is a dictionary of 2-tuples, which maps the names of all changed variables to `(old-value, new-value)`.

`univention-install-config-registry` installs the module file to `/etc/univention/templates/modules/`.

If a script `/etc/univention/templates/scripts/full-path-to-file` exists, it will be called after the file is committed. The script is called with the argument `postinst`. It receives the same list of changed variables as documented in *Script* (page 17).

#### Multifile

A multi file template is specified once as type `multifile`, which describes the target file name. In addition to that multiple sections of type `subfile` are used to describe source file fragments, which are concatenated to form the final target file. A typical multifile has the following structure:

```
Type: multifile
Multifile: <target-filename>
User: <owner>
Group: <group>
Mode: <file-mode>
Preinst: <module>
Postinst: <module>
Variables: <variable1>

Type: subfile
Multifile: <target-filename>
Subfile: <fragment-filename>
Variables: <variable1>
...
```

The following keys can be used:

**Multifile (required)**

This specifies the target file name. It is also used to link the `multifile` entry to its corresponding `subfile` entries.

**Subfile (required)**

The source file containing the template fragment must be put below the `conffiles/` directory in the Debian source package. The file can contain any textual content and is processed as described in *UCR Template files* `conffiles/path/to/file` (page 21). The template file is installed to `/etc/univention/templates/files/`.

Common best practice is to start the filename with two digits to allow consistent sorting and to put the file in the directory named like the target filename suffixed by `.d`, that is `conffiles/target-filename.d/00fragment-filename`.

**Variables (optional)**

Variables can be declared in both the `multifile` and `subfile` sections. The variables from all sections trigger the commit of the target file. Until UCS-2.4 only the `multifile` section was used, since UCS-3.0 the `subfile` section should be preferred (if needed).

**User (optional); Group (optional); Mode (optional); Preinst (optional); Postinst (optional)**

Same as above for `file`.

The same script hook as above for `file` is also supported.

**Script**

A script template allows an external program to be called when specific UCR variables are changed. A typical script entry has the following structure:

```
Type: script
Script: <filename>
Variables: <variable1>
```

The following keys can be used:

**Script (required)**

Specifies the filename of an executable, which is installed to `/etc/univention/templates/scripts/`.

The script is called with the argument `generate`. It receives the list of changed variables on standard input. For each changed variable a line containing the name of the variable, the old value, and the new value separated by `@%@` is sent.

**Variables (required)**

Specifies the UCR variable names, which should trigger the script.

**Warning**

There is **no** guarantee that `Script` is executed **after** a file has been committed. If this is required for example for restarting a service place the script instead at the location mentioned at the end of *File* (page 15).

**Note**

The script interface is quiet limited for historical reasons. Consider it deprecated in favor of *Module* (page 17).

**Module**

A module template allows a Python module to be run when specific UCR variables are changed. A typical module entry has the following structure:

```
Type: module
Module: <filename>
Variables: <variable1>
```

The file referenced with `Module: <filename>`, must locate in the `/conffiles` directory of the Debian package. For example: For the `Module: my_module.py` definition, you have the file `/conffiles/my_module.py` in the Debian package structure. The Python file defines at least the function mentioned at [Module \(required\)](#) (page 18).

The following keys can be used:

#### Module (required)

Specifies the filename of a Python module, which is installed to `/etc/univention/templates/modules/`.

The module must implement the following function:

```
def handler(
    config_registry: ConfigRegistry,
    changes: dict[str, tuple[str | None, str | None]],
) -> None:
    pass
```

The function receives two arguments: The first argument `config_registry` is a reference to an instance of `ConfigRegistry`. The second argument `changes` is a dictionary of 2-tuples, which maps the names of all changed variables to `(old-value, new-value)`.

`univention-install-config-registry` installs the module to `/etc/univention/templates/modules/`.

#### Variables (required)

Specifies the UCR variable names, which should trigger the module.

#### Warning

There is **no** guarantee that `Module` is executed **after** a file has been committed. If this is required for e.g. restarting a service use `Preinst` or `Postinst` as mentioned in [File](#) (page 15) instead.

### 3.2.2 `debian/package.univention-config-registry-variables`

For UCR variables a description should be registered. This description is shown in the *Univention Config Registry* module of the UMCas a mouse-over. It can also be queried by running `ucr info variable/name` on the command line.

The description is provided on a per-package basis as a file, which uses the ini-style format. The file is processed and copied by `univention-install-config-registry-info` into `/etc/univention/registry.info/variables/`. The command `univention-install-config-registry-info` is invoked indirectly by `univention-install-config-registry`, which should be called instead from `debian/rules`.

For each variable a section of the following structure is defined:

```
[<variable/name>]
Description[en]=<description>
Description[<language>]=<description>
Type=<type>
Elementtype=<type of all list elements>
Separator=<regular expression for separating list elements>
Min=<type constraint range minimum>
Max=<type constraint range maximum>
Regex=<type constraint regular expression>
Default=<default value>
```

(continues on next page)

(continued from previous page)

```
ReadOnly=<yes|no>
Categories=<category,...>
```

**[variable/name] (required)**

For each variable description one section needs to be created. The name of the section must match the variable name.

To describe multiple variables with a common prefix and/or suffix, the regular expression `.*` can be used to match any sequence of characters. This is the only supported regular expression!

**Description [language] (required)**

A descriptive text for the variable. It should mention the valid and default values. The description can be given in multiple languages, using the two-letter-code following *ISO 639-1: Alpha-2 code* [1].

**Type (required)**

The syntax type for the value. This is used since UCS 5.0-2 for validating the input. Valid values include:

- `str` for strings
- `json` for JSON strings
- `ipv4address` for IPv4 addresses
- `ipv6address` for IPv6 addresses
- `ipaddress` for IPv6 addresses
- `url_proxy` for HTTP/HTTP proxy URLs
- `bool` for boolean values
- `int` for integers ( $-\infty \dots \infty$ )
- `uint` for unsigned integers ( $0 \dots \infty$ )
- `pint` for positive integers ( $1 \dots \infty$ )
- `portnumber` for TCP/UDP port numbers 0-65535
- `list` for lists of items separated by some character

**Elementtype (required for Type=list)**

Specifies the type for all elements of type `list`.

**Separator (optional)**

For type `list` a regular expression used as the separator of the list elements. Default separator is a comma.

**Min (optional)**

Optional constraint for variables of type `int` defining the smallest possible value the variable can take.

**Max (optional)**

Optional constraint for variables of type `int` defining the largest possible value the variable can take.

**Regex (optional)**

Optional constraint for variables of type `str` defining a valid regular expression the value has to match.

**Default (optional)**

Added in version 5.0-0.

The default value of the UCR variable which is applied if the variable is not set. The default value might be a UCR pattern referencing other variables, for example `Default=%@another/variable@%` example.

**ReadOnly (optional)**

This declares a variable as read-only and prohibits changing the value through UMC. The restriction **isn't** applied when using the command line tool `ucr`. Valid values are `true` for read-only and `false`, which is the default.

**Categories (required)**

A list of categories, separated by comma. This is used to group related UCR variables. New categories don't need to be declared explicitly, but it is recommended to do so following *debian/package.univention-config-registry-categories* (page 20).

**3.2.3 debian/package.univention-config-registry-categories**

UCR variables can be grouped into categories, which can help administrators to find related settings. Categories are referenced from *.univention-config-registry-variables* files (see *debian/package.univention-config-registry-variables* (page 18)). They are created on-the-fly, but can be described further by explicitly defining them in a *.univention-config-registry-categories* file.

The description is provided on a per-package basis as a file, which uses the INI-style format. The file is processed and copied by **univention-install-config-registry-info** into */etc/univention/registry.info/categories/*. The command **univention-install-config-registry-info** is invoked indirectly by **univention-install-config-registry**, which should be called instead from *debian/rules*.

For each category a section of the following structure is defined:

```
[<category-name>]
name[en]=<name>
name[<language>]=<translated-name>
icon=<file-name>
```

**[category-name]**

For each category description one section needs to be created.

**name [language] (required)**

A descriptive text for the category. The description can be given in multiple languages, using the two-letter-code following *ISO 639-1: Alpha-2 code* [1].

**icon (required)**

The filename of an icon in either the Portable Network Graphics (PNG) format or Graphics Interchange Format (GIF). This is unused in UCS-3.1, but future versions might display this icon for variables in this category.

**3.2.4 debian/package.univention-service**

Long running services should be registered with UCR and UMC. This enables administrators to control these daemons using the UMC module *System services*.

The description is provided on a per-package basis as a file, which uses the ini-style format. The file is processed and copied by **univention-install-service-info** into */etc/univention/service.info/services/*. The command **univention-install-service-info** is invoked indirectly by **univention-install-config-registry**, which should be called instead from *debian/rules*.

For each service a section of the following structure is defined:

```
[<service-name>]
description[<language>]=<description>
start_type=<service-name>/autostart
systemd=<service-name>.service
icon=<service/icon_name>
programs=<executable>
name=<service-name>
init_script=<init.name>
```

**[service-name]; name=service-name (optional)**

For each daemon one section needs to be created. The service-name should match the name of the init-script in */etc/init.d/*. If the name differs, it can be overwritten by the `name=` property.

**description** [*language*] (required)

A descriptive text for the service. The description can be given in multiple languages, using the two-letter-code following *ISO 639-1: Alpha-2 code* [1].

**start\_type** (required)

Specifies the name of the UCR variable, which controls if the service should be started automatically. It is recommended to use the shell library `/usr/share/univention-config-registry/init-autostart.lib` to evaluate the setting from the init-script of the service. If the variable is set to `false` or `no`, the service should never be started. If the variable is set to `manually`, the service is explicitly not started during system boot. The service can still be started manually. It should be noted that if other services are started that have a dependency on a service marked as `manually`, the service marked as `manually` will also be started.

**systemd** (optional)

A comma separated list of **systemd** service names, which are enabled/disabled/masked when `start_type` is used. This defaults to the name of the service plus the suffix `.service`.

**init\_script** (optional)

The name of the legacy init script below `/etc/init.d/`. This defaults to the name of the service. This option should not be used any more in favor of **systemd**.

**programs** (required)

A comma separated list of commands, which must be running to qualify the service as running. Each command name is checked against `/proc/*/cmdline`. To check the processes for additional arguments, the command can also consist of additional shell-escaped arguments.

**icon** (unused)

This is unused in UCS, but future versions might display the icon for the service. The file name of an icon in either Portable Network Graphics (PNG) format or Graphics Interchange Format (GIF) format.

### 3.3 UCR Template files `conffiles/path/to/file`

For each file, which should be written, one or more template files need be created below the `conffiles/` directory. For a single file template (see [File](#) (page 15)), the filename must match the filename given in the `File:` stanza of the *file* entry itself. For a multi file template (see [Multifile](#) (page 16)), the filename must match the filename given in the `File:` stanza of the *subfile* entries.

Each template file is normally a text file, where certain sections get substituted by computed values during the file commit. Each section starts and ends with a special marker. UCR currently supports the following kinds of markers:

**@@@ variable reference**

Sections enclosed in @@@ are simple references to Univention Configuration Registry Variable. The section is replaced inline by the current value of the variable. If the variable is unset, an empty string is used.

**ucr** scans all `files` and `subfiles` on registration. All Univention Configuration Registry Variables used in @@@ are automatically extracted and registered for triggering the template mechanism. They don't need to be explicitly enumerated with `Variables:` statements in the file `debian/package.univention-config-registry`.

**@!@ Python code**

Sections enclosed in @!@ contain Python code. Everything printed to `STDOUT` by these sections is inserted into the generated file. The Python code can access the `configRegistry` variable, which is an already loaded instance of `ConfigRegistry`. Each section is evaluated separately, so no state is kept between different Python sections.

All Univention Configuration Registry Variables used in a @!@ Python section must be manually matched by a `Variables:` statement in the `debian/package.univention-config-registry` file. Otherwise the file is not updated on changes of the UCR variable.

**@@@UCRWARNING=%PREFIX@@@; @@@UCRWARNING\_ASCII=%PREFIX@@@**

This variant of the variable reference inserts a warning text, which looks like this:

```
# Warning: This file is auto-generated and might be overwritten by
#          univention-config-registry.
#          Please edit the following file(s) instead:
# Warnung: Diese Datei wurde automatisch generiert und kann durch
#          univention-config-registry überschrieben werden.
#          Bitte bearbeiten Sie an Stelle dessen die folgende(n) Datei(en):
#
#          /etc/univention/templates/files/etc/hosts.d/00-base
#          /etc/univention/templates/files/etc/hosts.d/20-static
#          /etc/univention/templates/files/etc/hosts.d/90-ipv6defaults
#
```

It should be inserted once at the top to prevent the user from editing the generated file. For single File templates, it should be on the top of the template file itself. For multi file templates, it should only be on the top the first sub-file.

Everything between the equal sign and the closing @%% defines the *PREFIX*, which is inserted at the beginning of each line of the warning text. For shell scripts, this should be # and a space character, but other files use different characters to start a comment. For files, which don't allow comments, the header should be skipped.

### Warning

Several file formats require the file to start with some *magic data*. For example shell scripts must start with a hash-bang (#!) and XML files must start with `<?xml version="1.0" encoding="UTF-8"?>` (if used). Make sure to put the warning after these headers!

The `UCRWARNING_ASCII` variant only emits 7-bit ASCII characters, which can be used for files, which are not 8 bit clean or unicode aware.

## 3.4 Build integration

During package build time `univention-install-config-registry` needs to be called. This should be done using the sequence `ucr` in `debian/rules`:

```
%.
dh $@ --with ucr
```

This invocation copies the referenced files to the right location in the binary package staging area `debian/package/etc/univention/`. Internally `univention-install-config-registry-info` and `univention-install-service-info` are invoked, which should not be called explicitly anymore.

The calls also insert code into the files `debian/package.preinst.debhelper`, `debian/package.postinst.debhelper` and `debian/package.prerm.debhelper` to register and de-register the templates. Therefore it's important that customized maintainer scripts use the `#DEBHELPER#` marker, so that the generated code gets inserted into the corresponding `preinst`, `postinst` and `prerm` files of the generated binary package.

The invocation also adds `univention-config` to `misc:Depends` to ensure that the package is available during package configuration time. Therefore it's important that `${misc:Depends}` is used in the `Depends` line of the package section in the `debian/control` file.

```
Package: ...
Depends: ..., ${misc:Depends}, ...
```

## 3.5 Examples

This sections contains several simple examples for the use of Univention Configuration Registry. The complete source of these examples is available separately. The download location is given in each example below. Since almost all Univention Corporate Server packages use UCR, their source code provides additional examples.

### 3.5.1 Minimal File example

This example provides a template for `/etc/papersize`, which is used to configure the default paper size. A Univention Configuration Registry Variable `print/papersize` is registered, which can be used to configure the paper size.

Source code: [UCS source: doc/developer-reference/ucr/papersize/](https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/ucr/papersize/)<sup>6</sup>

#### `conffiles/etc/papersize`

The template file only contains one line. Please note that this file does not start with the “UCRWARNING”, since the file must only contain the paper size and no comments.

```
@%@print/papersize@%
```

#### `debian/papersize.univention-config-registry`

The file defines the templates and is processed by `univention-install-config-registry` during the package build and afterwards by `univention-config-registry` during normal usage.

```
Type: file
File: etc/papersize
```

#### `debian/papersize.univention-config-registry-variables`

The file describes the newly defined Univention Configuration Registry Variable.

```
[print/papersize]
Description[en]=specify preferred paper size [a4]
Description[de]=Legt die bevorzugte Papiergröße fest [a4]
Type=str
Categories=service-cups
```

#### `debian/papersize.postinst`

Sets the Univention Configuration Registry Variable to a default value after package installation.

```
#!/bin/sh

case "$1" in
configure)
    ucr set print/papersize?a4
    ;;
esac

#DEBHELPER#

exit 0
```

#### `debian/rules`

Invoke `univention-install-config-registry` during package build to install the files to the appropriate location. It also creates the required commands for the maintainer scripts (see `debian/preinst`, `debian/prerm`, `debian/postinst`, `debian/postrm` (page 148)) to register and un-register the templates during package installation and removal.

<sup>6</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/ucr/papersize/>

```
#!/usr/bin/make -f
%:
    dh $@ --with ucr
```

**Note**

Tabulators must be used for indentation in this **Makefile**-type file.

**debian/control**

The automatically generated dependency on **univention-config** is inserted by **univention-install-config-registry** through `debian/papersize.substvars`.

```
Source: papersize
Section: univention
Priority: optional
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends:
    debhelper-compat (= 13),
    univention-config-dev (>= 15.0.3),
Standards-Version: 4.3.0.3

Package: papersize
Architecture: all
Depends: ${misc:Depends}
Description: An example package to configure the papersize
 This purpose of this package is to show how Univention Config
 Registry is used.
 .
 For more information about UCS, refer to:
 https://www.univention.de/
```

### 3.5.2 Multifile example

This example provides templates for `/etc/hosts.allow` and `/etc/hosts.deny`, which is used to control access to system services. See [hosts\\_access.5](#)<sup>7</sup> for more details.

Source code: UCS source: [doc/developer-reference/ucr/hosts](#)<sup>8</sup>

**conffiles/etc/hosts.allow.d/00header; conffiles/etc/hosts.deny.d/00header**

The first file fragment of the file. It starts with `@%@UCRWARNING=# @%@`, which is replaced by the warning text and a list of all sub-files.

```
@%@UCRWARNING=# @%@
# /etc/hosts.allow: list of hosts that are allowed to access the system.
# See the manual pages hosts_access(5) and hosts_options(5).
```

**conffiles/etc/hosts.allow.d/50dynamic; conffiles/etc/hosts.deny.d/50dynamic**

A second file fragment, which uses Python code to insert access control entries configured through the Univention Configuration Registry Variables `hosts/allow/` and `hosts/deny/`.

```
@!@
for key, value in sorted(configRegistry.items()):
    if key.startswith('hosts/allow/'):
        print(value)
@!@
```

<sup>7</sup> [https://manpages.debian.org/bookworm/libwrap0/hosts\\_access.5.en.html](https://manpages.debian.org/bookworm/libwrap0/hosts_access.5.en.html)

<sup>8</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/ucr/hosts/>

**debian/hosts.univention-config-registry**

The file defines the templates and is processed by `univention-install-config-registry`.

```
Type: multifile
Multifile: etc/hosts.allow

Type: subfile
Multifile: etc/hosts.allow
Subfile: etc/hosts.allow.d/00header

Type: subfile
Multifile: etc/hosts.allow
Subfile: etc/hosts.allow.d/50dynamic
Variables: ^hosts/allow/.*
```

```
Type: multifile
Multifile: etc/hosts.deny

Type: subfile
Multifile: etc/hosts.deny
Subfile: etc/hosts.deny.d/00header

Type: subfile
Multifile: etc/hosts.deny
Subfile: etc/hosts.deny.d/50dynamic
Variables: ^hosts/deny/.*
```

**debian/hosts.univention-config-registry-variables**

The file describes the newly defined Univention Configuration Registry Variables.

```
[hosts/allow/.*]
Description[en]=An permissive access control entry for system services, e.g.
↳"ALL: LOCAL"
Description[de]=Eine erlaubende Zugriffsregel für Systemdienste, z.B. "ALL:↳
↳LOCAL".
Type=str
Categories=service-net

[hosts/deny/.*]
Description[en]=An denying access control entry for system services, e.g.
↳"ALL: ALL".
Description[de]=Eine verbietende Zugriffsregel für Systemdienste, z.B. "ALL:↳
↳ALL".
Type=str
Categories=service-net
```

### 3.5.3 Services

This example provides a template to control the `atd` service through an Univention Configuration Registry Variable `atd/autostart`.

Source code: [UCS source: doc/developer-reference/ucr/service/](https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/ucr/service/)<sup>9</sup>

**conffiles/etc/init.d/atd**

The template replaces the original file with a version, which checks the Univention Configuration Registry Variable `atd/autostart` before starting the `at` daemon. Please note that the “UCRWARNING” is put after the hash-bash line.

<sup>9</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/ucr/service/>

```

#!/bin/sh
@@@UCRWARNING=# @@@
### BEGIN INIT INFO
# Provides:          atd
# Required-Start:    $syslog $time $remote_fs
# Required-Stop:     $syslog $time $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Deferred execution scheduler
# Description:       Debian init script for the atd deferred executions
#                    scheduler
### END INIT INFO
# pidfile: /var/run/atd.pid
#
# Author: Ryan Murray <rmurray@debian.org>
#

PATH=/bin:/usr/bin:/sbin:/usr/sbin
DAEMON=/usr/sbin/atd
PIDFILE=/var/run/atd.pid

test -x "$DAEMON" || exit 0

. /lib/lsb/init-functions

case "$1" in
  start)
    log_daemon_msg "Starting deferred execution scheduler" "atd"
    start_daemon -p "$PIDFILE" "$DAEMON"
    log_end_msg $?
    ;;
  stop)
    log_daemon_msg "Stopping deferred execution scheduler" "atd"
    killproc -p "$PIDFILE" "$DAEMON"
    log_end_msg $?
    ;;
  force-reload|restart)
    "$0" stop
    "$0" start
    ;;
  status)
    status_of_proc -p "$PIDFILE" "$DAEMON" atd && exit 0 || exit $?
    ;;
  *)
    echo "Usage: $0 {start|stop|restart|force-reload|status}"
    exit 1
    ;;
esac

exit 0

```

**Note**

The inclusion of `init-autostart.lib` and use of `check_autostart`.

`debian/service.univention-config-registry`

The file defines the templates.

```
Type: file
File: etc/init.d/atd
Mode: 755
Variables: atd/autostart
```

#### Note

The additional `Mode` statement to mark the file as executable.

#### debian/service.univention-config-registry-variables

The file adds a description for the Univention Configuration Registry Variable `atd/autostart`.

```
[atd/autostart]
Description[en]=Automatically start the AT daemon on system startup [yes]
Description[de]=Automatischer Start des AT-Dienstes beim Systemstart [yes]
Type=bool
Categories=service-at
```

#### debian/service.postinst

Set the Univention Configuration Registry Variable to automatically start the `atd` on new installations.

```
#!/bin/sh

case "$1" in
configure)
    ucr set atd/autostart=yes
    ;;
esac

#DEBHELPER#

exit 0
```

#### debian/control

`univention-base-files` must be added manually as an additional dependency, since it is used from within the shell code.

```
Source: service
Section: univention
Priority: optional
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends:
    debhelper-compat (= 13),
    univention-config-dev (>= 15.0.3),
Standards-Version: 4.3.0.3

Package: service
Architecture: all
Depends: ${misc:Depends},
    univention-base-files,
Description: An example package to configure services
    This purpose of this package is to show how Univention Config
    Registry is used.
    .
    For more information about UCS, refer to:
```

(continues on next page)

(continued from previous page)

```
https://www.univention.de/
```

## DOMAIN JOIN

A UCS system is normally joined into a domain. This establishes a trust relation between the different hosts, which enables users to access services provided by any host of the domain.

Joining a system into a domain requires write permission to create and modify entries in the Univention directory service (LDAP). Local `root` permission on the joining host is not sufficient to get write access to the domain wide LDAP service. Instead valid LDAP credentials must be entered interactively by the administrator doing the join.

### 4.1 Join scripts

Packages requiring write access to the Univention directory service can provide so called join scripts. They are installed into `/usr/lib/univention-install/`. The name of each join script is normally derived from the name of the binary package containing it. It is prefixed with a two-digit number, which is used to order the scripts lexicographically. The filename either ends in `.inst` or `.uinst`, which distinguishes between join script and unjoin script (see *Writing unjoin scripts* (page 37)). The file must have the executable permission bits set.

### 4.2 Join status

For each join script a version number is tracked. This is used to skip re-executing join scripts, which already have been executed. This is mostly a performance optimization, but is also used to find join scripts which need to be run.

The text file `/var/univention-join/status` is used to keep track of the state of all join scripts. For each successful run of a join script a line is appended to that file. That record consists of three space separated entries:

```
$script_name v$version successful
```

1. The first entry contains the name of the join script without the two-digit prefix and without the `.inst` suffix, usually corresponding to the package name.
2. The second entry contains a version number prefixed by a `v`. It is used to keep track of the latest version of the join script, which has been run successfully. This is used to identify, which join scripts need to be executed and which can be skipped, because they were already executed in the past.
3. The third column contains the word `successful`.

If a new version of the join script is invoked, it just appends a new record with a higher version number at the end of the file.

### 4.3 Running join scripts

The following commands related to running join scripts exist:

#### **univention-join**

When `univention-join` is invoked, the machine account is created, if it is missing. Otherwise an already existing account is re-used which allows it to be created beforehand. The distinguished name (dn) of that entry is stored locally in the Univention Configuration Registry Variable `ldap/hostdn`. A random password is generated, which is stored in the file `/etc/machine.secret`.

After that the file `/var/univention-join/status` is cleared and all join scripts located in `/usr/lib/univention-install/` are executed in lexicographical order.

### **univention-run-join-scripts**

This command is similar to `univention-join`, but skips the first step of creating a machine account. Only those join scripts are executed, whose current version is not yet registered in `/var/univention-join/status`.

### **univention-check-join-status**

This command only checks for join scripts in `/usr/lib/univention-install/`, whose version is not yet registered in `/var/univention-join/status`.

When packages are installed, it depends on the server role, if join scripts are invoked automatically from the `postinst` Debian maintainer script or not. This only happens on Primary Directory Node and Backup Directory Node system roles, where the local `root` user has access to the file containing the LDAP credentials. On all other system roles the join scripts need to be run manually by invoking `univention-run-join-scripts` or doing so through UMC.

## 4.4 Writing join scripts

Similar to the Debian maintainer scripts (see *debian/preinst*, *debian/prerm*, *debian/postinst*, *debian/postrm* (page 148)) they should be idempotent. They should transform the system from any state into the state required by the package, that is:

- They should create newly introduced objects in the Univention directory service.
- They should not fail, if the object already exists.
- They should be careful about modifying objects, which might have been modified by the administrator in the past.

### **Important**

Join scripts may be called from multiple system roles and different versions. Therefore, it is important that these scripts **do not destroy or remove data still used by other systems!**

### 4.4.1 Basic join script example

This example provides a template for writing join scripts. The package is called `join-template` and just contains a join and an unjoin script. They demonstrate some commonly used functions.

Source code: [UCS source: doc/developer-reference/join/join-template<sup>10</sup>](https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/join/join-template)

#### **50join-template.inst**

The join script in UCS packages is typically located in the package root directory. It has the following base structure:

```
#!/bin/sh

## joinscript api: bindpwdfile

VERSION=1
. /usr/share/univention-join/joinscripthelper.lib
joinscript_init

SERVICE="MyService"

eval "$(ucr shell)"
```

(continues on next page)

---

<sup>10</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/join/join-template/>

(continued from previous page)

```

. /usr/share/univention-lib/ldap.sh
ucs_addServiceToLocalhost "$SERVICE" "$@" || die

udm "computers/$server_role" modify "$@" \
  --dn "$ldap_hostdn" \
  --set inventoryNumber=1 || die

# create container for extended attributes to be placed in
udm container/cn create "$@" \
  --ignore_exists \
  --position "cn=custom attributes,cn=univention,$ldap_base" \
  --set name="myservice" || die

# some extended attributes would be added here

joinscript_save_current_version
exit 0

```

Note the essential argument "\$@" when **udm** is invoked, which passes on the required LDAP credentials described in *LDAP secrets* (page 41).

Added in version 4.3: Since [UCS 4.3 erratum 85<sup>11</sup>](#), credentials can also be passed through a file to prevent the password from being visible from the process tree.

To enable this API one of the following comments must be placed inside the join script:

#### ## joinscript api: bindpwdfile

The parameters `--binddn` and `--bindpwdfile` pass the credentials for the commands **univention-join** and **univention-run-join-script**. When UCS runs the join script on a Primary Directory Node, it doesn't use these parameters, because the join script has direct access to the credentials.

Deprecated since version 4.4: The old parameter `--bindpwd secret` is no longer supported and used.

Changed in version 5.0: This is the default since UCS 5.

#### ## joinscript api: nocredentials

The credentials will be stored in three files named:

- /var/run/univention-join/binddn
- /var/run/univention-join/bindpwd
- /var/run/univention-join/samba-authentication-file

They exist only while **univention-join** or **univention-run-join-script** are running. Each individual join script will be called with no extra options.

#### debian/control

The package uses two shell libraries, which are described in more detail in *Join script libraries* (page 32). Both packages providing them must be added as additional runtime dependencies.

The package needs to add **univention-join-dev** as build dependency.

```

Source: join-template
Section: univention
Priority: optional
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends:
  debhelper-compat (= 13),

```

(continues on next page)

<sup>11</sup> <https://errata.software-univention.de/#!/?erratum=4.3x85>

(continued from previous page)

```

univention-join-dev (>= 12),
Standards-Version: 4.3.0.3

Package: join-template
Architecture: all
Depends: univention-join (>= 5.0.20-1),
        shell-univention-lib (>= 2.0.17-1),
        ${misc:Depends}
Description: An example package for join scripts
 This purpose of this package is to show how
 Univention Join scripts are used.
 .
 For more information about UCS, refer to:
 https://www.univention.de/

```

**debian/rules**

During package build time `dh-univention-join-install` needs to be called. This should be done using the sequence `univention-join` in `debian/rules`:

```

#!/usr/bin/make -f
%:
    dh $@ --with univention-join

```

This installs the scripts into the right directories. It also adds code fragments to the `.debhelper` files to call them. Those calls are inserted into the Debian maintainer scripts at the location marked with `#DEBHELPER#`. As many join scripts need to restart services, which depend on configuration files managed through Univention Configuration Registry, new Univention Configuration Registry Variable should be set *before* this section.

**4.4.2 Join script exit codes**

Join scripts must return the following exit codes:

**0**

The join script was successful and completed all tasks to join the software package on the system into the domain. All required entries in the Univention directory service were created or do already exist as expected.

The script will be marked as successfully run. As a consequence the join script will not be called again in this version.

**1**

The script did not complete and some tasks to fully join the system into the domain are still pending. Some entries couldn't be created in LDAP or exist in a state, which is incompatible with this version of the package.

The script needs to be run again after fixing the problem, either manually or automatically.

**2**

Some internal functions were called incorrectly. For example the credentials were wrong.

Run the join script again.

**4.4.3 Join script libraries**

The package `univention-join` contains two shell libraries, which provide functions which help in writing join scripts:

**joinscripthelper.lib**

The package contains the shell library `/usr/share/univention-join/joinscripthelper.lib`. It provides functions related to updating the join status file. It is used by the join script itself.

**joinscript\_init**

This function parses the status file and exits the shell script, if a record is found with a version greater or equal to value of the environment variable `VERSION` (page 33). The name of the join script is derived from `$0`.

**joinscript\_save\_current\_version**

This function appends a new record to the end of the status file using the version number stored in the environment variable `VERSION` (page 33).

**joinscript\_check\_any\_version\_executed**

This function returns success (0), if any previous version of the join scripts was successfully executed. Otherwise it returns a failure (1).

**joinscript\_check\_specific\_version\_executed version**

This function returns success (0), if the specified version `version` of the join scripts was successfully executed. Otherwise it returns a failure (1).

**joinscript\_check\_version\_in\_range\_executed min max**

This function returns success (0), if any successfully run version of the join script falls within the range `min..`max``, inclusively. Otherwise it returns a failure (1).

**joinscript\_extern\_init join-script**

The check commands mentioned above can also be used in other shell programs, which are not join scripts. There the name of the join script to be checked must be explicitly given. Instead of calling `joinscript_init`, this function requires an additional argument specifying the name of the `join-script`.

**joinscript\_remove\_script\_from\_status\_file name**

Removes the given join script from the join script status file `/var/univention-join/status`. The name should be the basename of the `joinscript` without the prefixed digits and the suffix `.inst`. So if the `joinscript /var/lib/univention-install/50join-template.inst` shall be removed, one has to run `joinscript_remove_script_from_status_file join-template`. Primarily used in `unjoin` scripts.

**die**

A convenience function to exit the join script with an error code. Used to guarantee that LDAP modifications were successful: `some_udm_create_call || die`

These functions use the following environment variables:

**VERSION**

This variable must be set before `joinscript_init` is invoked. It specifies the version number of the join script and is used twice:

1. It defines the current version of the join script.
2. If that version is already recorded in the status file, the join script qualifies as having been run successfully and the re-execution is prevented. Otherwise the join status is incomplete and the script needs to be invoked again.

The version number should be incremented for a new version of the package, when the join script needs to perform additional modifications in LDAP compared to any previous packaged version.

The version number must be a positive integer. The variable assignment in the join script must be on its own line. It may optionally quote the version number with single quotes (') or double quotes ("). The following assignment are valid:

```
VERSION=1
VERSION='2'
VERSION="3"
```

**JS\_LAST\_EXECUTED\_VERSION**

This variable is initialized by `joinscript_init` with the latest version found in the join status file. If no version of the join script was ever executed and thus no record exists, the variable is set to 0. The join script can use this information to decide what to do on an upgrade.

### join.sh

The package contains the shell library `/usr/share/univention-lib/join.sh`. It is used by Debian maintainer scripts to register and call join scripts. Before UCS 5 the functions were part of `/usr/share/univention-lib/base.sh` provided by the package `shell-univention-lib`.

Since package version `>= 2.0.17-1` it provides the following functions:

**call\_joinscript** [--binddn *bind-dn* [--bindpwdfilename *filename*]] [*XXjoin-script.inst*]

This calls the join script called `XXjoin-script.inst` from the directory `/usr/lib/univention-install/`. The optional LDAP credentials `bind-dn` and `filename` are passed on as-is.

**call\_joinscript\_on\_dcmaster** [--binddn *bind-dn* [--bindpwdfilename *filename*]]

[*XXjoin-script.inst*]

Similar to `call_joinscript`, but also checks the system role and only executes the script on the Primary Directory Node.

**remove\_joinscript\_status** [*name*]

Removes the given join script name from the join script status file `/var/univention-join/status`.

Note that this command does the same as `joinscript_remove_script_from_status_file` provided by `univention-join` (see *joinscripthelper.lib* (page 32)).

**call\_unjoinscript** [--binddn *bind-dn* [--bindpwdfilename *filename*]]

[*XXunjoin-script.uinst*]

Calls the given unjoin script `unjoin-script` on Primary Directory Node and Backup Directory Node systems. The filename must be relative to the directory `/usr/lib/univention-install`. The optional LDAP credentials `bind-dn` and `bind-password` respective filename are passed on as-is. Afterwards the unjoin script is automatically deleted.

**delete\_unjoinscript** [*XXunjoin-script.uinst*]

Deletes the given unjoin script `XXunjoin-script.uinst`, if it does not belong to any package. The filename must be relative to the directory `/usr/lib/univention-install`.

**stop\_udm\_cli\_server**

When `univention-directory-manager` is used the first time a server is started automatically that caches some information about the available modules. When changing some of this information, for example when adding or removing extended attributes, the server should be stopped manually.

### ldap.sh

The package also contains the shell library `/usr/share/univention-lib/ldap.sh`. It provides convenience functions to query the Univention directory service and modify objects. For (un)join scripts the following functions might be important:

**ucs\_addServiceToLocalhost** *servicename* [--binddn *bind-dn* [--bindpwdfilename *filename*]]

Registers the additional service `servicename` in the LDAP object representing the local host. The optional LDAP credentials `bind-dn` and `bind-password` respective filename are passed on as-is.

Listing 4.1: Service registration in join script

```
ucs_addServiceToLocalhost "MyService" "$@"
```

**ucs\_removeServiceFromLocalhost** *servicename* [--binddn *bind-dn* [--bindpwdfilename

*filename*]]

Removes the service `servicename` from the LDAP object representing the local host, effectively reverting an `ucs_addServiceToLocalhost` call. The optional LDAP credentials `bind-dn` and `bind-password` respective filename are passed on as-is.

Listing 4.2: Service un-registration in unjoin script

```
ucs_removeServiceFromLocalhost "MyService" "$@"
```

**ucs\_isServiceUnused** *servicename* [--binddn *bind-dn* [--bindpwdfilename *filename*]]  
Returns 0, if no LDAP host object exists where the service *servicename* is registered with.

Listing 4.3: Check for unused service in unjoin script

```
if ucs_isServiceUnused "MyService" "$@"
then
    uninstall_my_service
fi
```

**ucs\_registerLDAPExtension** [--binddn *bind-dn* --bindpwdfilename *filename*] [--schema *filename.schema* | --acl *filename.acl* | --udm\_syntax *filename.py* | --udm\_hook *filename.py* ...] | --udm\_module *filename.py* [--messagecatalog *filename*] [--umregistration *filename*] [--icon *filename*] } [--packagename *packagename*] [--packageversion *packageversion*] [--name *objectname*] [--ucsversionstart *ucsversion*] [--ucsversionend *ucsversion*]

The shell function **ucs\_registerLDAPExtension** from the Univention shell function library (see *Function libraries* (page 134)) can be used to register several extension in LDAP. This shell function offers several modes:

- schema** <filename>.schema  
Register one or more LDAP schema extension (see *Packaging LDAP Schema Extensions* (page 39))
- acl** <filename>.acl  
Register one or more LDAP access control list (see *Packaging LDAP ACL Extensions* (page 40))
- udm\_syntax** <filename>.py  
Register one or more UDM syntax extension (see *UDM syntax* (page 83))
- udm\_hook** <filename>.py  
Register one or more UDM hook (see *Extended attribute hooks* (page 94))
- udm\_module** <filename>.py  
Register a single UDM module (see *UDM modules* (page 72))

The modes can be combined. If more than one mode is used in one call of the function, the modes are always processed in the order as listed above. Each of these options expects a filename as an required argument.

It is possible to register different extensions to different UCS versions:

- name** <name>  
The option can be used to supply an object name to be used to store the extension. If not set *filename* will be used. If combined with **--udm\_module** (page 35) the name must include a forward slash.
- ucsversionstart** <ucsversion>  
The option can be used to supply the earliest version of UCS to which the UDM extension should be deployed.
- ucsversionend** <ucsversion>  
The option can be used to supply the last version of UCS to which the UDM extension should be deployed. Together with **--ucsversionstart** and **--name**, it is possible to deploy different versions of a UDM extension.

The following options can be given multiple times, but only after the option **--udm\_module** (page 35):

- messagecatalog** <prefix>/<language>.mo  
The option can be used to supply message translation files in GNU message catalog format. The language must be a valid language tag, i.e. must correspond to a subdirectory of `/usr/share/locale/`.

**--umcmessagecatalog** <prefix>/<language>-<module\_id>-<application\_name>.mo

Similar to the option above this option can be used to supply message translation files in GNU message catalog format, but for the UMC. The filename takes the form `language-moduleid.mo`, e.g. `de-udm.mo`, where *language* must be a valid language tag, i.e. must correspond to a subdirectory of `/usr/share/locale/`. The *moduleid* is specified in the UMC registration file (see *UMC module declaration file* (page 105)). The MO files are then placed under `/usr/share/univention-management-console/i18n/` in a subdirectory with the corresponding language short code.

**--umcregistration** <filename>.xml

The option can be used to supply an UMC registration file (see *UMC module declaration file* (page 105)) to make the UDM module accessible through Univention Management Console (UMC).

**--icon** <filename>

The option can be used to supply icon files (PNG or JPEG, in 16×16 or 50×50, or SVGZ).

#### Note

UDM extensions will only be deployed to UCS 5 if either `--ucsversionstart` or `--ucsversionend` are set.

Called from a joinscript, the function automatically determines some required parameters, like the app identifier plus Debian package name and version, required for the creation of the corresponding object. After creation of the object the function waits up to 3 minutes for the Primary Directory Node to signal availability of the new extension and reports success or failure.

For UDM extensions it additionally checks that the corresponding file has been made available in the local file system. Failure conditions may occur e.g. in case the new LDAP schema extension collides with the schema currently active. The Primary Directory Node only activates a new LDAP schema or ACL extension if the configuration check succeeded.

#### Note

The corresponding UDM modules are documented in *Univention Directory Manager (UDM)* (page 71).

Before calling the shell, function the shell variable `UNIVENTION_APP_IDENTIFIER` should be set to the versioned app identifier (and exported to the environment of sub-processes). The shell function will then register the specified app identifier with the extension object to indicate that the extension object is required as long as this app is installed anywhere in the UCS domain.

The options `--packagename` and `--packageversion` should usually not be used, as these parameters are determined automatically. To prevent accidental downgrades the function `ucs_registerLDAPExtension` (as well as the corresponding UDM module) only execute modifications of an existing object if the Debian package version is not older than the previous one.

`ucs_registerLDAPExtension` supports two additional options to specify a valid range of UCS versions, where an extension should be activated. The options are `--ucsversionstart` and `--ucsversionend`. The version check is only performed whenever the extension object is modified. By calling this function from a joinscript, it will automatically update the Debian package version number stored in the object, triggering a re-evaluation of the specified UCS version range. The extension is activated up to and excluding the UCS version specified by `--ucsversionend`. This validity range is not applied to LDAP schema extensions, since they must not be undefined as long as there are objects in the LDAP directory which make use of it.

Listing 4.4: Extension registration in join script

```
$ export UNIVENTION_APP_IDENTIFIER="appID-appVersion" ## example
$ . /usr/share/univention-lib/ldap.sh
```

(continues on next page)

(continued from previous page)

```

$ ucs_registerLDAPExtension "$@" \
  --schema /path/to/appschemaextension.schema \
  --acl /path/to/appaclextension.acl \
  --udm_syntax /path/to/appudmsyntax.py

$ ucs_registerLDAPExtension "$@" \
  --udm_module /path/to/appudmmodule.py \
  --messagecatalog /path/to/de.mo \
  --messagecatalog /path/to/eo.mo \
  --umcregistration /path/to/module-object.xml \
  --icon /path/to/moduleicon16x16.png \
  --icon /path/to/moduleicon50x50.png

```

```

ucs_unregisterLDAPExtension [--binddn bind-dn --bindpwdfilename filename] { --schema
objectname | --acl objectname | --udm_syntax objectname | --udm_hook objectname |
--udm_module objectname ...}

```

There is a corresponding `ucs_unregisterLDAPExtension` function, which can be used to un-register extension objects. This only works if no app is registered any longer for the object. It must not be called unless it has been verified that no object in LDAP still requires this schema extension. For this reason it should generally not be called in unjoin scripts.

Listing 4.5: Schema un-registration in unjoin script

```

. /usr/share/univention-lib/ldap.sh
ucs_unregisterLDAPExtension "$@" --schema appschemaextension

```

## 4.5 Writing unjoin scripts

On package removal, packages should clean up the data in Univention directory service. Removing data from LDAP also requires appropriate credentials, while removing a package only requires local `root` privileges. Therefore, UCS provides support for so-called unjoin scripts. In most cases it reverts the changes of a corresponding join script.

### Warning

A domain is a distributed system. Just because one local system no longer wants to store some information in Univention directory service does not mean that the data should be deleted. There might still be other systems in the domain that still require the data.

Therefore, *the first system to come* should setup the data, while only *the last system to go* may clean up the data.

Just like join scripts an unjoin script is prefixed with a two-digit number for lexicographical ordering. To reverse the order of the unjoin scripts in comparison to the corresponding join scripts, the number of the unjoin script should be 100 minus the number of the corresponding join script. The suffix of an unjoin script is `.uinst` and it should be installed in `/usr/lib/univention-uninstall/`.

On package removal the unjoin script would be deleted as well, while the Univention directory service might still contain data managed by the package. Therefore, the script must be copied from there to `/usr/lib/univention-install/` in the `prepm` maintainer script.

### Example:

The package `univention-fetchmail` provides both a join script `/usr/lib/univention-install/91univention-fetchmail.inst` and the corresponding unjoin script as `/usr/lib/univention-uninstall/09univention-fetchmail.uinst`.

As of UCS 3.1 `.inst` and `.uinst` are not distinguishable in the *UMC Join module* by the user. Internally join scripts are always executed before unjoin scripts and then ordered lexicographically by their prefix.

To decide if an unjoin script is the last instance and should remove the data from LDAP, a service can be registered for each host where the package is installed.

For example the package `univention-fetchmail` uses `ucs_addServiceFromLocalhost "Fetchmail" "$@"` in the join script to register and `ucs_removeServiceFromLocalhost "Fetchmail" "$@"` in the unjoin script to un-register a service for the host. The data is removed from LDAP, when in the unjoin script `ucs_isServiceUnused "Fetchmail" "$@"` returns 0. As a side effect adding the service also allows using this information to find and list those servers currently providing the Fetchmail service.

### 50join-template.uinst

This unjoin script reverts the changes of the join script from *Basic join script example* (page 30).

```
#!/bin/sh

## joinscript api: bindpwdfile

# VERSION is needed for some tools to recognize that as a join script
VERSION=1
. /usr/share/univention-join/joinscripthelper.lib
joinscript_init

SERVICE="MyService"

eval "$(ucr shell)"

. /usr/share/univention-lib/ldap.sh
ucs_removeServiceFromLocalhost "$SERVICE" "$@" || die
if ucs_isServiceUnused "$SERVICE" "$@"
then
    # was last server to implement service. now the data
    # may be removed
    univention-directory-manager container/cn remove "$@" --dn \
        "cn=myservice,cn=custom attributes,cn=univention,$ldap_base" || die

    # Terminate UDM server to force module reload
    . /usr/share/univention-lib/base.sh
    stop_udm_cli_server
fi

# do NOT call "joinscript_save_current_version"
# otherwise an entry will be appended to /var/univention-join/status
# instead the join script needs to be removed from the status file
joinscript_remove_script_from_status_file join-template

exit 0
```

## LIGHTWEIGHT DIRECTORY ACCESS PROTOCOL (LDAP) IN UCS

An LDAP server provides authenticated and controlled access to directory objects over the network. LDAP objects consist of a collection of attributes which conform to so called LDAP schemata. An in depth documentation of LDAP is beyond the scope of this document. Other resources cover this topic exhaustively, for example <https://www.zytrax.com/books/ldap/> or the manual pages `slapd.conf.5`<sup>12</sup> and `slapd.access.5`<sup>13</sup>.

At least it should be noted that OpenLDAP offers two fundamentally different tool sets for direct access or modification of LDAP data:

1. The `slap*` commands (`slapcat`, etc.) are very low level, operating directly on the LDAP backend data and should only be used in rare cases, usually with the LDAP server not running.
2. The `ldap*` commands (`ldapsearch`, etc.) on the other hand are the proper way to perform LDAP operations from the command line and their functionality can equivalently be used from all major programming languages.

On top of the raw LDAP layer, the Univention Directory Manager offers an object model on a higher level, featuring advanced object semantics (see *Univention Directory Manager (UDM)* (page 71)). That level is usually appropriate for app developers, which should be considered before venturing down to the level of direct LDAP operations. On the other hand, for the development of new UDM extensions it is important to understand some of the essential concepts of LDAP as used in UCS.

One essential trait of LDAP as used in UCS, is the strict enforcement of LDAP schemata. An LDAP server refuses to start if an unknown LDAP attribute is referenced either in the configuration or in the backend data. This makes it critically important to install schemata on all systems. To simplify this task UCS features a built-in mechanism for automatic schema replication to all UCS hosted LDAP servers in the UCS domain (see *Univention Directory Listener* (page 43)). The schema replication mechanism is triggered by installation of a new schema extension package on the Primary Directory Node. For redundancy it is strongly recommended to install schema extension packages also on each Backup Directory Node. This way, a Backup Directory Node can replace a Primary Directory Node in case the Primary Directory Node needs to be replaced for some reason. To simplify these tasks even further, UCS offers methods to register new LDAP schemata and associated LDAP ACLs from any UCS system.

### 5.1 Packaging LDAP Schema Extensions

For some purposes, for example for app installation, it is convenient to be able to register a new LDAP schema extension from any system in the UCS domain. For this purpose, the schema extension can be stored as a special type of UDM object. The module responsible for this type of objects is called `settings/ldapschema`. As these objects are replicated throughout the UCS domain, the Primary Directory Node and Backup Directory Node systems listen for modifications of these objects and integrate them into their local LDAP schema directory. As noted above, this simplifies the task of keeping the schema on the Backup Directory Node systems up to date with those on the Primary Directory Node.

The commands to create the LDAP schema extension objects in UDM may be put into any join script (see *Domain join* (page 29)). A LDAP schema extension object is created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. LDAP schema extension objects can be stored anywhere in the LDAP directory, but the recommended location would be `cn=ldapschema,cn=univention`, below the LDAP

---

<sup>12</sup> <https://manpages.debian.org/bookworm/slapd/slapd.conf.5.en.html>

<sup>13</sup> <https://manpages.debian.org/bookworm/slapd/slapd.access.5.en.html>

base. Since the join script creating the attribute may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The UDM module `settings/ldapschema` requires several parameters:

**name (required)**

Name of the schema extension.

**data (required)**

The actual schema data in `bzip2` and `base64` encoded format.

**filename (required)**

The file name the schema should be written to on Primary Directory Node and Backup Directory Node. The filename must not contain any path elements.

**package (required)**

Name of the Debian package which creates the object.

**packageversion (required)**

Version of the Debian package which creates the object. For object modifications the version number needs to increase unless the package name is modified as well.

**appid (optional)**

The identifier of the app which creates the object. This is important to indicate that the object is required as long as the app is installed anywhere in the UCS domain. Defaults to `string`.

**active (internal)**

A boolean flag used internally by the Primary Directory Node to signal availability of the schema extension (default: `FALSE`).

Since many of these parameters are determined automatically by the `ucs_registerLDAPExtension` (page 35) shell library function, it is recommended to use the shell library function to create these objects (see `join.sh` (page 34)).

Listing 5.1: Schema registration in join script

```
export UNIVENTION_APP_IDENTIFIER="appID-appVersion" ## example
. /usr/share/univention-lib/ldap.sh

ucs_registerLDAPExtension "$@" \
  --schema /path/to/appschemaextension.schema
```

## 5.2 Packaging LDAP ACL Extensions

For some purposes, for example for app installation, it is convenient to be able to register a new LDAP ACL extension from any system in the UCS domain. For this purpose, the UCR template for an ACL extension can be stored as a special type of UDM object. The module responsible for this type of objects is called `settings/ldapacl`. As these objects are replicated throughout the UCS domain, the Primary Directory Node, Backup Directory Node and Replica Directory Node systems listen for modifications on these objects and integrate them into the local LDAP ACL UCR template directory. This simplifies the task of keeping the LDAP ACLs on the Backup Directory Node systems up to date with those on the Primary Directory Node.

The commands to create the LDAP ACL extension objects in UDM may be put into any join script (see *Domain join* (page 29)). A LDAP ACL extension object is created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. LDAP ACL extension objects can be stored anywhere in the LDAP directory, but the recommended location would be `cn=ldapacl,cn=univention`, below the LDAP base. Since the join script creating the attribute may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The UDM module `settings/ldapacl` requires several parameters:

**name (required)**

Name of the ACL extension.

**data (required)**

The actual ACL UCR template data in `bzip2` and `base64` encoded format.

**filename (required)**

The filename the ACL UCR template data should be written to on Primary Directory Node, Backup Directory Node and Replica Directory Node. The filename must not contain any path elements.

**package (required)**

Name of the Debian package which creates the object.

**packageversion (required)**

Version of the Debian package which creates the object. For object modifications the version number needs to increase unless the package name is modified as well.

**appid (optional)**

The identifier of the app which creates the object. This is important to indicate that the object is required as long as the app is installed anywhere in the UCS domain. Defaults to `string`.

**ucsversionstart (optional)**

Minimal required UCS version. The UCR template for the ACL is only activated by systems with a version higher than or equal to this.

**ucsversionend (optional)**

Maximal required UCS version. The UCR template for the ACL is only activated by systems with a version lower or equal than this. To specify validity for the whole 4.1-x release range a value like `4.1-99` may be used.

**active (internal)**

A boolean flag used internally by the Primary Directory Node to signal availability of the ACL extension on the Primary Directory Node (default: `FALSE`).

Since many of these parameters are determined automatically by the `ucs_registerLDAPExtension` (page 35) shell library function, it is recommended to use the shell library function to create these objects (see `join.sh` (page 34)).

Listing 5.2: LDAP ACL registration in join script

```
export UNIVENTION_APP_IDENTIFIER="appID-appVersion" ## example
. /usr/share/univention-lib/ldap.sh

ucs_registerLDAPExtension "$@" \
  --acl /path/to/appaclextension.acl
```

## 5.3 LDAP secrets

The credentials for different UCS domain accounts are stored in plain-text files on some UCS systems. The files are named `/etc/*.secret`. They are owned by the user `root` and allow read-access for different groups.

**/etc/ldap.secret for cn=admin, ldap\_base**

This account has full write access to all LDAP entries. The file is only available on Primary Directory Node and Backup Directory Node systems and is owned by the group `DC Backup Hosts`.

**/etc/machine.secret for ldap/hostdn**

Each host uses its account to get at least read-access to LDAP. Directory Nodes, for example Domain controllers, in the container `cn=dc, cn=computers, ldap_base` get additional rights to access LDAP attributes. The file is available on all joined system roles and is readable only by the local `root` user and group.

During package installation, only the maintainer scripts (see `debian/preinst`, `debian/prerm`, `debian/postinst`, `debian/postrm` (page 148)) on Primary Directory Node and Backup Directory Node can use their `root` permission to directly read `/etc/ldap.secret`. Thus only on those roles, the join scripts get automatically executed when the package is installed. On all other system roles, the join scripts need to be executed manually. This can either be done through the *UMC Join module* or through the command line tool `univention-run-join-scripts`. Both methods require appropriate credentials.

### 5.3.1 Password change

To reconfirm the trust relation between UCS systems, computers need to regularly change the password associated with the machine account. This is controlled through the Univention Configuration Registry Variable `server/password/change`<sup>14</sup>. For UCS servers this is evaluated by the script `/usr/lib/univention-server/server_password_change`, which is invoked nightly at 01:00 by `cron`.<sup>8</sup><sup>15</sup>. The interval is controlled through a second Univention Configuration Registry Variable `server/password/interval`<sup>16</sup>, which defaults to 21 days.

The password is stored in the plain text file `/etc/machine.secret`. Many long running services read these credentials only on startup, which breaks when the password is changed while they are still running. Therefore, UCS provides a mechanism to invoke arbitrary commands, when the machine password is changed. This can be used for example to restart specific services.

Hook scripts should be placed in the directory `/usr/lib/univention-server/server_password_change.d/`. The name must consist of only digits, upper and lower ASCII characters, hyphens and underscores. The file must be executable and is called through `run-parts`.<sup>8</sup><sup>17</sup>. It receives one argument, which is used to distinguish three phases:

1. Each script will be called with the argument `prechange` before the password is changed. If any script terminates with an exit status unequal zero, the change is aborted.
2. A new password is generated locally using `makepasswd`.<sup>1</sup><sup>18</sup>. It's changed in the Univention directory service through UDM and stored in `/etc/machine.secret`. The old password is logged in `/etc/machine.secret.old`.

If anything goes wrong in this step, the change is aborted and the changes need to be rolled back.

3. All hook scripts are called again.
  - If the password change was successful, `postchange` gets passed to the hook scripts. This should complete any change prepared in the `prechange` phase.
  - If the password change failed for any reason, all hook scripts are called with the argument `nochange`. This should undo any action already done in the `prechange` phase.

Install this file to `/usr/lib/univention-server/server_password_change.d/`.

Listing 5.3: Server password change example

```
#!/bin/sh
case "$1" in
prechange)
    # nothing to do before the password is changed
    exit 0
    ;;
nochange)
    # nothing to do after a failed password change
    exit 0
    ;;
postchange)
    # restart daemon after password was changed
    deb-systemd-invoke restart my-daemon
    ;;
esac
```

`init-scripts` should only be invoked indirectly through `deb-systemd-invoke`.<sup>1</sup><sup>19</sup>. This is required for `chroot` environments and allows the policy layer to control starting and stopping in certain special situations like during an system upgrade.

<sup>14</sup> <https://docs.software-univention.de/manual/5.2/en/appendix/variables.html#envvar-server-password-change>

<sup>15</sup> <https://manpages.debian.org/bookworm/cron/cron.8.en.html>

<sup>16</sup> <https://docs.software-univention.de/manual/5.2/en/appendix/variables.html#envvar-server-password-interval>

<sup>17</sup> <https://manpages.debian.org/bookworm/debianutils/run-parts.8.en.html>

<sup>18</sup> <https://manpages.debian.org/bookworm/makepasswd/makepasswd.1.en.html>

<sup>19</sup> <https://manpages.debian.org/bookworm/init-system-helpers/deb-systemd-invoke.1p.en.html>

## UNIVENTION DIRECTORY LISTENER

Replication of the directory data within a UCS domain is provided by the Univention Directory Listener/Notifier mechanism:

- The Univention Directory Listener service runs on all UCS systems.
- On the Primary Directory Node (and possibly existing Backup Directory Node systems) the Univention Directory Notifier service monitors changes in the LDAP directory and makes the selected changes available to the Univention Directory Listener services on all UCS systems joined into the domain.

The active Univention Directory Listener instances in the domain connect to a Univention Directory Notifier service. If an LDAP change is performed on the Primary Directory Node (all other LDAP servers in the domain are read-only), this is registered by the Univention Directory Notifier and reported to the listener instances.

Each Univention Directory Listener instance hosts a range of Univention Directory Listener modules. These modules are shipped by the installed applications; the print server package includes, for example, listener modules which generate the CUPS configuration.

Univention Directory Listener modules can be used to communicate domain changes to services which are not LDAP-aware. The print server CUPS is an example of this: The printer definitions are not read from the LDAP, but instead from the file `/etc/cups/printers.conf`. Now, if a printer is saved in the printer management of the Univention Management Console, it is stored in the LDAP directory. This change is detected by the Univention Directory Listener module `cups-printers` and an entry gets added to, modified in or deleted from `/etc/cups/printers.conf` based on the modification in the LDAP directory.

By default the Listener loads all modules from the directory `/usr/lib/univention-directory-listener/system/`. Other directories can be specified using the option `-m` when starting the `univention-directory-listener` daemon.

### 6.1 Structure of Listener Modules

Listener modules can be implemented using the *High-level Listener modules API* (page 47) or the *Low-level Listener module* (page 52).

Added in version 4.2: New implementations should be based on the newer high-level API, which is available since [UCS 4.2 erratum 311](#)<sup>20</sup>.

Each listener module must declare several string constants. They are required by the Univention Directory Listener to handle each module.

```
your_module.name: str21
```

(optional)

This name is used to uniquely identify the module. It defaults to the filename containing this listener module without the `.py` suffix. The name is used to keep track of the module state in `/var/lib/univention-directory-listener/handlers/`.

---

<sup>20</sup> <https://errata.software-univention.de/#!/?erratum=4.2x311>

<sup>21</sup> <https://docs.python.org/3/library/stdtypes.html#str>

`your_module.get_name()`

**Return type**

`str`<sup>22</sup>

For description, see *name* (page 43).

`your_module.description: str`<sup>23</sup>

(required)

A short description. It is currently unused and displayed nowhere.

`your_module.get_description()`

**Return type**

`str`<sup>24</sup>

For description, see *description* (page 44).

`your_module.filter: str`<sup>25</sup>

(required)

Defines a LDAP filter which is used to match the objects the listener is interested in. This filter is similar to the LDAP search filter as defined in [RFC 2254](https://datatracker.ietf.org/doc/html/rfc2254)<sup>26</sup>, but more restricted:

- it is case sensitive
- it only supports equal matches

`your_module.get_ldap_filter()`

**Return type**

`str`<sup>27</sup>

For description, see *filter* (page 44).

`your_module.ldap_filter: str`<sup>28</sup>

(high-level API)

For description, see *filter* (page 44).

`your_module.attributes: list`<sup>29</sup> [`str`<sup>30</sup>]

(optional)

A Python list of LDAP attribute names which further narrows down the condition under which the listener module gets called. By default the module is called on all attribute changes of objects matching the filter. If the list is specified, the module is only invoked when at least one of the listed attributes is changed.

`your_module.get_attributes()`

**Return type**

`list`<sup>31</sup> [`str`<sup>32</sup>]

For description, see *attributes* (page 44).

---

<sup>22</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>23</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>24</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>25</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>26</sup> <https://datatracker.ietf.org/doc/html/rfc2254.html>

<sup>27</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>28</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>29</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>30</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>31</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>32</sup> <https://docs.python.org/3/library/stdtypes.html#str>

`your_module.modrdn: str`<sup>33</sup>

(low-level API, optional)

Setting this variable to the string `1` changes the signature of the function `handler()`. It receives an additional `forth` argument, which specifies the LDAP operation triggering the change.

`your_module.handle_every_delete: bool`<sup>34</sup>

(low-level API, optional)

The Listener uses its *cache* (page 65) to keep track of objects, especially their previous values and which modules handles which objects. The Univention Configuration Registry Variable `listener/cache/filter` can be used to prevent certain objects from being stored in the cache. But then the Listener no longer knows which module must be called when such an object is deleted. Setting this variable to `True` will make the Listener call the function `handler()` of this module whenever any object is deleted. The function then must use other means to determine itself if the deleted object is of the appropriate type as `old` will be empty `dict`.

`your_module.priority: float`<sup>35</sup>

(optional)

This variable can be used to explicitly overwrite the default order in which the modules are executed. By default modules are executed in random order. `replication.py` defaults to `0.0` as it must be executed first, all other modules default to `50.0`.

`your_module.get_priority()`

#### Return type

`float`<sup>36</sup>

For description, see *priority* (page 45).

## 6.1.1 Handle LDAP objects

For handling changes to matching LDAP objects a *handler* must be implemented. This function is called for different events:

- when the object is first created.
- when attributes of an existing object are changed.
- when the object is moved to a different location within the LDAP tree.
- when the object is finally removed.
- when a LDAP schema change happens.

The low-level API requires writing a single function `handler()` to handle all those cases. The high-level API already splits this into separate methods `create()`, `modify()` and `remove()`, which are easier to overwrite.

## 6.1.2 Initialize and clean

Each module gets initialized once by calling its function `initialize()` (page 45). This state of each module is tracked in a file below `/var/lib/univention-directory-listener/handlers/`.

`your_module.initialize()`

#### Return type

`None`

(optional)

<sup>33</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>34</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>35</sup> <https://docs.python.org/3/library/functions.html#float>

<sup>36</sup> <https://docs.python.org/3/library/functions.html#float>

The function `initialize()` (page 45) is called once when the Univention Directory Listener loads the module for the first time. This is recorded persistently in the file `/var/lib/univention-directory-listener/name`, where `name` equals the value from the header.

If for whatever reason the listener module should be reset and re-run for all matching objects, the state can be reset by running the following command:

```
$ univention-directory-listener-ctrl resync $name
```

In that case the function `initialize()` (page 45) will be called again.

```
your_module.clean()
```

### Return type

None

(optional)

The function `clean()` (page 46) is only called when the Univention Directory Listener is initialized for the first time or is forced to *re-initialize from scratch* using the option `-g`, `-i`, or `-P`. The function should purge all previously generated files and return the module into a clean state.

Afterwards the module will be re-initialized by calling the function `initialize()` (page 45).

### 6.1.3 Suspend and resume

For efficiency reasons the API provides two functions, which resumes and suspends modules when no transactions are processed for 15 seconds. All modules start in the state `suspended`. Before a `suspended` module is called to handle a change, the function `prerun()` (page 46) is called for that module. If no transactions happen within a time span of 15 seconds the Listener suspends all active modules by calling the function `postrun()` (page 46). This mechanism is most often used to batch changes by collecting multiple changes and applying them only on suspend.

```
your_module.prerun()
```

### Return type

None

(optional);

For optimization the Univention Directory Listener does not keep open an LDAP connection all time. Instead the connection is opened once at the beginning of a change and closed only if no new change arrives within 15 seconds. The opening is signaled by the invocation of the function `prerun()` (page 46) and the closing by `postrun()` (page 46).

The function `postrun()` (page 46) is most often used to restart services, as restarting a service takes some time and makes the service unavailable during that time. It's best practice to use the `handler()` only to process the stream of changes, set UCR variables or generate new configuration files. Restarting associated services should be delayed to the `postrun()` (page 46) function.

```
your_module.postrun()
```

### Return type

None

For description, see `prerun()` (page 46).

#### Warning

The function `postrun()` (page 46) is only called, when no change happens for 15 seconds. This is not on a per-module basis, but globally. In an ever changing system, where the stream of changes never pauses for 15 seconds, the functions may never be called!

## 6.2 High-level Listener modules API

Univention Directory Listener ships with a template in UCS source: `management/univention-directory-listener/examples/listener_module_template.py`<sup>37</sup>. This should be used as a starting point for new modules. The more complex example in UCS source: `management/univention-directory-listener/examples/complex_handler.py`<sup>38</sup> can also be used.

Alternatively the implementation can start from scratch:

1. Create a subclass of `univention.listener.ListenerModuleHandler`.
2. Add an inner class called `Configuration` which at least has the attributes `name` (page 47), `description` (page 47) and `ldap_filter` (page 47).

The inner class `Configuration` is used to configure global module settings. For most properties a corresponding method exists, which just returns the value of the property by default. The methods can be overwritten if values should be computed once on module load.

`high_level.get_name()`

### Return type

`str`<sup>39</sup>

(optional)

The internal name of the handler, see `name` (page 43).

`high_level.name: str`<sup>40</sup>

The internal name of the handler, see `name <your_module.name<`.

`high_level.get_description()`

A descriptive text, see `description` (page 44).

`high_level.description: str`<sup>41</sup>

A descriptive text, see `description` (page 44).

`high_level.get_ldap_filter()`

The LDAP filter string, see `filter` (page 44).

`high_level.ldap_filter: str`<sup>42</sup>

The LDAP filter string, see `filter` (page 44).

`high_level.get_attributes()`

The list of attributes, for when they are changed, the module is called; see `attributes` (page 44).

`high_level.attributes: str`<sup>43</sup>

The list of attributes, for when they are changed, the module is called; see `attributes` (page 44).

`high_level.get_priority()`

The priority for ordering; see `priority` (page 45).

`high_level.priority: float`<sup>44</sup>

The priority for ordering; see `priority` (page 45).

<sup>37</sup> [https://github.com/univention/univention-corporate-server/blob/5.2-5/management/univention-directory-listener/examples/listener\\_module\\_template.py](https://github.com/univention/univention-corporate-server/blob/5.2-5/management/univention-directory-listener/examples/listener_module_template.py)

<sup>38</sup> [https://github.com/univention/univention-corporate-server/blob/5.2-5/management/univention-directory-listener/examples/complex\\_handler.py](https://github.com/univention/univention-corporate-server/blob/5.2-5/management/univention-directory-listener/examples/complex_handler.py)

<sup>39</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>40</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>41</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>42</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>43</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>44</sup> <https://docs.python.org/3/library/functions.html#float>

```
high_level.get_listener_module_instance()
```

**Return type**

ListenerModuleHandler

This creates an instance of the handler module and returns it.

```
high_level.get_listener_module_class()
```

**Return type**

`list`<sup>45</sup>[ListenerModuleHandler]

(optional)

Class that implements the module. Will be set automatically by the handlers meta-class.

```
high_level.listener_module_class: list46[ListenerModuleHandler]
```

```
high_level.get_active()
```

**Return type**

`bool`<sup>47</sup>

This returns the value of the Univention Configuration Registry Variable `listener/module/name/deactivate` as a boolean. Setting the variable to `False` will prevent the module from being called for all changes.

**Note**

Re-enabling the module will not result in the module being called for all previously missed changes. For this the module must be fully resynchronized.

The handler itself should inherit from `univention.listener.ListenerModuleHandler` and then overwrite some methods to provide its own implementation:

```
high_level.create(dn: str48, new: dict49[str50, list51[bytes52]])
```

**Parameters**

- `dn` (`str`<sup>53</sup>)
- `new` (`dict`<sup>54</sup>[`str`<sup>55</sup>, `list`<sup>56</sup>[`bytes`<sup>57</sup>]])

**Return type**

None

Called when a new object was created.

```
high_level.modify(dn: str58, new: dict59[str60, list61[bytes62]], old: dict63[str64, list65[bytes66]], old_dn: str67 | None68)
```

**Parameters**

- `dn` (`str`<sup>69</sup>)

<sup>45</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>46</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>47</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>48</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>49</sup> <https://docs.python.org/3/library/stdtypes.html#dict>

<sup>50</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>51</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>52</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

<sup>53</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>54</sup> <https://docs.python.org/3/library/stdtypes.html#dict>

<sup>55</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>56</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>57</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

- `new(dict70[str71, list72[bytes73]])`
- `old(dict74[str75, list76[bytes77]])`
- `old_dn(str78 | None)`

**Return type**

None

Called when a new object was modified or moved. In case of a move `old_dn` is set. During a move attributes may be modified, too.

```
high_level.remove(dn: str79, old: dict80[str81, list82[bytes83]])
```

**Parameters**

- `dn(str84)`
- `old(dict85[str86, list87[bytes88]])`

**Return type**

None

Called when a new object was deleted.

```
high_level.initialize()
```

**Return type**

None

Called once when the module is not initialized yet.

```
high_level.clean()
```

**Return type**

None

Called once before a module is resynchronized.

---

<sup>58</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>59</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>60</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>61</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>62</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>63</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>64</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>65</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>66</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>67</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>68</sup> <https://docs.python.org/3/library/constants.html#None>  
<sup>69</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>70</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>71</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>72</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>73</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>74</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>75</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>76</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>77</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>78</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>79</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>80</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>81</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>82</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>83</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>84</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>85</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>86</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>87</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>88</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

`high_level.pre_run()`

**Return type**

None

Called once each time before a batch of transactions is processed.

`high_level.post_run()`

**Return type**

None

Called once each time after a batch of transactions is processed.

In addition to those handler functions the class also provides several convenience functions:

`high_level.as_root()`

**Return type**

None

A context manager to temporarily change the effective UID of the current to 0. Also see `listener.SetUID()` described in *User-ID and Credentials* (page 65).

`high_level.diff` (*old*: `dict`<sup>89</sup>[`str`<sup>90</sup>, `list`<sup>91</sup>[`bytes`<sup>92</sup>]], *new*: `dict`<sup>93</sup>[`str`<sup>94</sup>, `list`<sup>95</sup>[`bytes`<sup>96</sup>]], *keys*: `Iterable`<sup>97</sup>[`str`<sup>98</sup>] | `None`<sup>99</sup>, *ignore\_metadata*: `bool`<sup>100</sup>)

**Parameters**

- `old` (`dict`<sup>101</sup>[`str`<sup>102</sup>, `list`<sup>103</sup>[`bytes`<sup>104</sup>]])
- `new` (`dict`<sup>105</sup>[`str`<sup>106</sup>, `list`<sup>107</sup>[`bytes`<sup>108</sup>]])
- `keys` (`Iterable`<sup>109</sup>[`str`<sup>110</sup>] | `None`)
- `ignore_metadata` (`bool`<sup>111</sup>)

**Return type**

`dict`<sup>112</sup>[`str`<sup>113</sup>, `tuple`<sup>114</sup>[`list`<sup>115</sup>[`bytes`<sup>116</sup>] | `None`], `list`<sup>117</sup>[`bytes`<sup>118</sup>] | `None`]]

Calculate difference between old and new LDAP attributes. By default all attributes are compared, but this can be limited by naming them via `keys`. By default *operational attributes* are excluded unless `ignore_metadata` is enabled.

`high_level.error_handler` (*dn*: [str](#)<sup>119</sup>, *old*: [dict](#)<sup>120</sup> [[str](#)<sup>121</sup>, [list](#)<sup>122</sup> [[bytes](#)<sup>123</sup>]], *new*: [dict](#)<sup>124</sup> [[str](#)<sup>125</sup>, [list](#)<sup>126</sup> [[bytes](#)<sup>127</sup>]], *command*: [str](#)<sup>128</sup>, *exc\_type*: [Type](#)<sup>129</sup> [[BaseException](#)<sup>130</sup>] | [None](#)<sup>131</sup>, *exc\_value*: [BaseException](#)<sup>132</sup> | [None](#)<sup>133</sup>, *exc\_traceback*: [types.TracebackType](#)<sup>134</sup> | [None](#)<sup>135</sup>)

### Parameters

- **dn** ([str](#)<sup>136</sup>)
- **old** ([dict](#)<sup>137</sup> [[str](#)<sup>138</sup>, [list](#)<sup>139</sup> [[bytes](#)<sup>140</sup>]])
- **new** ([dict](#)<sup>141</sup> [[str](#)<sup>142</sup>, [list](#)<sup>143</sup> [[bytes](#)<sup>144</sup>]])
- **command** ([str](#)<sup>145</sup>)
- **exc\_type** ([Type](#)<sup>146</sup> [[BaseException](#)<sup>147</sup>] | [None](#))
- **exc\_value** ([BaseException](#)<sup>148</sup> | [None](#))
- **exc\_traceback** ([types.TracebackType](#)<sup>149</sup> | [None](#))

### Return type

[None](#)

This method will be called for unhandled exceptions in create/modify/remove. By default it logs the exception and re-raises it.

<sup>89</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>90</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>91</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>92</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>93</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>94</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>95</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>96</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>97</sup> <https://docs.python.org/3/library/typing.html#typing.Iterable>  
<sup>98</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>99</sup> <https://docs.python.org/3/library/constants.html#None>  
<sup>100</sup> <https://docs.python.org/3/library/functions.html#bool>  
<sup>101</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>102</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>103</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>104</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>105</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>106</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>107</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>108</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>109</sup> <https://docs.python.org/3/library/typing.html#typing.Iterable>  
<sup>110</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>111</sup> <https://docs.python.org/3/library/functions.html#bool>  
<sup>112</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>113</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>114</sup> <https://docs.python.org/3/library/stdtypes.html#tuple>  
<sup>115</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>116</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>117</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>118</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

**property** `high_level.lo`: `univention.uldap.access`

This property returns a LDAP connection object to access the local LDAP server.

**property** `high_level.po`: `univention.uldap.position`

This property returns a LDAP position object for the LDAP base DN.

Any instance also has the following variables:

`high_level.logger`: `logging.Logger`<sup>150</sup>

An instance of `logging.Logger`.

`high_level.ucr`: `univention.config_registry.ConfigRegistry`

A reference to the shared instance `listener.configRegistry`.

## 6.3 Low-level Listener module

Each Listener module is implemented as a plain Python module. The required variables and functions must be declared at the module level.

```
description : str = "Module description"
filter      : str = "(!(objectClass=lock))"
attributes  : list[str] = ["objectClass"]
modrdn     : str = "1"
```

On top of the description in *Structure of Listener Modules* (page 43) the following extra notes apply:

`low_level.filter`: `str`<sup>151</sup>

(required)

### Note

- <sup>119</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>120</sup> <https://docs.python.org/3/library/stdtypes.html#dict>
- <sup>121</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>122</sup> <https://docs.python.org/3/library/stdtypes.html#list>
- <sup>123</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>
- <sup>124</sup> <https://docs.python.org/3/library/stdtypes.html#dict>
- <sup>125</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>126</sup> <https://docs.python.org/3/library/stdtypes.html#list>
- <sup>127</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>
- <sup>128</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>129</sup> <https://docs.python.org/3/library/typing.html#typing.Type>
- <sup>130</sup> <https://docs.python.org/3/library/exceptions.html#BaseException>
- <sup>131</sup> <https://docs.python.org/3/library/constants.html#None>
- <sup>132</sup> <https://docs.python.org/3/library/exceptions.html#BaseException>
- <sup>133</sup> <https://docs.python.org/3/library/constants.html#None>
- <sup>134</sup> <https://docs.python.org/3/library/types.html#types.TracebackType>
- <sup>135</sup> <https://docs.python.org/3/library/constants.html#None>
- <sup>136</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>137</sup> <https://docs.python.org/3/library/stdtypes.html#dict>
- <sup>138</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>139</sup> <https://docs.python.org/3/library/stdtypes.html#list>
- <sup>140</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>
- <sup>141</sup> <https://docs.python.org/3/library/stdtypes.html#dict>
- <sup>142</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>143</sup> <https://docs.python.org/3/library/stdtypes.html#list>
- <sup>144</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>
- <sup>145</sup> <https://docs.python.org/3/library/stdtypes.html#str>
- <sup>146</sup> <https://docs.python.org/3/library/typing.html#typing.Type>
- <sup>147</sup> <https://docs.python.org/3/library/exceptions.html#BaseException>
- <sup>148</sup> <https://docs.python.org/3/library/exceptions.html#BaseException>
- <sup>149</sup> <https://docs.python.org/3/library/types.html#types.TracebackType>
- <sup>150</sup> <https://docs.python.org/3/library/logging.html#logging.Logger>

The name `filter` (page 52) has the drawback that it shadows the Python built-in function `filter()`<sup>152</sup>, but its use has historical reasons. If that function is required for implementing the listener module, an alias-reference may be defined before overwriting the name or it may be explicitly accessed through the Python `__builtin__` module.

In addition to the static strings, a module must implement several functions. They are called in different situations of the lifecycle of the module.

```
def initialize() -> None:
    pass
def handler(
    dn: str,
    new: dict[str, list[bytes]],
    old: dict[str, list[bytes]],
    command: str = '',
) -> None:
    pass
def clean() -> None:
    pass
def prerun() -> None:
    pass
def postrun() -> None:
    pass
def setdata(key: str, value: str) -> None:
    pass
```

```
low_level.handler(dn: str153, new: dict154[str155, list156[bytes157]], old: dict158[str159, list160[bytes161]],
                  command: str162 = "")
```

### Parameters

- `dn` (str<sup>163</sup>)
- `new` (dict<sup>164</sup>[str<sup>165</sup>, list<sup>166</sup>[bytes<sup>167</sup>]])
- `old` (dict<sup>168</sup>[str<sup>169</sup>, list<sup>170</sup>[bytes<sup>171</sup>]])
- `command` (str<sup>172</sup>)

### Return type

None

(required)

This function is called for each change matching the `filter` and `attributes` as declared in the header of the module. The distinguished name (`dn`) of the object is supplied as the first argument `dn`.

Depending on the type of modification, `old` and `new` may each independently either be `None` or reference a Python dictionary of lists. Each list represents one of the multi-valued attributes of the object. The Univention Directory Listener uses a local cache to store the values of each object as it has seen most recently. This cache is used to supply the values for `old`, while the values in `new` are those retrieved from that LDAP directory service which is running on the same server as the Univention Directory Notifier (Primary Directory Node or Backup Directory Node servers in the domain).

If and only if the global `modrdn` setting is enabled, `command` is passed as a fourth argument. It contains a single letter, which indicates the original type of modification. This can be used to further distinguish an `modrdn` operation from a delete operation followed by a create operation.

### m (modify)

Signals a modify operation, where an existing object is changed. `old` contains a copy of the previously

<sup>151</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>152</sup> <https://docs.python.org/3.11/library/functions.html#filter>

values from the listener cache. `new` contains the current values as retrieved from the leading LDAP directory service.

#### a (add)

Signals the addition of a new object. `old` is `None` and `new` contains the latest values of the added object.

#### d (delete)

Signals the removal of a previously existing object. `old` contains a copy of the previously cached values, while `new` is `None`.

#### r (rename: modification of distinguished name through `modrdn`)

Signals a change in the distinguished name, which may be caused by renaming the object or moving the object from one container into another. The module is called with this command instead of the `delete` command, so that modules can recognize this special case and avoid deletion of local data associated with the object. The module will be called again with the `add` command just after the `modrdn` command, where it should process the rename or move operation. Each module is responsible for keeping track of the rename-case by internally storing the previous distinguished name during the `modrdn` phase of this two phased operation.

#### n (new or schema change)

This command can signal two changes:

- If `dn` is `cn=Subschema`, it signals that a schema change occurred.
- All other cases signal the creation of a new intermediate object, which should be handled just like a normal `add()` operation. This happens when an object is moved into a new container, which does not yet exist in the local LDAP service.

### Important

The listener only retrieves the latest state and passes it to this function. Due to stopped processes or due to network issues this can lead to multiple changes being aggregated into the first change. This may cause `command` to no longer match the values supplied through `new`. For example, if the object has been deleted in the meantime, the function is called once with `new=None` and `command='m'`. This can also lead to the function being called multiple times with `old` being equal to `new`.

`low_level.setdata(key: str173, value: str174)`

### Parameters

- **key** ([str](#)<sup>175</sup>)
- **value** ([str](#)<sup>176</sup>)

<sup>153</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>154</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>155</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>156</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>157</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>158</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>159</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>160</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>161</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>162</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>163</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>164</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>165</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>166</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>167</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>168</sup> <https://docs.python.org/3/library/stdtypes.html#dict>  
<sup>169</sup> <https://docs.python.org/3/library/stdtypes.html#str>  
<sup>170</sup> <https://docs.python.org/3/library/stdtypes.html#list>  
<sup>171</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>  
<sup>172</sup> <https://docs.python.org/3/library/stdtypes.html#str>

**Return type**

None

(optional)

This function is called up to four times by the Univention Directory Listener main process to pass configuration data into the modules. The following `keys` are supplied in the following order:

**basedn**

The base distinguished name the Univention Directory Listener is using.

**binddn**

The distinguished name the Univention Directory Listener is using to authenticate to the LDAP server (through `simple bind`).

**bindpw**

The password the Univention Directory Listener is using to authenticate to the LDAP server.

**ldapservers**

The hostname of the LDAP server the Univention Directory Listener is currently reading from.

**Note**

It's strongly recommended to avoid initiating LDAP modifications from a listener module. This potentially creates a complex modification dynamic, considering that a module may run on several systems in parallel at their own timing.

## 6.4 Listener tasks and examples

All changes trigger a call to the function `handler()`. For simplicity and readability it is advisable to delegate the different change types to different sub-functions.

### 6.4.1 Listener API example

The following boilerplate code uses the newer listener API.

Source code: UCS source: `management/univention-directory-listener/examples/listener_module_template.py`<sup>177</sup>

```
# SPDX-FileCopyrightText: 2017-2026 Univention GmbH
# SPDX-License-Identifier: AGPL-3.0-only

from univention.listener import ListenerModuleHandler

class ListenerModuleTemplate(ListenerModuleHandler):

    class Configuration(object):
        name = 'unique_name'
        description = 'listener module description'
        ldap_filter = '(&(objectClass=inetOrgPerson)(uid=example))'
        attributes = ['sn', 'givenName']

    def create(self, dn: str, new: dict[str, list[bytes]]) -> None:
```

(continues on next page)

<sup>173</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>174</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>175</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>176</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>177</sup> [https://github.com/univention/univention-corporate-server/blob/5.2-5/management/univention-directory-listener/examples/listener\\_module\\_template.py](https://github.com/univention/univention-corporate-server/blob/5.2-5/management/univention-directory-listener/examples/listener_module_template.py)

(continued from previous page)

```

        self.logger.debug('dn: %r', dn)

    def modify(
        self,
        dn: str,
        old: dict[str, list[bytes]],
        new: dict[str, list[bytes]],
        old_dn: str | None,
    ) -> None:
        self.logger.debug('dn: %r', dn)
        if old_dn:
            self.logger.debug('it is (also) a move! old_dn: %r', old_dn)
            self.logger.debug('changed attributes: %r', self.diff(old, new))

    def remove(self, dn: str, old: dict[str, list[bytes]]) -> None:
        self.logger.debug('dn: %r', dn)

```

## 6.4.2 Basic example

The following boilerplate code delegates each change type to a separate function. It does not handle renames and moves explicitly, but only as the removal of the object at the old dn and the following addition at the new dn.

Source code: [UCS source: doc/developer-reference/listener/simple.py](https://github.com/univention/univention-corporate-server/blob/5.2-5/doc/developer-reference/listener/simple.py)<sup>178</sup>

```

def handler(
    dn: str,
    new: dict[str, list[bytes]],
    old: dict[str, list[bytes]],
) -> None:
    if new and not old:
        handler_add(dn, new)
    elif new and old:
        handler_modify(dn, old, new)
    elif not new and old:
        handler_remove(dn, old)
    else:
        pass # ignore

def handler_add(dn: str, new: dict[str, list[bytes]]) -> None:
    """Handle addition of object."""
    pass # replace this

def handler_modify(
    dn: str,
    old: dict[str, list[bytes]],
    new: dict[str, list[bytes]],
) -> None:
    """Handle modification of object."""
    pass # replace this

def handler_remove(dn: str, old: dict[str, list[bytes]]) -> None:
    """Handle removal of object."""

```

(continues on next page)

<sup>178</sup> <https://github.com/univention/univention-corporate-server/blob/5.2-5/doc/developer-reference/listener/simple.py>

(continued from previous page)

```
pass # replace this
```

### 6.4.3 Rename and move

In case rename and move actions should be handled separately, the following code may be used:

Source code: UCS source: [doc/developer-reference/listener/modrdn.py](https://github.com/univention/univention-corporate-server/blob/5.2-5/doc/developer-reference/listener/modrdn.py)<sup>179</sup>

```
modrdn = "1"

_delay = None

def handler(
    dn: str,
    new: dict[str, list[bytes]],
    old: dict[str, list[bytes]],
    command: str = "",
) -> None:
    global _delay
    if _delay:
        old_dn, old = _delay
        _delay = None
        if "a" == command and old['entryUUID'] == new['entryUUID']:
            handler_move(old_dn, old, dn, new)
            return
        handler_remove(old_dn, old)

    if "n" == command and "cn=Subschema" == dn:
        handler_schema(old, new)
    elif new and not old:
        handler_add(dn, new)
    elif new and old:
        handler_modify(dn, old, new)
    elif not new and old:
        if "r" == command:
            _delay = (dn, old)
        else:
            handler_remove(dn, old)
    else:
        pass # ignore, reserved for future use

def handler_add(dn: str, new: dict[str, list[bytes]]) -> None:
    """Handle creation of object."""
    pass # replace this

def handler_modify(
    dn: str,
    old: dict[str, list[bytes]],
    new: dict[str, list[bytes]],
) -> None:
    """Handle modification of object."""
    pass # replace this
```

(continues on next page)

<sup>179</sup> <https://github.com/univention/univention-corporate-server/blob/5.2-5/doc/developer-reference/listener/modrdn.py>

(continued from previous page)

```

def handler_remove(dn: str, old: dict[str, list[bytes]]) -> None:
    """Handle removal of object."""
    pass # replace this

def handler_move(
    old_dn: str,
    old: dict[str, list[bytes]],
    new_dn: str,
    new: dict[str, list[bytes]],
) -> None:
    """Handle rename or move of object."""
    pass # replace this

def handler_schema(
    old: dict[str, list[bytes]],
    new: dict[str, list[bytes]],
) -> None:
    """Handle change in LDAP schema."""
    pass # replace this

```

**Warning**

Please be aware that tracking the two subsequent calls for `modrdn` in memory might cause duplicates, in case the Univention Directory Listener is terminated while such an operation is performed. If this is critical, the state should be stored persistently into a temporary file.

## 6.4.4 Full example with packaging

The following example shows a listener module, which logs all changes to users into the file `/root/UserList.txt`.

Source code: UCS source: `doc/developer-reference/listener/printusers`<sup>180</sup>

```

"""
Example for a listener module, which logs changes to users.
"""

import errno
import os
from collections import namedtuple

import univention.debug as ud
from listener import SetUID

name = 'printusers'
description = 'print all names/users/uidNumbers into a file'
filter = ''.join("""\
(&
  (|
    (&
      (objectClass=posixAccount)

```

(continues on next page)

<sup>180</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/doc/developer-reference/listener/printusers/>

(continued from previous page)

```

        (objectClass=shadowAccount)
    )
    (objectClass=univentionMail)
    (objectClass=sambaSamAccount)
    (objectClass=simpleSecurityObject)
    (objectClass=inetOrgPerson)
)
(! (objectClass=univentionHost))
(! (uidNumber=0))
(! (uid=*$))
)"".split())
attributes = ['uid', 'uidNumber', 'cn']
_Rec = namedtuple('_Rec', 'uid uidNumber cn')

USER_LIST = '/root/UserList.txt'

def handler(dn: str, new: dict[str, list[bytes]], old: dict[str, list[bytes]]) ->
↳None:
    """
    Write all changes into a text file.
    This function is called on each change.
    """
    if new and old:
        _handle_change(dn, new, old)
    elif new and not old:
        _handle_add(dn, new)
    elif old and not new:
        _handle_remove(dn, old)

def _handle_change(dn: str, new: dict[str, list[bytes]], old: dict[str,
↳list[bytes]]) -> None:
    """
    Called when an object is modified.
    """
    o_rec = _rec(old)
    n_rec = _rec(new)
    ud.debug(ud.LISTENER, ud.INFO, 'Edited user "%s"' % (o_rec.uid,))
    _writeit(o_rec, u'edited. Is now:')
    _writeit(n_rec, u'')

def _handle_add(dn: str, new: dict[str, list[bytes]]) -> None:
    """
    Called when an object is newly created.
    """
    n_rec = _rec(new)
    ud.debug(ud.LISTENER, ud.INFO, 'Added user "%s"' % (n_rec.uid,))
    _writeit(n_rec, u'added')

def _handle_remove(dn: str, old: dict[str, list[bytes]]) -> None:
    """
    Called when an previously existing object is removed.
    """

```

(continues on next page)

(continued from previous page)

```

o_rec = _rec(old)
ud.debug(ud.LISTENER, ud.INFO, 'Removed user "%s"' % (o_rec.uid,))
_writeit(o_rec, u'removed')

def _rec(data: dict[str, list[str]]) -> _Rec:
    """
    Retrieve symbolic, numeric ID and name from user data.
    """
    return _Rec(*(data.get(attr, (None,))[0] for attr in attributes))

def _writeit(rec: _Rec, comment: str) -> None:
    """
    Append CommonName, symbolic and numeric User-Identifier, and comment to file.
    """
    nuid = u'*****' if rec.uid in ('root', 'spam') else rec.uidNumber
    indent = '\t' if comment is None else ''
    try:
        with SetUID():
            with open(USER_LIST, 'a') as out:
                print(u'%sName: "%s"' % (indent, rec.cn), file=out)
                print(u'%sUser: "%s"' % (indent, rec.uid), file=out)
                print(u'%sUID: "%s"' % (indent, nuid), file=out)
                if comment:
                    print(u'%s%s' % (indent, comment,), file=out)
    except IOError as ex:
        ud.debug(
            ud.LISTENER, ud.ERROR,
            'Failed to write "%s": %s' % (USER_LIST, ex))

def initialize() -> None:
    """
    Remove the log file.
    This function is called when the module is forcefully reset.
    """
    try:
        with SetUID():
            os.remove(USER_LIST)
        ud.debug(
            ud.LISTENER, ud.INFO,
            'Successfully deleted "%s"' % (USER_LIST,))
    except OSError as ex:
        if errno.ENOENT == ex.errno:
            ud.debug(
                ud.LISTENER, ud.INFO,
                'File "%s" does not exist, will be created' % (USER_LIST,))
        else:
            ud.debug(
                ud.LISTENER, ud.WARN,
                'Failed to delete file "%s": %s' % (USER_LIST, ex))

```

Some comments on the code:

- The LDAP filter is specifically chosen to only match user objects, but not computer objects, which have a `uid` characteristically terminated by a `$`-sign.

- The `attribute` filter further restricts the module to only trigger on changes to the numeric and symbolic user identifier and the last name of the user.
- To test this run a command like `tail -f /root/UserList.txt &`. Then create a new user or modify the `lastname` of an existing one to trigger the module.

For packaging the following files are required:

#### **debian/printusers.install**

The module should be installed into the directory `/usr/lib/univention-directory-listener/system/`.

```
printusers.py usr/lib/univention-directory-listener/system/
```

#### **debian/printusers.postinst**

The Univention Directory Listener must be restarted after package installation and removal:

```
#!/bin/sh
set -e

case "$1" in
configure)
    deb-systemd-invoke restart univention-directory-listener
    ;;
abort-upgrade|abort-remove|abort-deconfigure)
    ;;
*)
    echo "postinst called with unknown argument \"$1\"" >&2
    exit 1
    ;;
esac

#DEBHELPER#

exit 0
```

#### **debian/printusers.postrm**

```
#!/bin/sh
set -e

case "$1" in
remove)
    deb-systemd-invoke restart univention-directory-listener
    ;;
purge|upgrade|failed-upgrade|abort-install|abort-upgrade|disappear)
    ;;
*)
    echo "postrm called with unknown argument \"$1\"" >&2
    exit 1
    ;;
esac

#DEBHELPER#

exit 0
```

## 6.4.5 A little bit more object oriented

For larger modules it might be preferable to use a more object oriented design like the following example, which logs referential integrity violations into a file.

Source code: UCS source: [doc/developer-reference/listener/obj.py](https://github.com/univention/univention-corporate-server/blob/5.2-5/doc/developer-reference/listener/obj.py)<sup>181</sup>

```
import os
from pwd import getpwnam

import ldap
import univention.debug as ud
from listener import SetUID

name = "refcheck"
description = "Check referential integrity of uniqueMember relations"
filter = "(uniqueMember=*)"
attribute = ["uniqueMember"]
modrdn = "1"

class LocalLdap(object):
    PORT = 7636

    def __init__(self) -> None:
        self.data: dict[str, str] = {}
        self.con: ldap.ldapobject.LDAPObject | None = None

    def setdata(self, key: str, value: str):
        self.data[key] = value

    def prerun(self) -> None:
        try:
            self.con = ldap.initialize('ldaps://%s:%d' % (self.data["ldapservers"],
↳self.PORT))
            self.con.simple_bind_s(self.data["binddn"], self.data["bindpw"])
        except ldap.LDAPError as ex:
            ud.debug(ud.LISTENER, ud.ERROR, str(ex))

    def postrun(self) -> None:
        if not self.con:
            return
        try:
            self.con.unbind()
            self.con = None
        except ldap.LDAPError as ex:
            ud.debug(ud.LISTENER, ud.ERROR, str(ex))

class LocalFile(object):
    USER = "listener"
    LOG = "/var/log/univention/refcheck.log"

    def initialize(self) -> None:
        try:
            ent = getpwnam(self.USER)
            with SetUID():
```

(continues on next page)

<sup>181</sup> <https://github.com/univention/univention-corporate-server/blob/5.2-5/doc/developer-reference/listener/obj.py>

(continued from previous page)

```

        with open(self.LOG, "w"):
            pass
        os.chown(self.LOG, ent.pw_uid, -1)
    except OSError as ex:
        ud.debug(ud.LISTENER, ud.ERROR, str(ex))

def log(self, msg) -> None:
    with open(self.LOG, 'a') as log:
        print(msg, file=log)

def clean(self) -> None:
    try:
        with SetUID():
            os.remove(self.LOG)
    except OSError as ex:
        ud.debug(ud.LISTENER, ud.ERROR, str(ex))

class ReferentialIntegrityCheck(LocalLdap, LocalFile):
    MESSAGES = {
        (False, False): "Still invalid: ",
        (False, True): "Now valid: ",
        (True, False): "Now invalid: ",
        (True, True): "Still valid: ",
    }

    def __init__(self) -> None:
        super(ReferentialIntegrityCheck, self).__init__()
        self._delay: tuple[str, dict[str, list[bytes]]] | None = None

    def handler(
        self,
        dn: str,
        new: dict[str, list[bytes]],
        old: dict[str, list[bytes]],
        command: str = '',
    ) -> None:
        if self._delay:
            old_dn, old = self._delay
            self._delay = None
            if "a" == command and old['entryUUID'] == new['entryUUID']:
                self.handler_move(old_dn, old, dn, new)
                return
            self.handler_remove(old_dn, old)

        if "n" == command and "cn=Subschema" == dn:
            self.handler_schema(old, new)
        elif new and not old:
            self.handler_add(dn, new)
        elif new and old:
            self.handler_modify(dn, old, new)
        elif not new and old:
            if "r" == command:
                self._delay = (dn, old)
            else:
                self.handler_remove(dn, old)

```

(continues on next page)

(continued from previous page)

```

    else:
        pass # ignore, reserved for future use

def handler_add(self, dn: str, new: dict[str, list[bytes]]) -> None:
    if not self._validate(new):
        self.log("New invalid object: " + dn)

def handler_modify(
    self,
    dn: str,
    old: dict[str, list[bytes]],
    new: dict[str, list[bytes]],
) -> None:
    valid = (self._validate(old), self._validate(new))
    msg = self.MESSAGES[valid]
    self.log(msg + dn)

def handler_remove(self, dn: str, old: dict[str, list[bytes]]) -> None:
    if not self._validate(old):
        self.log("Removed invalid: " + dn)

def handler_move(
    self,
    old_dn: str,
    old: dict[str, list[bytes]],
    new_dn: str,
    new: dict[str, list[bytes]],
) -> None:
    valid = (self._validate(old), self._validate(new))
    msg = self.MESSAGES[valid]
    self.log("%s %s -> %s" % (msg, old_dn, new_dn))

def handler_schema(
    self,
    old: dict[str, list[bytes]],
    new: dict[str, list[bytes]],
) -> None:
    self.log("Schema change")

def _validate(self, data: dict[str, list[bytes]]) -> bool:
    assert self.con
    try:
        for dn in data["uniqueMember"]:
            self.con.search_ext_s(dn, ldap.SCOPE_BASE, attrlist=[], ↵
↵attrsonly=1)
            return True
    except ldap.NO_SUCH_OBJECT:
        return False
    except ldap.LDAPError as ex:
        ud.debug(ud.LISTENER, ud.ERROR, str(ex))
        return False

_instance = ReferentialIntegrityCheck()
initialize = _instance.initialize
handler = _instance.handler

```

(continues on next page)

(continued from previous page)

```
clean = _instance.clean
prerun = _instance.prerun
postrun = _instance.postrun
setdata = _instance.setdata
```

## 6.5 Technical Details

### 6.5.1 User-ID and Credentials

The listener runs with the effective permissions of the user `listener`. If root-privileges are required, `listener.SetUID()` can be used as a context manager or method wrapper to switch the effective `UID`.

```
from listener import SetUID

@SetUID()
def prerun() -> None:
    pass

def postrun() -> None:
    with SetUID(0):
        pass
```

### 6.5.2 Internal Cache

The directory `/var/lib/univention-directory-listener/` contains several files:

#### **cache/cache.mdb, cache/lock.mdb**

Starting with UCS 4.2, the LMDB cache database contains a copy of all objects and their attributes. It is used to supply the old values supplied through the `old` parameter, when the function `handler()` is called.

The cache is also used to keep track, for which object which module was called. This is required when a new module is added, which is invoked for all already existing objects when the Univention Directory Listener is restarted.

On domain controllers the cache could be replaced by doing a query to the local LDAP server, before the new values are written into it. But Managed Node doesn't have a local LDAP server, so there the cache is needed. Also note that the cache keeps track of the associated listener modules, which is not available from the LDAP.

It also contains the [KB 13149 - CacheMasterEntry<sup>182</sup>](https://help.univention.com/t/13149), which stores the notifier and schema ID.

#### **cache.lock**

Starting with UCS 4.2, this file is used to detect if a listener opened the cache database.

#### **cache.db, cache.db.lock**

Before UCS 4.2, the BDB cache file contained a copy of all objects and their attributes. With the update to UCS 4.2, it gets converted into an LMDB database.

#### **notifier\_id**

This legacy file contains the last notifier ID read from the Univention Directory Notifier.

#### **handlers/**

For each module the directory contains a text file consisting of a single number. The name of the file is derived from the values of the variable `name` as defined in each listener module. The number is to be interpreted as a bit-field of `HANDLER_INITIALIZED=0x1` and `HANDLER_READY=0x2`. If both bits are set, it indicates that the module was successfully initialized by running the function `initialize()` (page 45). Otherwise both bits are unset.

The package `univention-directory-listener` contains several commands useful for controlling and debugging problems with the Univention Directory Listener. This can be useful for debugging listener cache inconsistencies.

<sup>182</sup> <https://help.univention.com/t/13149>

### `univention-directory-listener-ctrl`

The command `univention-directory-listener-ctrl status` shows the status of the Listener. This includes the transaction from the Primary Directory Node in comparison to the last processes transaction. It also shows a list of all installed modules and their status.

The command `univention-directory-listener-ctrl resync $name` can be used to reset and re-initialize a module. It stops any currently running listener process, removes the state file for the specified module and starts the listener process again. This forces the functions `clean()` (page 46) and `initialize()` (page 45) to be called one after the other.

### `univention-directory-listener-dump`

The command `univention-directory-listener-dump` can be used to dump the cache file `/var/lib/univention-directory-listener/cache.db`. The Univention Directory Listener must be stopped first by invoking `systemctl stop univention-directory-listener`. It outputs the cache in format compatible to the LDAP Data Interchange Format (LDIF).

### `univention-directory-listener-verify`

The command `univention-directory-listener-verify` can be used to compare the content of the cache file `/var/lib/univention-directory-listener/cache.db` to the content of an LDAP server. The Univention Directory Listener must be stopped first by invoking `systemctl stop univention-directory-listener`. LDAP credentials must be supplied at the command line. For example, the following command would use the machine password:

```
$ univention-directory-listener-verify \  
-b "$(ucr get ldap/base)" \  
-D "$(ucr get ldap/hostdn)" \  
-y /etc/machine.secret
```

### `get_notifier_id.py`

The command `/usr/share/univention-directory-listener/get_notifier_id.py` can be used to get the latest ID from the notifier. This is done by querying the Univention Directory Notifier running on the LDAP server configured through the Univention Configuration Registry Variable `ldap/master`<sup>183</sup>. The returned value should be equal to the value currently stored in the file `/var/lib/univention-directory-listener/notifier_id`. Otherwise, the Univention Directory Listener might still be processing a transaction or it might indicate a problem with the Univention Directory Listener

## 6.5.3 Internal working

The Listener/Notifier mechanism is used to trigger arbitrary actions when changes occur in the LDAP directory service. In addition to the LDAP server `slapd` it consists of two other services: The Univention Directory Notifier service runs next to the LDAP server and broadcasts change information to interested parties. The Univention Directory Listener service listens for those notifications, downloads the changes and runs listener modules performing arbitrary local actions like storing the data in a local LDAP server for replication or generating configuration files for non-LDAP-aware local services.

On startup the listener connects to the notifier and opens a persistent TCP connection to port 6669. The host can be configured through several Univention Configuration Registry Variables:

- If `notifier/server` is explicitly set, only that named host is used. In addition, the Univention Configuration Registry Variable `notifier/server/port` can be used to explicitly configure a different TCP port other than 6669.
- Otherwise, on the Primary Directory Node and on all Backup Directory Nodes, only the host named in `ldap/master`<sup>184</sup> is used.

---

<sup>183</sup> <https://docs.software-univention.de/manual/5.2/en/appendix/variables.html#envvar-ldap-master>

<sup>184</sup> <https://docs.software-univention.de/manual/5.2/en/appendix/variables.html#envvar-ldap-master>

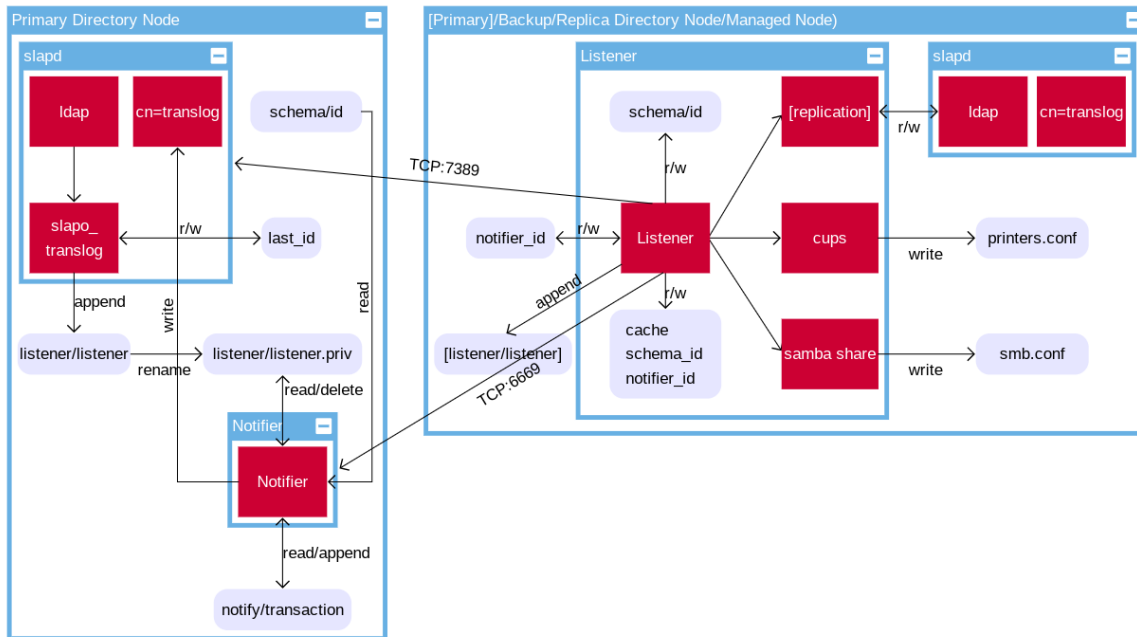


Fig. 6.1: Listener/Notifier mechanism

- Otherwise, on all other system roles a host is chosen randomly from the combined list of names in `ldap/master`<sup>185</sup> and `ldap/backup`.

This list of Backup Directory Nodes stored in the Univention Configuration Registry Variable `ldap/backup` is automatically updated by the listener module `ldap_server.py`.

The following steps occur on changes:

1. An LDAP object is modified on the Primary Directory Node. Changes initiated on all other system roles are re-directed to the Primary Directory Node.
2. The UCS-specific overlay-module `translog` assigns the next transaction number. It uses the file `/var/lib/univention-ldap/last_id` to keep track of the last transaction number.

As a fallback the transaction number of the last entry from the file `/var/lib/univention-ldap/listener/listener` or `/var/lib/univention-ldap/notify/transaction` is used. The module appends the transaction ID, DN and change type to the file `/var/lib/univention-ldap/listener/listener`.

Referred to as `FILE_NAME_LISTENER`, `TRANSACTION_FILE` in the source code.

3. The Univention Directory Notifier watches that file and waits until it becomes non empty. The file is then renamed to `/var/lib/univention-ldap/listener/listener.priv` (referred to as `FILE_NAME_NOTIFIER_PRIV`) and the original file is re-created empty. The transactions from the renamed file are processed line-by-line and are appended to the file `/var/lib/univention-ldap/notify/transaction` (referred to as `FILE_NAME_TF` in the source code), including the DN. Since protocol version 3 the notifier also stores the same information within the LDAP server by creating the entry `reqSession=ID,cn=translog`. After successful processing the renamed file is deleted. For efficient access by transaction ID the index `transaction.index` is updated.
4. All listeners get notified of the new transaction. Before UCS 4.3 erratum 427<sup>186</sup> the information already included the latest transaction ID, DN and the change type. With protocol version 3 only the transaction ID is included.

<sup>185</sup> <https://docs.software-univention.de/manual/5.2/en/appendix/variables.html#envvar-ldap-master>

<sup>186</sup> <https://errata.software-univention.de/#/?erratum=4.3x427>

5. Each listener opens a connection to the LDAP server running on the UCS system which was used to query the Notifier. With protocol version 3 the listener first queries the LDAP server for the missing DN and change type information by retrieving the entry `reqSession=ID, cn=translog`. With that it retrieves the latest state of the object identified through the DN. If access is blocked, for example, by selective replication, the change is handled as a delete operation instead.
6. The old state of the object is fetched from the local *Internal Cache* (page 65) located in `/var/lib/univention-directory-listener/cache/`.
7. For each module it is checked, if either the old or new state of the object matches the `filter` and `attributes` specified in the corresponding Python variables. If not, the module is skipped. By default `replication.py` is always called first to guarantee that the data is available from the local LDAP server for all subsequent modules. Since [UCS 5.0 erratum 164](#)<sup>187</sup> the order of how modules are called can be configured using the per module property `priority` (page 45).
8. If the function `prerun()` (page 46) of module was not called yet, this is done to signal the start of changes.
9. The function `handler()` (page 53) specified in the module is called, passing in the DN and the old and new state.
10. The main listener process updates its cache with the new values, including the names of the modules which successfully handled that object. This guarantees that the module is still called, even when the filter criteria would no longer match the object after modification.
11. On a Backup Directory Node the Univention Directory Listener writes the transaction data to the file `/var/lib/univention-ldap/listener/listener` (referred to as `FILE_NAME_LISTENER`, `TRANSACTION_FILE` in the source code) to allow the Univention Directory Notifier to be cascaded. This is configured internally with the option `-o` of `univention-directory-listener` and is done for load balancing and failover reasons.
12. The transaction ID is written into the legacy local file `/var/lib/univention-directory-listener/notifier_id`. It also is written into the *master record* of the listener cache.

After 15 seconds of inactivity the function `postrun()` (page 46) is invoked for all prepared modules. This signals a break in the stream of changes and requests the module to release its resources and/or start pending operations.

### 6.5.4 LDAP Schema handling

The LDAP Schema is managed on the Primary Directory Node. Extensions must be made available there first. All other systems running LDAP replica download it from there using the Univention Directory Notifier / Univention Directory Listener mechanism.

1. On the Primary Directory Node the LDAP Schema is extracted by the script `/etc/init.d/slaped` on each start. The MD5 hash is stored in `/var/lib/univention-ldap/schema/md5`.
2. On each change the counter in file `/var/lib/univention-ldap/schema/id/id` is incremented.
3. Univention Directory Notifier monitors that file and makes the value available over the network. It can be queried by running `/usr/share/univention-directory-listener/get_notifier_id.py -s`.
4. Univention Directory Listener retrieves the value during each transaction. It is stored in the local file `/var/lib/univention-ldap/schema/id/id` and in the `CacheMasterEntry` of the *Internal Cache* (page 65).
5. On change the Listener downloads the current Schema from the LDAP server of the Primary Directory Node, saves it to the local schema file `/var/lib/univention-ldap/schema.conf` and restarts the local service `slaped`.
6. The Listener then continues processing transactions.

---

<sup>187</sup> <https://errata.software-univention.de/#!/?erratum=5.0x164>

## 6.5.5 Python 3 migration

Since UCS 5.0 the Univention Directory Listener uses Python 3 to execute listener modules.

For a successful migration all functions must be migrated to work with Python 3. There is no change in the module variables (`name`, `description`, `filter`, ...) necessary.

The data structure of the arguments `new` and `old` given to the `handler()` (page 53) function now explicitly differentiates between byte strings (`bytes`<sup>188</sup>) and unicode strings (`str`<sup>189</sup>). The dictionary keys are strings while the LDAP attribute values are list of byte strings:

```
{
  'associatedDomain': [b'example.net'],
  'krb5RealmName': [b'EXAMPLE.NET'],
  'dc': [b'example'],
  'nisDomain': [b'example.net'],
  'objectClass': [
    b'top',
    b'krb5Realm',
    b'univentionPolicyReference',
    b'nisDomainObject',
    b'domainRelatedObject',
    b'domain',
    b'univentionBase',
    b'univentionObject'
  ],
  'univentionObjectType': [b'container/dc'],
}
```

While in UCS 4 `handler()` (page 53) typically looked like:

```
def handler(
    dn: str,
    new: dict[str, list[str]],
    old: dict[str, list[str]],
) -> None:
    if new and 'myObjectClass' in new.get('objectClass', []):
        value = new['myAttribute'][0]
        ...
```

In UCS 5.2 it would look like:

```
def handler(
    dn: str,
    new: dict[str, list[bytes]],
    old: dict[str, list[bytes]],
) -> None:
    if new and b'myObjectClass' in new.get('objectClass', []):
        value = new['myAttribute'][0].decode('UTF-8')
        ...
```

<sup>188</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

<sup>189</sup> <https://docs.python.org/3/library/stdtypes.html#str>



## UNIVENTION DIRECTORY MANAGER (UDM)

The Univention Directory Manager (UDM) is a wrapper for LDAP objects. Traditionally, LDAP stores objects as a collection of attributes, which are defined by so-called schemata. Modifying entries is slightly complicated, as there are no high-level operations to add or remove values from multi-valued attributes, or to keep the password used by different authentication schemes such as Windows NTLM-hashes, Unix MD5 hashes, or Kerberos tickets in sync.

The command line client `udm` provides different modes of operation.

```
udm [--binddn bind-dn --bindpwfile bind-password-file] [module] [mode] [options]
```

### Creating object

```
udm module create --set property=value ...
```

```
$ eval "$(ucr shell)"
$ udm container/ou create --position "$ldap_base" --set name="xxx"
```

Multiple `--sets` may be used to set the values of a multi-valued property.

The equivalent LDAP command would look like this:

```
$ eval "$(ucr shell)"
$ ldapadd -D "cn=admin,$ldap_base" -y /etc/ldap.secret <<__EOT__
dn: uid=xxx,$ldap_base
objectClass: organizationalRole
cn: xxx
__EOT__
```

### List object

```
udm module list [--dn dn | --filter property=value]
```

```
$ udm container/ou list --filter name="xxx"
```

```
$ univention-ldapsearch cn=xxx
```

### Modify object

```
udm module modify [--dn dn | --filter property=value] [--set property=value |
--append property=value | --remove property=value ...]
```

```
$ udm container/ou modify --dn "cn=xxx,$ldap_base" --set name="xxx"
```

For multi-valued attributes `--append` and `--remove` can be used to add additional values or remove existing values. `--set` overwrites any previous value, but can also be used multiple times to specify further values. `--set` and `--append` should not be mixed for any property in one invocation.

### Delete object

```
udm module remove [--dn dn | --filter property=value]
```

```
$ udm container/ou delete --dn "cn=xxx,$ldap_base"
```

If `--filter` is used, it must match exactly one object. Otherwise `udm` refuses to delete any object.

This chapter has the following content:

## 7.1 UDM modules

Univention Directory Manager uses a flexible and extensible structure of Python modules to manage the directory service data. Additional modules are automatically recognized after being saved to the file system and made available for use at the command line and web interface. The development of custom modules enables the flexible extension of the Univention Directory Manager beyond the scope of extended attributes.

### 7.1.1 Overview

Univention Directory Manager (UDM for short) uses its own module structure to map LDAP objects. Usually one of these UDM modules corresponds to an LDAP object (for example a user, a group or a container).

The modules are stored in the `/usr/lib/python3/dist-packages/univention/admin/handlers/` directory and organized by task. The modules for managing the various computer objects are located below the `computers/` folder, for example. It can be addressed by the command line interface through `computers/windows`.

Custom modules should, if possible, be placed in their own subdirectory to avoid conflicts with any standard modules that may later be integrated into UCS. For the modules to be initialized, a `__init__.py` file must exist in the directory.

### 7.1.2 Structure of a module

Modules contain the definition of the UDM properties and the definition of a class named `object`, which is derived from `univention.admin.handlers.simpleLdap`.

#### Note

The default name of the base class `object` has historical reasons. It must be kept despite the name collision with the Python type `object`<sup>190</sup>.

This section will begin with a detailed description of the variables to be defined. The *The Python class object* (page 76) takes a closer look at the `object` class and lists necessary definitions and functions within the class.

#### Global variables

The global variables with specific meanings in a Univention Directory Manager module are described below. Mandatory and optional variables are separated into mandatory variables and optional arguments.

#### Mandatory variables

`udm_modules_globals.module`

A string matching the name of the UDM module, for example `computers/computer`.

`udm_modules_globals.operations`

A list of strings which contains all LDAP operations allowed with this object. Available operations are `add`, `edit`, `remove`, `search`, `subtree_move`, and `copy`.

`udm_modules_globals.short_description`

This description is displayed as the name in the Univention Management Console. Within the UMC module LDAP navigation it is displayed in the selection list for possible object types.

`udm_modules_globals.long_description`

A detailed description of the module.

---

<sup>190</sup> <https://docs.python.org/3/library/functions.html#object>

`udm_modules_globals.childs`

Indicates whether this LDAP object is a container. If so, this variable is set to the value `True`, and otherwise to `False`.

`udm_modules_globals.options`

Variable `options` is a Python dictionary and defines various options that can either be set manually or left at default. These options can be changed later.

For example through the web interface of the UDM using the *Options* tab. If an option is activated, one or more LDAP object classes (given by parameter `objectClass`) are added to the object and further fields and/or tabs are activated in the Univention Management Console tabs (for example the groupware option for users). The dictionary assigns a unique string to each option (as *property\_descriptions* (page 73)).

Each instance has the following parameters:

`options.short_description`

A short description of the option, used for example in the Univention Management Console as descriptive text about the input fields.

`options.long_description`

A longer description of the option.

`options.default`

defines whether the option is enabled by default: `True` means active and `False` inactive.

`options.editable`

Defines whether this option can be set and removed multiple times, or always remains set after having been activated once.

`options.objectClasses`

A list of LDAP object classes, which the LDAP entry must consist of so that the option is enabled for the object.

Example:

```
options = {
    'opt1': univention.admin.option(
        short_description=_('short description'),
        default=True,
        objectClasses=['class1'],
    ),
}
```

`udm_modules_globals.property_descriptions`

This Python dictionary contains all UDM properties provided by the module. They are referenced using a unique string as a key (in this case as `univention.admin.property` objects). Usually, this kind of UDM property corresponds to an LDAP attribute, but can also be obtained or calculated from other sources.

Example:

```
property_descriptions = {
    'prop1': univention.admin.property(
        short_description=_('name'),
        long_description=_('long description'),
        syntax=univention.admin.syntax.string,
        multivalue=False,
        required=True,
        may_change=True,
        identifies=False,
        dontsearch=True,
        default=('default value'),
```

(continues on next page)

```

        options=['opt1'],
    ),
}

```

A short explanation of the parameters seen above:

`property_descriptions.short_description: str`<sup>191</sup>

A short description used for instance in the Univention Management Console as descriptive text to the input fields.

`property_descriptions.long_description: str`<sup>192</sup>

A detailed description used in the Univention Management Console for the tooltips.

`property_descriptions.syntax: type`<sup>193</sup>

This parameter specifies the property type. Based on these type definitions, the Univention Directory Manager can check the specified values for the property and provide a detailed error message in case of invalid values. A list of syntax classes is available in *UDM LDAP search* (page 84).

`property_descriptions.multivalue: bool`<sup>194</sup>

Accepts the values `True` or `False`. If set to `True` the properties value is a list. In this case, the `syntax` parameter specifies the type of elements within this list.

`property_descriptions.required: bool`<sup>195</sup>

If this parameter is set to `True`, a value must be specified for this property.

`property_descriptions.may_change: bool`<sup>196</sup>

If set to `True`, the properties value can be modified at a later point, if not, it can only be specified once when the object is created.

`property_descriptions.editable: bool`<sup>197</sup>

If set to `False`, the properties value can't even be specified when the object is created. This is usually only interesting or useful for automatically generated or calculated values.

`property_descriptions.identifies: bool`<sup>198</sup>

This option should be set to `True` if the property uniquely identifies the object (through the LDAP DN). In most cases it should be set for exactly one property of a module.

`property_descriptions.dontsearch: bool`<sup>199</sup>

If set to `False`, the property is not searchable.

`property_descriptions.default: Any`

The default value of a property, when the object is created through the Univention Management Console.

`property_descriptions.options: list`<sup>200</sup> [`str`<sup>201</sup>]

A list of keywords identifying options with which this property can be shown or hidden.

`udm_modules_globals.layout`

The UDM properties of an object can be arranged in groups. They are represented as tabs in the Univention Directory Manager for example. For each tab, an instance of `univention.admin.layout#Tab` must

<sup>191</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>192</sup> <https://docs.python.org/3/library/stdtypes.html#str>

<sup>193</sup> <https://docs.python.org/3/library/functions.html#type>

<sup>194</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>195</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>196</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>197</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>198</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>199</sup> <https://docs.python.org/3/library/functions.html#bool>

<sup>200</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>201</sup> <https://docs.python.org/3/library/stdtypes.html#str>

be created in the array `layout`. The name, a description for the tab and a list of rows are expected as parameters. A line can contain up to two properties, for each of which an instance of `univention.admin.layout#Group` must be created. The UDM property name from `property_descriptions` (page 73) is expected as a parameter for each instance.

```
from univention.admin.layout import Tab, Group
layout = [
    Tab(_('Tab header'), _('Tab description'), layout=[
        Group('Group', 'group description', [
            ['prop1', 'prop2']
            ['prop3', ]
        ]),
        ...
    ], advanced=True),
    ...
]
```

The optional `advanced=True` setting controls whether the tab should be displayed on the *Advanced settings* by default.

#### `udm_modules_globals.mapping`

Maps the UDM properties to LDAP attributes. Usually, a mapping is registered for each property, linking the name of a UDM property (`udm_name`) to the associated LDAP attribute (`ldap_name`):

```
mapping.register(udm_name, ldap_name)
mapping.register(udm_name, ldap_name, map_value, unmap_value)
```

Two functions are available to convert the values between UDM properties and LDAP attribute. To convert from UDM → LDAP, `map_value()` is used, while `unmap_value()` is used to convert in the opposite direction (LDAP → UDM). The second function is necessary for all single-valued UDM properties, since these are always implemented as null or one-element lists within LDAP. The default implementation `univention.admin.mapping.ListToString()` always returns the first entry of the list and can therefore generally be specified as a `unmap_value()` function for all single-valued attributes. For `map_value()` (UDM → LDAP), it is sufficient to specify `None`, which ensures that any existing value, if present, is converted to a single-element list.

#### Warning

UDM properties always contain either a string (single-valued attributes) or a list of strings (multi-valued attributes), never just a number or any other Python type!

## Optional arguments

The following specifications are optional and only need to be defined if a module has these special properties:

#### `udm_modules_globals.virtual`

Modules that set this variable to `True` are a kind of helper module for other modules that have no associated LDAP objects. An example of this is the `computers/computer` module, which is an auxiliary module for all types of computers.

#### `udm_modules_globals.template`

A module that sets this variable to another UDM module (e.g. `settings/usertemplate`), gains the ability to define default values for UDM properties from other modules. An example of this is the user template (more specifically the `settings/usertemplate` module). Such a template can for example be selected when creating a user so that the values defined in it are taken over as defaults in the input masks.

## The Python class `object`

The Python class `object` of a module provides the interface between Univention Directory Manager and the LDAP operations triggered when an object is created, modified, moved or deleted. It supports the Univention Directory Manager in mapping the UDM module and its properties to LDAP objects and attributes.

This requires adhering to the predefined API of the class. The base class `univention.admin.handlers.simpleLdap` provides the essential functionality for simple LDAP objects, so usually only a few adjustments are necessary. An instance (`self`) encapsulates all information of an object, which can be accessed in various ways:

```
class udm_modules_globals.object
```

```
self.dn → String
```

Distinguished Name in the LDAP DIT

```
self.position → univention.admin.uldap#Position
```

Container element in the LDAP DIT

```
self['UDM-property-name'] → [values, ...]
```

Wrapper around `self.info` which also checks the value against the syntax when assigned and returns default values when read.

```
self.info['UDM-property-name'] → [values, ...]
```

Dictionary with the currently set values of the UDM properties. Direct access to it allows the initialization of `editable=False` properties and skips any syntax checks.

```
self.oldinfo['UDM-property-name'] → [values, ...]
```

Dictionary of the originally read values converted to UDM property names. It is primarily needed to internally propagate changes to the Python object back to the corresponding entry in the LDAP.

```
self.ldapattr['LDAP-Attributname'] → [values, ...]
```

Dictionary of the attributes originally read from LDAP.

```
self.oldpolicies → [Policy-DNs, ...]
```

Copy of the list of DN's of the referenced `univentionPolicyReference`

```
self.policies → [Policy-DNs, ...]
```

List of DN's of the referenced `univentionPolicyReference`

```
self.policyObjects[Policy-DN] → univention.admin.handlers#SimplePolicy
```

Dictionary of the loaded policies.

```
self.extended_ldap_attributes → [univention.admin#Extended_attribute, ...]
```

Complete list of the objects extended attributes

The `simpleLdap` class also provides the possibility of additional customization before and after the LDAP operation by calling functions. For example, before creating an LDAP object the function `_ldap_pre_create()` is called and after the operation the function `_ldap_post_create()` is called. Such pre- and post-functions similarly exist for the `modify()`, `move()` and `remove()` functions. The following table lists all used functions in calling order from top to bottom:

Table 7.1: LDAP actions and hooks

Description	Create	Modify	Remove
Before validation	<code>_ldap_pre_rea</code>		
Validates, that all required attributes are set	<code>ready()</code>		
	<code>_ldap_pre_cre</code>	<code>_ldap_pre_mod</code>	<code>_ldap_pre_re-</code>
	<code>ate()</code>	<code>ify()</code>	<code>move()</code>
Policy Copy-on-Write	<code>_up-</code>	<code>_up-</code>	
	<code>date_poli-</code>	<code>date_poli-</code>	
	<code>cies()</code>	<code>cies()</code>	
Extension point for Extended Attribute	<code>hook_ldap_pre</code>	<code>hook_ldap_pre</code>	<code>hook_ldap_pre_re-</code>
	<code>ate()</code>	<code>ify()</code>	<code>move()</code>
Returns initial list of (LDAP-attribute-name, value)- resp. (LDAP-attribute-name, [values]) tuples	<code>_ldap_ad-</code>		
	<code>dlist()</code>		
Calculates difference between <code>self.oldinfo</code> and <code>self.info</code>	<code>_ldap_modlist()</code>		
Extension point for Extended Attribute	<code>hook_ldap_ad-</code>	<code>hook_ldap_mod</code>	
	<code>dlist()</code>	<code>list()</code>	
Real action	<b>ADD</b>	<b>MODIFY</b>	<b>DELETE</b>
	<code>_ldap_post_cr</code>	<code>_ldap_post_mo</code>	<code>_ldap_post_re-</code>
	<code>ate()</code>	<code>ify()</code>	<code>move()</code>
Extension point for Extended Attribute	<code>hook_ldap_pos</code>	<code>hook_ldap_pos</code>	<code>hook_ldap_post_re-</code>
	<code>ate()</code>	<code>ify()</code>	<code>move()</code>

The functions `hook_ldap_*` are described in *Extended attribute hooks* (page 94).

### The `identify()` and `lookup()` functions

These functions are used to find the corresponding objects for search queries from the Univention Management Console (`lookup()`) and to assign LDAP objects to a Univention Directory Manager module. For simple LDAP objects, no modifications are necessary. They can be assigned to the `generic` objects class methods:

```
lookup = object.lookup
lookup_filter = object.lookup_filter
identify = object.identify
```

### 7.1.3 Example module

The following is an example module for the Univention Directory Manager which is also available as a package. (**univention-directory-manager-module-example**) The complete source code is available at [UCS source: packaging/univention-directory-manager-module-example](https://github.com/univention/univention-corporate-server/tree/5.2-5/packaging/univention-directory-manager-module-example/)<sup>202</sup>.

The directory contains a source package in Debian format, from which two binary packages are created during package build through `./debian/rules binary`: A schema package, which must be installed on the Primary Directory Node, and the package containing the UDM module itself. The sample code also includes a `ip-phone-tool` script that shows an example of using the UDM Python API in a Python script.

A Univention Directory Manager module almost always consists of two components:

- The Python module, which contains the implementation of the interface to the Univention Directory Manager.
- A LDAP schema, which defines the LDAP object to be managed. Both parts are described below, with the focus lying on the creation of the Python module.

The following module for the Univention Directory Manager demonstrates the rudimentary administration of IP telephones. It tries to show as many possibilities of a Univention Directory Manager module as possible within a simple example.

<sup>202</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/packaging/univention-directory-manager-module-example/>

## Python code of the example module

Before defining the actual module source code, some basic Python modules need to be imported, which are always necessary:

```
import re

import univention.admin.handlers
import univention.admin.syntax
import univention.admin.localization
from univention.admin.layout import Tab
```

This list of Python modules can of course be extended. As described in *Global variables* (page 72), some necessary global variables are defined at the beginning of a Univention Directory Manager module, which provide a description of the module:

```
module = 'test/ip_phone'
childs = False
short_description = _('IP-Phone')
long_description = _('An example module for the Univention Directory Manager')
operations = ['add', 'edit', 'remove', 'search', 'move', 'copy']
```

Another global variable important for the Univention Management Console, is *layout* (page 74).

```
layout = [
    Tab(_('General'), _('Basic Settings'), layout=[
        ["name", "active"],
        ["ip", "protocol"],
        ["priuser"],
    ]),
    Tab(_('Advanced'), _('Advanced Settings'), layout=[
        ["users"],
    ], advanced=True),
    Tab(_('Redirect'), _('Redirect Option'), layout=[
        ["redirect_user"],
    ], advanced=True),
]
```

It structures the layout of the objects individual properties on the tabs. The list consists of elements whose type is `univention.admin.layout.Tab`, each determining the content of a tab. In this case there are the `General`, `Advanced` and `Redirect` tabs. Next, the options (*options* (page 73)) and properties (*property\_descriptions* (page 73)) of the module should be defined. In this case, the `default` and `redirection` options are created, whose functions will be explained later. To configure the parameters, the `univention.admin.option` object is passed to the `short_description` option for a short description. `default` defines the pre-configuration. `True` activates the option while `False` deactivates it.

```
options = {
    'default': univention.admin.option(
        short_description=short_description,
        default=True,
        objectClasses=['top', 'testPhone'],
    ),
    'redirection': univention.admin.option(
        short_description=_('Call redirect option'),
        default=True,
        editable=True,
        objectClasses=['testPhoneCallRedirect'],
    ),
}
```

After the modules options, its properties are defined. UDM properties are defined through textual descriptions, syntax definitions and instructions for the Univention Management Console.

```
property_descriptions = {
    ...
}
```

The `name` property defines the `hostname` of the IP phone. The `syntax` parameter tells the Univention Directory Manager that valid values for this property must match the syntax of a computer name. Additional predefined syntax definitions can be found in the `property_descriptions` (page 73) section.

```
'name': univention.admin.property(
    short_description=_('Name'),
    long_description=_('ID of the IP-phone'),
    syntax=univention.admin.syntax.hostName,
    required=True,
    identifies=True,
),
```

The `active` is an example of a boolean/binary property which can only take the values `True` or `False`. In this example, it defines an activation/blocking of the IP phone. The parameter `default=True` initially unlocks the phone:

```
'active': univention.admin.property(
    short_description=_('active'),
    long_description=_('The IP-phone can be deactivated'),
    syntax=univention.admin.syntax.TrueFalseUp,
    default='TRUE',
),
```

The `protocol` property specifies which VoIP protocol is supported by the phone. No standard syntax definition is used for this property, but a specially declared `SynVoIP_Protocols` class. (The source code of this class follows in a later section). The syntax of the class defines a selection list with a predefined set of possibilities. The `default` parameter preselects the value with the `sip` key.

```
'protocol': univention.admin.property(
    short_description=_('Protocol'),
    long_description=_('Supported VoIP protocols'),
    syntax=SynVoIP_Protocols
    default='sip',
),
```

The `ip` property specifies the phones IP address. The predefined class `univention.admin.syntax.ipAddress` is specified as the syntax definition. Additionally, the `required` parameter enforces that setting this property is mandatory.

```
'ip': univention.admin.property(
    short_description=_('IP-Address'),
    long_description=_('IP-Address of the IP-phone'),
    syntax=univention.admin.syntax.ipAddress,
    required=True,
),
```

The `priuser` property sets the primary user of the IP phone. A separate syntax definition is again used, which in this case is a class that defines the valid values by means of a regular expression. (The source code is shown later)

```
'priuser': univention.admin.property(
    short_description=_('Primary User'),
    long_description=_('The primary user of this IP-phone'),
```

(continues on next page)

(continued from previous page)

```

syntax=SynVoIP_Address,
required=True,
),

```

The `users` property indicates that options are used. Since `multivalued` is set to `True` in this example, the `users` object is a list of addresses.

```

'users': univention.admin.property(
    short_description=_('Additional Users'),
    long_description=_('Users, that may register with this phone'),
    syntax=SynVoIP_Address,
    multivalued=True,
),

```

The `redirect_user` property is used to redirect incoming calls to a different phone number. It is only shown if the `options=['redirection']` is set.

```

'redirect_user': univention.admin.property(
    short_description=_('Redirection User'),
    long_description=_('Address for call redirection'),
    syntax=SynVoIP_Address,
    options=['redirection'],
),

```

The following two classes are the syntax definitions used for the `protocols`, `priuser` and `users` properties. `SynVoIP_Protocols` is based on the predefined `univention.admin.syntax.select` class, which provides the basic functionality for select lists. Derived classes, as seen in the following class, only need to define a name and the list of choices.

```

class SynVoIP_Protocols(univention.admin.syntax.select):
    name = _('VoIP_Protocol')
    choices = [('sip', _('SIP')), ('h323', _('H.323')), ('skype', _('Skype'))]

```

The other syntax definition (`SynVoIP_Address`) is based on the `univention.admin.syntax.simple` class, which provides basic functionality for syntax definitions utilizing regular expressions. As with the other definition, a name must be assigned. Additionally, the attributes `min_length` and `max_length` must be specified. If one of these attributes is set to 0, it corresponds to a nonexistent limit in the respective direction. In addition to the attributes mentioned, the `parse()` function must also be defined, which passes the value to be checked as a parameter. By means of the Python module `re` it is in this case checked whether the value corresponds to the pattern of a VoIP address, e.g. `sip:hans@mustermann.de`.

```

class SynVoIP_Address(univention.admin.syntax.simple):
    name = _('VoIP_Address')
    min_length = 4
    max_length = 256
    _re = re.compile('(^((sip|h323|skype):)?([a-zA-Z])[a-zA-Z0-9._-]+)@[a-zA-Z0-9._-]+$')

    def parse(self, text):
        if self._re.match(text) is not None:
            return text
        raise univention.admin.ueexceptions.valueError(_('Not a valid VoIP Address'))

```

Mapping the UDM module properties to the Attributes of the to be created LDAP object is the next step. (*mapping* (page 75)). To do this, the `univention.admin.mapping.mapping` class is used, which provides a simple way to register mappings for the individual LDAP attributes to UDM properties with the `register()` function. This function's first argument is the modules UDM property name and the second the LDAP attribute name. The following

two arguments of the `register()` function can be used to specify mapping functions for conversion from the modules UDM property to the LDAP attribute and vice versa.

```
mapping = univention.admin.mapping.mapping()
mapping.register('name', 'cn', None, univention.admin.mapping.ListToString)
mapping.register('active', 'testPhoneActive', None, univention.admin.mapping.
↳ListToString)
mapping.register('protocol', 'testPhoneProtocol', None, univention.admin.mapping.
↳ListToString)
mapping.register('ip', 'testPhoneIP', None, univention.admin.mapping.ListToString)
mapping.register('priuser', 'testPhonePrimaryUser', None, univention.admin.mapping.
↳ListToString)
mapping.register('users', 'testPhoneUsers')
mapping.register('redirect_user', 'testPhoneRedirectUser', None, univention.admin.
↳mapping.ListToString)
```

Finally, *The Python class object* (page 76) must be defined for the module that conforms to the specifications defined in *Structure of a module* (page 72). For the IP phone, the class would look like this:

```
class object(univention.admin.handlers.simpleLdap):

    module = module

    def open(self):
        super(object, self).open()
        self.save()

    def _ldap_pre_create(self):
        return super(object, self)._ldap_pre_create()

    def _ldap_post_create(self):
        return super(object, self)._ldap_post_create()

    def _ldap_pre_modify(self):
        return super(object, self)._ldap_pre_modify()

    def _ldap_post_modify(self):
        return super(object, self)._ldap_post_modify()

    def _ldap_pre_remove(self):
        return super(object, self)._ldap_pre_remove()

    def _ldap_post_remove(self):
        return super(object, self)._ldap_post_remove()

    def _ldap_modlist(self):
        ml = super(object, self)._ldap_modlist()
        return ml
```

To enable searching for objects managed by this module, two additional functions are available: `lookup()` and `identify()` (see *The identify() and lookup() functions* (page 77)). The functions provided here should be sufficient for simple LDAP objects that can be identified by a single `objectClass`.

```
lookup = object.lookup
lookup_filter = object.lookup_filter
identify = object.identify
```

## LDAP schema extension for the example module

Before the developed module can be used within the Univention Directory Manager, the new object class, in this case `testPhone`, must be made known to the LDAP server together with its attributes. Such object definitions are defined via so-called schemas in LDAP. They are specified in files looking like the following:

```

attributetype ( 1.3.6.1.4.1.10176.9999.1.1 NAME 'testPhoneActive'
  DESC 'state of the IP phone'
  EQUALITY caseIgnoreIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

attributetype ( 1.3.6.1.4.1.10176.9999.1.2 NAME 'testPhoneProtocol'
  DESC 'The supported VoIP protocol'
  EQUALITY caseExactIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

attributetype ( 1.3.6.1.4.1.10176.9999.1.3 NAME 'testPhoneIP'
  DESC 'The IP address of the phone'
  EQUALITY caseExactIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

attributetype ( 1.3.6.1.4.1.10176.9999.1.4 NAME 'testPhonePrimaryUser'
  DESC 'The primary user of the phone'
  EQUALITY caseIgnoreIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )

attributetype ( 1.3.6.1.4.1.10176.9999.1.5 NAME 'testPhoneUsers'
  DESC 'A list of other users allowed to use the phone'
  EQUALITY caseIgnoreIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26 )

objectclass ( 1.3.6.1.4.1.10176.9999.2.1 NAME 'testPhone'
  DESC 'IP Phone'
  SUP top STRUCTURAL
  MUST ( cn $ testPhoneActive $ testPhoneProtocol $ testPhoneIP $
↪testPhonePrimaryUser )
  MAY ( testPhoneUsers )
  )

```

Detailed documentation on creating LDAP schema files can be found on the [OpenLDAP project website](https://www.openldap.org/)<sup>203</sup> and is not the focus of this documentation.

## Installing the module

The last step is to install the Python module and LDAP schema, documented in the following.

The Python module must be copied to the `/usr/lib/python3/dist-packages/univention/admin/handlers/` directory for the Univention Directory Manager to find it. In this directory a subdirectory has to be created corresponding to the first part of the module name. For example, if the module name is `test/ip-phone`, the directory should be named `test/`. The Python module must then be copied to this directory. Ideally, a UDM module is integrated into a separate Debian package.

Documentation for this can be found in the *Introduction* (page 3) section. The newly created package will now be included in the display when `univention-directory-manager modules` is called.

In principle, the file containing the LDAP schema can be copied to any directory. Univention schema definitions, for example, are stored in the `/usr/share/univention-ldap/schema/` directory. For the LDAP server to find this schema, it must be included in the `/etc/ldap/slapd.conf` configuration file. Since this file is under the control

<sup>203</sup> <https://www.openldap.org/>

of the Univention Configuration Registry, do not edit the file directly, but create a Univention Configuration Registry template. (see *UCR Template files conffiles/path/to/file* (page 21))

### Downloading the sample code

The latest version of the sample code can be found at UCS source: [packaging/univention-directory-manager-module-example](https://github.com/univention/univention-directory-manager-module-example)<sup>204</sup>.

It contains a source package in Debian format from which two binary packages are created during package building through `./debian/rules binary`: A schema package that needs to be installed on the master and the package containing the UDM module itself. The sample code also includes a script `ip-phone-tool`, which exemplifies the use of the UDM Python API in a Python script.

## 7.2 UDM syntax

Every UDM property has a syntax, which is used to check the value for correctness. Univention Corporate Server already provides several syntax types, which are defined in the Python file `/usr/lib/python3/dist-packages/univention/admin/syntax.py`. The following syntax list is not complete. For a complete overview, consult the file directly.

**string; string64; OneThirdString; HalfString; TwoThirdsString; FourThirdsString; OneAndAHalfString; FiveThirdsString; TextArea**

Different string classes, which are mapped in Univention Management Console to text input widgets with different widths and heights.

**string\_numbers\_letters\_dots; string\_numbers\_letters\_dots\_spaces; IA5string; ...**

Different string classes with restrictions on the allowed character set.

**Upload; Base64Upload; jpegPhoto**

Binary data.

**integer**

Positive integers.

**boolean; booleanNone; TrueFalse; TrueFalseUpper; TrueFalseUp**

Different boolean types which map to `yes` and `no` or `true` and `false`.

**hostName; DNS\_Name; windowsHostName; ipv4Address; ipAddress; hostOrIP; v4netmask; netmask; IPv4\_AddressRange; IP\_AddressRange; ...**

Different classes for host names or addresses.

**unixTime; TimeString; iso8601Date; date**

Date and time.

**GroupDN; UserDN; UserID; HostDN; DomainController; Windows\_Server; UCS\_Server; ...**

Dynamic classes, which do an LDAP search to provide a list of selectable values like users, groups and hosts.

**LDAP\_Search, UDM\_Objects, UDM\_Attribute**

These syntaxes do an LDAP search and display the result as a list. They are further described in *UDM LDAP search* (page 84).

Additional syntax classes can be added by placing a Python file in `/usr/lib/python3/dist-packages/univention/admin/syntax.d/`. They're automatically imported by UDM.

### 7.2.1 UDM syntax override

Sometimes the predefined syntax is inappropriate in some scenarios. This can be because of performance problems with LDAP searches or the need for more restrictive or lenient value checking. The latter case might require a change to the LDAP schema, since `slapd` also checks the provided values for correctness.

<sup>204</sup> <https://github.com/univention/univention-corporate-server/tree/5.2-5/packaging/univention-directory-manager-module-example/>

The syntax of UDM properties can be overwritten by using Univention Configuration Registry Variables. For each module and each property the variable `directory/manager/web/modules/module/properties/property/syntax` can be set to the name of a syntax class. For example `directory/manager/web/modules/users/user/properties/username/syntax=uid` would restrict the name of users to not contain umlauts.

Since UCR variables only affect the local system, the variables must be set on all systems where UDM is used. This can be either done through a Univention Configuration Registry policy or by setting the variable in the `.postinst` script of some package, which is installed on all hosts.

## 7.2.2 UDM LDAP search

It is often required to present a list of entries to the user, from which they can select one or — in case of a multi-valued property — more entries. Several syntax classes derived from `select` provide a fixed list of choices. If the set of values is known and fixed, it's best to derive an own class from `select` and provide the Python file in `/usr/lib/python3/dist-packages/univention/admin/syntax.d/`.

If on the other hand the list is dynamic and is stored in LDAP, UDM provides three methods to retrieve the values.

### **class UDM\_Attribute**

This class does a UDM search. For each object found all values of a multi-valued property are returned.

For a derived class the following class variables can be used to customize the search:

#### **udm\_module**

The name of the UDM module, which does the LDAP search and retrieves the properties.

#### **udm\_filter**

An LDAP search filter which is used by the UDM module to filter the search. The special value `dn` skips the search and directly returns the property of the UDM object specified by `depends`.

#### **attribute**

The name of a multi-valued UDM property which stores the values to be returned.

#### **is\_complex; key\_index; label\_index**

Some UDM properties consist of multiple parts, so called complex properties. These variables are used to define, which part is displayed as the label and which part is used to reference the entry.

#### **label\_format**

A Python format string, which is used to format the UDM properties to a label string presented to the user. `%(property-name)s` should be used to reference properties. The special property name `static_value$` is replaced by the value of variable `static_value` declared above.

#### **regex**

This defines an optional regular expression, which is used in the front end to check the value for validity.

#### **static\_values**

A list of two-tuples `(value, display-string)`, which are added as additional selection options.

#### **empty\_value**

If set to `True`, the empty value is inserted before all other static and dynamic entries.

#### **depends**

This variable may contain the name of another property, which this property depends on. This can be used to link two properties. For example, one property can be used to select a server, while the second dependent property then only lists the services provided by that selected host. For the dependent syntax `attribute` must be set to `dn`.

#### **error\_message**

This error message is shown when the user enters a value which is not in the set of allowed values.

The following example syntax would provide a list of all users with their telephone numbers:

```
class DelegateTelephonedNumber(UDM_Attribute):
    udm_module = 'users/user'
    attribute = 'phone'
    label_format = '%(displayName)s: %($attribute)s'
```

### class UDM\_Objects

This class performs a UDM search returning each object found.

For a derived class the following class variables can be used to customize the search:

#### udm\_modules

A List of one or more UDM modules, which do the LDAP search and retrieve the properties.

#### key

A Python format string generating the key value used to identify the selected object. The default is `dn`, which uses the distinguished name of the object.

#### label

A Python format string generating the display label to represent the selected object. The default is `None`, which uses the UDM specific `description`. `dn` can be used to use the distinguished name.

#### regex

This defines an optional regular expression, which is used in the frontend to check the value for validity. By default only valid distinguished names are accepted.

#### simple

By default a widget for selecting multiple entries is used. Setting this variable to `True` changes the widget to a combo-box widget, which only allows to select a single value. This should be in-sync with the `multivalue` property of UDM properties.

#### use\_objects

By default UDM opens each LDAP object through a UDM module implemented in Python. This can be a performance problem if many entries are returned. Setting this to `False` disables the Python code and directly uses the attributes returned by the LDAP search. Several properties can then no longer be used as key or label, as those are not explicitly stored in LDAP but are only calculated by the UDM module. For example, to get the fully qualified domain name of a host `%(name)s.%(domain)s` must be used instead of the calculated property `%(fqdn)s`.

#### udm\_filter; static\_values; empty\_value; depends; error\_message

Same as above with `UDM_Attribute` (page 84).

The following example syntax would provide a list of all servers providing a required service:

```
class MyServers(UDM_Objects):
    udm_modules = (
        'computers/domaincontroller_master',
        'computers/domaincontroller_backup',
        'computers/domaincontroller_slave',
        'computers/memberserver',
    )
    label = '%(fqdn)s'
    udm_filter = 'service=MyService'
```

### class LDAP\_Search

This is the old implementation, which should only be used, if `UDM_Attribute` (page 84) and `UDM_Objects` (page 85) are not sufficient. In addition to ease of use it has the drawback that Univention Management Console can not do as much caching, which can lead to severe performance problems.

LDAP search syntaxes can be declared in two equivalent ways:

### Python API

By implementing a Python class derived from `LDAP_Search` (page 85) and providing that implementation in `/usr/lib/python3/dist-packages/univention/admin/syntax.d/`.

### UDM API

By creating a UDM object in LDAP using the module `settings/syntax`.

`class Python_API (LDAP_Search)`

The Python API uses the following variables:

#### **syntax\_name**

This variable stores the common name of the LDAP object, which is used to define the syntax. It is only used internally and should never be needed when creating syntaxes programmatically.

#### **filter**

An LDAP filter to find the LDAP objects providing the list of choices.

#### **attribute**

A list of UDM module property definitions like “`shares/share: dn`”. They are used as the human readable label for each element of the choices.

#### **value**

The UDM module attribute that will be stored to identify the selected element. The value is specified like `shares/share: dn`

#### **viewonly**

If set to `True` the values can not be changed.

#### **addEmptyValue**

If set to `True` the empty value is add to the list of choices.

#### **appendEmptyValue**

Same as `addEmptyValue` but added at the end. Used to automatically choose an existing entry in the frontend.

```
class MyServers (LDAP_Search) :
    def __init__(self) :
        LDAP_Search.__init__(self,
            filter=(' (&(univentionService=MyService) '
                '(univentionServerRole=member)) '),
            attribute=(
                'computers/memberserver: fqdn',
            ),
            value='computers/memberserver: dn'
        )
        self.name = 'LDAP_Search' # required workaround
```

`class LDAP_Search.UDM_API`

The UDM API uses the following properties:

#### **name**

(required)

The name for the syntax.

#### **description**

(optional)

Some descriptive text.

**filter**

(required)

An LDAP filter, which is used to find the objects.

**base**

(optional)

The LDAP base, where the search starts.

**attribute**

(optional, multi-valued)

The name of UDM properties, which are display as a label to the user. Alternatively LDAP attribute names may be used directly.

**ldapattribute**

(optional, multi-valued)

Description, see *attribute* (page 87).**value**

(optional);

The name of the UDM property, which is used to reference the object. Alternatively an LDAP attribute name may be used directly.

**ldapvalue**

(optional)

Description, see *value* (page 87).**viewonly**

(optional)

If set to `True` the values can not be changed.**addEmptyValue**

(optional)

If set to `True` the empty value is add to the list of choices.

```

$ eval "$(ucr shell) "
$ udm settings/syntax create "$@" --ignore_exists \
  --position "cn=custom attributes,cn=univention,$ldap_base" \
  --set name="MyServers" \
  --set filter='(&(univentionService=MyService)(univentionServerRole=member))' \
  --set attribute='computers/memberserver: fqdn' \
  --set value='computers/memberserver: dn'

```

## 7.3 Package extended attributes

Each UDM module provides a set of mappings from LDAP attributes to properties. This set is not complete, because LDAP objects can be extended with additional *auxiliary objectClasses*. *Extended Attributes* can be used to extend modules to show additional properties. These properties can be mapped to any already defined LDAP attribute, but objects can also be extended by adding additional auxiliary object classes, which can provide new attributes.

For packing purpose any additional LDAP schema needs to be registered on the Primary Directory Node, which is replicated from there to all other domain controllers through the Listener/Notifier mechanism (see *Univention Directory Listener* (page 43)). This is best done trough a separate schema package, which should be installed on the Primary Directory Node and Backup Directory Node. Since *Extended Attributes* are declared in LDAP, the commands to create them can be put into any join script (see *Domain join* (page 29)). To be convenient, the declaration should be also

included with the schema package, since installing it there does not require the Administrator to provide additional LDAP credentials.

An *Extended Attribute* is created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. The module is called `settings/extended_attribute`. *Extended Attributes* can be stored anywhere in the LDAP, but the default location would be `cn=custom attributes,cn=univention`, below the LDAP base. Since the join script creating the attribute may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The module `settings/extended_attribute` requires many parameters. They are described in [Expansion of UMC modules with extended attributes](#)<sup>205</sup> in *Univention Corporate Server - Manual for users and administrators* [2].

**name (required)**

Name of the attribute.

**CLIName (required)**

An alternative name for the command line version of UDM.

**shortDescription (required)**

Default short description.

**translationShortDescription (optional, multiple)**

Translation of short description.

**longDescription (required)**

Default long description.

**translationLongDescription (optional, multiple)**

Translation of long description.

**objectClass (required)**

The name of an LDAP object class which is added to store this property.

**deleteObjectClass (optional)**

Remove the object class when the property is unset.

**ldapMapping (required)**

The name of the LDAP attribute the property matches to.

**syntax (optional)**

A syntax class, which also controls the visual representation in UDM. Defaults to `string`.

**default (optional)**

The default value is used when a new UDM object is created.

**valueRequired (optional)**

A value must be entered for the property.

**multivalue (optional)**

Controls if only a single value or multiple values can be entered. This must be in sync with the `SINGLE-VALUE` setting of the attribute in the LDAP schema.

**mayChange (optional)**

The property may be modified later.

**notEditable (optional)**

Disable all modification of the property, even when the object is first created. The property is only modified through hooks.

**copyable (optional)**

Copy the value of the property when the entry is cloned.

**hook (optional)**

The name of a Python class implementing hook functions. See *Extended attribute hooks* (page 94) for more information.

---

<sup>205</sup> <https://docs.software-univention.de/manual/5.2/en/central-management-umc/extended-attributes.html#central-extended-attribs>

**doNotSearch (optional)**

If this is enabled, the property is not show in the drop-down list of properties when searching for UDM objects.

**tabName (optional)**

The name of the tab in the UMC where the property should be displayed. The name of existing tabs can be copied from UMC session with the `English` locale. A new tab is automatically created for new names.

**translationTabName (optional, multiple)**

Translation of tab name.

**tabPosition (optional)**

This setting is only relevant, when a new tab is created by using a `tabName`, for which no tab exists. The integer value defines the position where the newly tab is inserted. By default the newly created tab is appended at the end, but before the *Extended settings* tab.

**overwriteTab (optional)**

If enabled, the tab declared by the UDM module with the name from the `tabName` settings is replaces by a new clean tab with only the properties defined by *Extended Attributes*.

**tabAdvanced (optional)**

If this setting is enabled, the tab is created inside the *Extended settings* tab instead of being a tab by its own.

**groupName (optional)**

The name of the group inside a tab where the property should be displayed. The name of existing groups can be copied from UMC session with the `English` locale. A new tab is automatically created for new names. If no name is given, the property is placed before the first tab.

**translationGroupName (optional, multiple)**

Translation of group name.

**groupPosition (optional)**

This setting is only relevant, when a new group is created by using a `groupName`, for which no group exists. The integer value defines the position where the newly group is inserted. By default the newly created group is appended at the end.

**overwritePosition (optional)**

The name of an existing property this property wants to overwrite.

**preventUmcDefaultPopup (optional)**

If this setting is enabled, the pop-up that is shown when the default value automatically in UMC is suppressed.

**disableUDMweb (optional)**

Disables showing this property in the UMC.

**fullwidth (optional)**

The widget for the property should span both columns.

**module (required, multiple)**

A list of module names where this *Extended Attribute* should be added to.

**options (required, multiple)**

A list of options, which enable this *Extended Attribute*.

**version (required)**

The version of the *Extended Attribute* format. The current version is 2.

**Tip**

Create the *Extended Attribute* first through UMC-UDM. Modify it until you're satisfied. Only then dump it using `udm settings/extended_attribute list` and convert the output to an equivalent shell script creating it.

The following example provides a simple LDAP schema called `extended-attribute.schema`, which declares one object class `univentionExamplesUdmOC` and one attribute `univentionExamplesUdmAttribute`.

Listing 7.1: *Extended Attribute* for custom LDAP schema

```

#objectIdentifier univention 1.3.6.1.4.1.10176
#objectIdentifier univentionCustomers univention:99999
#objectIdentifier univentionExamples univentionCustomers:0
objectIdentifier univentionExamples 1.3.6.1.4.1.10176:99999:0
objectIdentifier univentionExmaplesUdm univentionExamples:1
objectIdentifier univentionExmaplesUdmAttributeType univentionExmaplesUdm:1
objectIdentifier univentionExmaplesUdmObjectClass univentionExmaplesUdm:2

attributetype ( univentionExmaplesUdmAttributeType:1
    NAME 'univentionExamplesUdmAttribute'
    DESC 'An example attribute for UDM'
    EQUALITY caseIgnoreMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{42}
    SINGLE-VALUE
)

objectClass ( univentionExmaplesUdmObjectClass:1
    NAME 'univentionExamplesUdmOC'
    DESC 'An example object class for UDM'
    SUP top
    AUXILIARY
    MUST ( univentionExamplesUdmAttribute )
)

```

The schema is shipped as `/usr/share/extended-attribute/extended-attribute.schema` and installed by calling `ucs_registerLDAPExtension` from the `join-script 50extended-attribute.inst`.

```

#!/bin/bash

## joinscript api: bindpwdfile

VERSION=1
. /usr/share/univention-join/joinscripthelper.lib
. /usr/share/univention-lib/ldap.sh
joinscript_init

# register LDAP schema for new extended attribute
ucs_registerLDAPExtension "$@" \
    --schema /usr/share/extended-attribute/extended-attribute.schema

# Register new service entry for this host
eval "$(ucr shell)"
udm settings/extended_attribute create "$@" --ignore_exists \
    --position "cn=custom attributes,cn=univention,$ldap_base" \
    --set name="My Attribute" \
    --set CLIName="myAttribute" \
    --set shortDescription="Example attribute" \
    --append translationShortDescription="'de_DE' 'Beispielattribut'" \
    --append translationShortDescription="'fr_FR' 'Exemple d'attribut'" \
    --set longDescription="An example attribute" \
    --append translationLongDescription="'de_DE' 'Ein Beispielattribut'" \
    --append translationLongDescription="'fr_FR' 'Un exemple d'attribut'" \
    --set tabAdvanced=1 \
    --set tabName="Examples" \

```

(continues on next page)

(continued from previous page)

```

--append translationTabName='"de_DE" "Beispiele"' \
--append translationTabName='"fr_FR" "Exemples"' \
--set tabPosition=1 \
--set module="groups/group" \
--set module="computers/memberserver" \
--set syntax=string \
--set default="Lorem ipsum" \
--set multivalue=0 \
--set valueRequired=0 \
--set mayChange=1 \
--set doNotSearch=1 \
--set objectClass=univentionExamplesUdmOC \
--set ldapMapping=univentionExamplesUdmAttribute \
--set deleteObjectClass=0
# --set overwritePosition=
# --set overwriteTab=
# --set hook=
# --set options=

# Terminate UDM server to force module reload
. /usr/share/univention-lib/base.sh
stop_udm_cli_server

joinscript_save_current_version
exit 0

```

This example is deliberately missing an unjoin-script (see *Writing unjoin scripts* (page 37)) to keep this example simple. It should check if the *Extended Attribute* is no longer used in the domain and then remove it.

### 7.3.1 Selection lists

Sometimes an *Extended Attribute* should show a list of options to choose from. This list can either be static or dynamic. After defining such a new syntax it can be used by referencing its name in the `syntax` property of an *Extended Attribute*.

#### Static selections

The static list of available selections is defined once and can not be modified interactively through UMC. Such a list is best implemented though a custom syntax class. As the implementation must be available on all system roles, the new syntax is best registered in LDAP. This can be done by using *ucs\_registerLDAPExtension* (page 35) which is described in *join.sh* (page 34).

As an alternative the file can be put into the directory `/usr/lib/python3/dist-packages/univention/admin/syntax.d/`.

The following example is comparable to the default example in file `/usr/lib/python3/dist-packages/univention/admin/syntax.d/example.py`:

```

class StaticSelection(select):
    choices = [
        ('value1', 'Description for selection 1'),
        ('value2', 'Description for selection 2'),
        ('value3', 'Description for selection 3'),
    ]

```

## Dynamic selections

A dynamic list is implemented as an LDAP search, which is described in *UDM LDAP search* (page 84). For performance reason it is recommended to implement a class derived from *UDM\_Attribute* (page 84) or *UDM\_Objects* (page 85) instead of using *LDAP\_Search* (page 85). The file `/usr/lib/python3/dist-packages/univention/admin/syntax.py` contains several examples.

The idea is to create a container with sub-entries for each selection. This following listing declares a new syntax class for selecting a profession level.

Listing 7.2: Dynamic selection list for *Extended Attributes*

```
class DynamicSelection(UDM_Objects):
    udm_modules = ('container/cn',)
    udm_filter = '(&(objectClass=organizationalRole)(ou:dn:=DynamicSelection))'
    simple = True # only one value is selected
    empty_value = True # allow selecting nothing
    key = '%(name)s' # this is stored
    label = '%(description)s' # this is displayed
    regex = None # no validation in frontend
    error_message = 'Invalid value'
```

The Python code should be put into a file named `DynamicSelection.py`. The following code registers this new syntax in LDAP and adds some values. It also creates an *Extended Attribute* for user objects using this syntax.

```
$ syntax='DynamicSelection'
$ base="cn=univention, $(ucr get ldap/base)"

$ udm container/ou create \
  --position "$base" \
  --set name="$syntax" \
  --set description='UCS profession level'
$ dn="ou=$syntax,$base"

$ udm container/cn create \
  --position "$dn" \
  --set name="value1" \
  --set description='UCS Guru (> 5)'

$ udm container/cn create \
  --position "$dn" \
  --set name="value2" \
  --set description='UCS Regular (1..5)'

$ udm container/cn create \
  --position "$dn" \
  --set name="value3" \
  --set description='UCS Beginner (< 1)'

$ udm container/cn create \
  --ignore_exists \
  --position "$base" \
  --set name='udm_syntax'
$ dn="cn=udm_syntax,$base"

$ udm settings/udm_syntax create \
  --position "$dn" \
  --set name="$syntax" \
  --set filename="DynamicSelection.py" \
```

(continues on next page)

(continued from previous page)

```

--set data="$ (bzip2 <DynamicSelection.py | base64) " \
--set package="$syntax" \
--set packageversion="1"

$ udm settings/extended_attribute create \
--position "cn=custom attributes,$base" \
--set name='Profession' \
--set module='users/user' \
--set tabName='General' \
--set translationTabName='"de_DE" "Allgemein"' \
--set groupName='Personal information' \
--set translationGroupName='"de_DE" "Persönliche Informationen"' \
--set shortDescription='UCS profession level' \
--set translationShortDescription='"de_DE" "UCS Erfahrung"' \
--set longDescription='Select a level of UCS experience' \
--set translationLongDescription='"de_DE" "Wählen Sie den Level der Erfahrung_
↳mit UCS"' \
--set objectClass='univentionFreeAttributes' \
--set ldapMapping='univentionFreeAttribute1' \
--set syntax="$syntax" \
--set mayChange=1 \
--set valueRequired=0

```

### 7.3.2 Known issues

- The `tabName` and `groupName` values must exactly match the values already used in the modules. If they do not match, a new tab or group is added. This also applies to the translation: They must match the already translated strings and must be repeated for every *Extended Attribute* again and again. The untranslated strings are best extracted directly from the Python source code of the modules in `/usr/lib/python3/dist-packages/univention/admin/handlers/**/*.py`. For the translated strings run `msgunfmt /usr/share/locale/$language-code/LC_MESSAGES/univention-admin*.mo`.
- The `overwritePosition` values must exactly match the name of an already defined property. Otherwise UDM will crash.
- *Extended Attributes* may be removed, when matching data is still stored in LDAP. The schema on the other hand must only be removed when all matching data is removed. Otherwise the server `slapd` will fail to start.
- Removing `objectClasses` from LDAP objects must be done manually. Currently UDM does not provide any functionality to remove unneeded object classes or methods to force-remove an object class including all attributes, for which the object class is required.

### 7.3.3 Extended options

UDM properties can be enabled and disabled through options. For example, all properties of a user related to Samba can be switched *on* or *off* to reduce the settings shown to an administrator. If many *Extended Attributes* are added to a UDM module, it might prove necessary to also create new options. Options are per UDM module.

Similar to *Extended Attributes* an *Extended Option* is created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. The module is called `settings/extended_options`. *Extended Options* can be stored anywhere in the LDAP, but the default location would be `cn=custom attributes, cn=univention,` below the LDAP base. Since the join script creating the option may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The module `settings/extended_options` has the following properties:

**name (required)**

Name of the option.

**shortDescription (required)**

Default short description.

**translationShortDescription (optional, multiple)**

Translation of short description.

**longDescription (required)**

Default long description.

**translationLongDescription (optional, multiple)**

Translation of long description.

**default (optional)**

Enable the option by default.

**editable (optional)**

Option may be repeatedly turned on and off.

**module (required, multiple)**

A list of module names where this *Extended Option* should be added to.

**objectClass (optional, multiple)**

A list of LDAP object classes, which when found, enable this option.

Listing 7.3: *Extended Option*

```
$ eval "$(ucr shell)"
$ udm settings/extended_options create "$@" --ignore_exists \
--position "cn=custom attributes,cn=univention,$ldap_base" \
--set name="My Option" \
--set shortDescription="Example option" \
--set translationShortDescription="'de_DE' 'Beispieloption'" \
--set longDescription="An example option" \
--set translationLongDescription="'de_DE' 'Eine Beispieloption'" \
--set default=0 \
--set editable=0 \
--set module="users/user" \
--set objectClass=univentionExamplesUdmOC
```

### 7.3.4 Extended attribute hooks

Hooks provide a mechanism to pre- and post-process the values of *Extended Attributes*. Normally, UDM properties are stored as-is in LDAP attributes. Hooks can modify the LDAP operations when an object is created, modified, deleted or retrieved. They are implemented in Python and the file must be placed in the directory `/usr/lib/python3/dist-packages/univention/admin/hooks.d/`. The filename must end with `.py`.

The module `univention.admin.hook` (page 94) provides the class `simpleHook`, which implements all required hook functions. By default they don't modify any request, but do log all calls. This class should be used as a base class for inheritance.

```
class univention.admin.hook.simpleHook
```

```
    map (value, encoding)
```

Added in version 5.2-6: The corresponding `map` and `unmap` methods are available since UCS 5.2 erratum 416<sup>206</sup>.

**Parameters**

- **value** (*Any*) – The UDM property value
- **encoding** (*tuple*<sup>207</sup>) – Optional encoding information

**Returns**

The list of LDAP attributes.

**Return type**`list208[bytes209]`

This method can be overwritten to define a special method to map an UDM property value to LDAP attribute value.

**unmap** (*value*, *encoding*)

Added in version 5.2-6: The corresponding `map` and `unmap` methods are available since UCS 5.2 erratum 416<sup>210</sup>.

**Parameters**

- **value** (`list211[bytes212]`) – The list of LDAP attributes
- **encoding** (`tuple213`) – Optional encoding information

**Returns**

The UDM property value.

**Return type**

Any

This method can be overwritten to define a special un-map method to map a LDAP attribute value to an UDM property value.

**hook\_open** (*obj*)

**Parameters**

**obj** (`univention.admin.handlers.simpleLdap`)

**Return type**

None

This method is called by the default `open()` handler just before the current state of all properties is saved.

**hook\_ldap\_pre\_create** (*obj*)

**Parameters**

**obj** (`univention.admin.handlers.simpleLdap`)

**Return type**

None

This method is called before a UDM object is created. It is called after the module validated all properties, but before the *add-list* is created.

**hook\_ldap\_addlist** (*obj*, *al*: `AddList = []`)

**Parameters**

- **obj** (`univention.admin.handlers.simpleLdap`)
- **al** (`AddList`)

**Return type**

`AddList`

This method is called before a UDM object is created. It gets passed a list of two-tuples (`ldap-attribute-name`, `list-of-values`), which will be used to create the LDAP object. The method must return the (modified) list. Notice that `hook_ldap_modlist()` (page 96) will also be called next.

**hook\_ldap\_post\_create** (*obj*)

**Parameters**

**obj** (`univention.admin.handlers.simpleLdap`)

**Return type**

None

This method is called after the object was created in LDAP.

`hook_ldap_pre_modify (obj)`

**Parameters**

`obj` (*univention.admin.handlers.simpleLdap*)

**Return type**

None

This method is called before a UDM object is modified. It is called after the module validated all properties, but before the *modification-list* is created.

`hook_ldap_modlist (obj, ml: ModList = [])`

**Parameters**

- `obj` (*univention.admin.handlers.simpleLdap*)
- `ml` (*ModList*)

**Return type**

ModList

This method is called before a UDM object is created or modified. It gets passed a list of tuples, which are either two-tuples (*ldap-attribute-name, list-of-new-values*) or three-tuples (*ldap-attribute-name, list-of-old-values, list-of-new-values*). It will be used to create or modify the LDAP object. The method must return the (modified) list.

`hook_ldap_pre_remove (obj)`

**Parameters**

`obj` (*univention.admin.handlers.simpleLdap*)

**Return type**

None

This method is called before a UDM object is removed.

`hook_ldap_post_remove (obj)`

**Parameters**

`obj` (*univention.admin.handlers.simpleLdap*)

**Return type**

None

This method is called after the object was removed from LDAP.

## Examples

- The following example implements a hook, which removes the object-class `univentionFreeAttributes`, if the property `isSampleUser` is no longer set.

```
from univention.admin.hook import simpleHook

class RemoveObjClassUnused(simpleHook):
    type = 'RemoveObjClassUnused'
```

(continues on next page)

<sup>206</sup> <https://errata.software-univention.de/#!/erratum=5.2x416>

<sup>207</sup> <https://docs.python.org/3/library/stdtypes.html#tuple>

<sup>208</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>209</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

<sup>210</sup> <https://errata.software-univention.de/#!/erratum=5.2x416>

<sup>211</sup> <https://docs.python.org/3/library/stdtypes.html#list>

<sup>212</sup> <https://docs.python.org/3/library/stdtypes.html#bytes>

<sup>213</sup> <https://docs.python.org/3/library/stdtypes.html#tuple>

(continued from previous page)

```
def hook_ldap_post_modify(self, obj):
    """Remove unused objectClass."""
    ext_attr_name = 'isSampleUser'
    class_name = 'univentionFreeAttributes'

    if obj.ldapinfo.get(ext_attr_name) in ('1',) and \
        obj.info.get(ext_attr_name) in ('0', None):
        if class_name in obj.ldapattr.get('objectClass', []):
            obj.ldapmodify(obj.dn,
                [('objectClass', class_name, '')])
```

After installing the file, the hook can be activated by setting the `hook` property of an *Extended Attribute* to `RemoveObjClassUnused`:

```
$ udm settings/extended_attribute modify \
  --dn ... \
  --set hook=RemoveObjClassUnused
```

**Note**

This is just an example to demonstrate UDM hooks for extended attributes. The actual functionality can be achieved by setting the `deleteObjectClass` flag for extended attributes.

- In this example we have an extended attribute with the UDM date syntax `iso8601Date` (1970-01-01). But we want to store the value as a `GeneralizedTime` string (19700101000000Z) in LDAP. We can map the values between these formats by providing the appropriate `map` and `unmap` functions in the hook.

Added in version 5.2-6: The corresponding `map` and `unmap` methods are available since UCS 5.2 erratum 416<sup>214</sup>.

```
from univention.admin.hook import simpleHook

class myDate(simpleHook):
    type = 'myDate'
    ldap_date_format = "%Y%m%d%H%M%SZ"
    udm_date_format = "%Y-%m-%d"

    def unmap(self, value: list[bytes], encoding=()) -> str:
        """Convert [b'20090101000000Z'] to u'2009-01-01'. Ignores timezones."""
        ↪
        ldap_val = value[0].decode(*encoding)
        ldap_date = datetime.datetime.strptime(ldap_val, self.ldap_date_
        ↪format)
        return ldap_date.strftime(self.udm_date_format)

    def map(self, value: str, encoding=()) -> list[bytes]:
        """Convert u'2009-01-01' to [b'20090101000000Z']. Ignores timezones."""
        ↪
        if not value:
            return []
        udm_date = datetime.datetime.strptime(value, self.udm_date_format)
        ldap_date = udm_date.strftime(self.ldap_date_format)
        return [udm_date.encode(*encoding)]
```

<sup>214</sup> <https://errata.software-univention.de/#/erratum=5.2x416>

## 7.4 Package UDM hooks

For some purposes, for example for app installation, it is convenient to be able to deploy a new UDM hook in the UCS domain from any system in the domain. For this purpose, a UDM hook can be stored as a special type of UDM object. The module responsible for this type of objects is called `settings/udm_hook`. As these objects are replicated throughout the UCS domain, the UCS servers listen for modifications on these objects and integrate them into the local UDM.

The commands to create the UDM hook objects in UDM may be put into any join script (see *Domain join* (page 29)). Like every UDM object a UDM hook object can be created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. UDM hook objects can be stored anywhere in the LDAP directory, but the recommended location would be `cn=udm_hook, cn=univention`, below the LDAP base. Since the join script creating the attribute may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The module `settings/udm_hook` requires several parameters. Since many of these are determined automatically by the `ucs_registerLDAPExtension` shell library function, it is recommended to use the shell library function to create these objects (see *join.sh* (page 34)).

**name (required)**

Name of the UDM hook.

**data (required)**

The actual UDM hook data in bzip2 and base64 encoded format.

**filename (required)**

The filename the UDM hook data should be written to by the listening servers. The filename must not contain any path elements.

**package (required)**

Name of the Debian package which creates the object.

**packageversion (required)**

Version of the Debian package which creates the object. For object modifications the version number needs to increase unless the package name is modified as well.

**appidentifier (optional)**

The identifier of the app which creates the object. This is important to indicate that the object is required as long as the app is installed anywhere in the UCS domain. Defaults to `string`.

**ucsversionstart (optional)**

Minimal required UCS version. The UDM hook is only activated by systems with a version higher than or equal to this.

**ucsversionend (optional)**

Maximal required UCS version. The UDM hook is only activated by systems with a version lower than or equal to this. To specify validity for the whole 5.0-x release range a value like `5.0-99` may be used.

**active (internal)**

A boolean flag used internally by the Primary Directory Node to signal availability of the new UDM hook on the Primary Directory Node (default: `FALSE`).

## 7.5 Package UDM extension modules

For some purposes, for example for app installation, it is convenient to be able to deploy a new UDM module in the UCS domain from any system in the domain. For this purpose, a UDM module can be stored as a special type of UDM object. The module responsible for this type of objects is called `settings/udm_module`. As these objects are replicated throughout the UCS domain, the UCS servers listen for modifications on these objects and integrate them into the local UDM.

The commands to create the UDM module objects in UDM may be put into any join script (see *Domain join* (page 29)). Like every UDM object a UDM module object can be created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. UDM module objects can be stored anywhere in the

LDAP directory, but the recommended location would be `cn=udm_module,cn=univention`, below the LDAP base. Since the `join` script creating the attribute may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The module `settings/udm_module` requires several parameters. Since many of these are determined automatically by the `ucs_registerLDAPExtension` shell library function, it is recommended to use the shell library function to create these objects (see *join.sh* (page 34)).

**name (required)**

Name of the UDM module, e.g. `newapp/someobject`.

**data (required)**

The actual UDM module data in bzip2 and base64 encoded format.

**filename (required)**

The filename the UDM module data should be written to by the listening servers. The filename may contain path elements and should conform to the name of the UDM module (e.g. `newapp/someobject.py`).

**messagecatalog (optional)**

Multi-valued property to supply message translation files (syntax: `<language tag> <base64 encoded GNU message catalog>`).

**umregistration (optional)**

XML definition required to make the UDM module available through the Univention Management Console (base64 encoded XML)

**icon (optional)**

Multi-valued property to supply icons for the Univention Management Console (base64 encoded `png`, `jpeg` or `svgz`).

**package (required)**

Name of the Debian package which creates the object.

**packageversion (required)**

Version of the Debian package which creates the object. For object modifications the version number needs to increase unless the package name is modified as well.

**appidentifier (optional)**

The identifier of the app which creates the object. This is important to indicate that the object is required as long as the app is installed anywhere in the UCS domain. Defaults to `string`.

**ucsversionstart (optional)**

Minimal required UCS version. The UDM module is only activated by systems with a version higher than or equal to this.

**ucsversionend (optional)**

Maximal required UCS version. The UDM module is only activated by systems with a version lower than or equal to this. To specify validity for the whole 5.0-x release range a value like `5.0-99` may be used.

**active (internal)**

A boolean flag used internally by the Primary Directory Node to signal availability of the new UDM module on the Primary Directory Node (default: `FALSE`).

## 7.6 Package UDM syntax extension

For some purposes, for example for app installation, it is convenient to be able to deploy a new UDM syntax in the UCS domain from any system in the domain. For this purpose, a UDM syntax can be stored as a special type of UDM object. The module responsible for this type of objects is called `settings/udm_syntax`. As these objects are replicated throughout the UCS domain, the UCS servers listen for modifications on these objects and integrate them into the local UDM.

The commands to create the UDM syntax objects in UDM may be put into any `join` script (see *Domain join* (page 29)). Like every UDM object a UDM syntax object can be created by using the UDM command line interface `univention-directory-manager` or its alias `udm`. UDM syntax objects can be stored anywhere in the LDAP directory,

but the recommended location would be `cn=udm_syntax,cn=univention`, below the LDAP base. Since the `join` script creating the attribute may be called on multiple hosts, it is a good idea to add the `--ignore_exists` option, which suppresses the error exit code in case the object already exists in LDAP.

The module `settings/udm_syntax` requires several parameters. Since many of these are determined automatically by the `ucs_registerLDAPExtension` shell library function, it is recommended to use the shell library function to create these objects (see *join.sh* (page 34)).

**name (required)**

Name of the UDM syntax.

**data (required)**

The actual UDM syntax data in bzip2 and base64 encoded format.

**filename (required)**

The filename the UDM syntax data should be written to by the listening servers. The filename must not contain any path elements.

**package (required)**

Name of the Debian package which creates the object.

**packageversion (required)**

Version of the Debian package which creates the object. For object modifications the version number needs to increase unless the package name is modified as well.

**appidentifier (optional)**

The identifier of the app which creates the object. This is important to indicate that the object is required as long as the app is installed anywhere in the UCS domain. Defaults to `string`.

**ucsversionstart (optional)**

Minimal required UCS version. The UDM syntax is only activated by systems with a version higher than or equal to this.

**ucsversionend (optional)**

Maximal required UCS version. The UDM syntax is only activated by systems with a version lower than or equal to this. To specify validity for the whole 5.0-x release range a value like `5.0-99` may be used.

**active (internal)**

A boolean flag used internally by the Primary Directory Node to signal availability of the new UDM syntax on the Primary Directory Node (default: `FALSE`).

## 7.7 UDM HTTP REST API

The content about the *UDM HTTP REST API* moved to another document. Except for the section about *API clients* (page 100), continue reading at [UDM HTTP REST API<sup>215</sup>](#) in *Nubus Customization and Modification Manual 1.x* [3].

### See also

For an architectural overview, see [UDM HTTP REST API<sup>216</sup>](#) in *Univention Corporate Server 5.2 Architecture* [4].

### 7.7.1 API clients

The following API clients implemented in Python exist for the UDM HTTP REST API:

- `python3-univention-directory-manager-rest-client`:

Every UCS system has it installed by default. You can use it the following way:

<sup>215</sup> <https://docs.software-univention.de/nubus-customization/latest/en/api/udm-rest.html#customization-api-udm-rest>

<sup>216</sup> <https://docs.software-univention.de/architecture/5.2/en/services/udm-rest-api.html#services-udm-rest-api>

Listing 7.4: Example for using Python UDM HTTP REST API client

```

from univention.admin.rest.client import UDM

uri = 'https://ucs-primary.example.com/univention/udm/'
udm = UDM.http(uri, 'Administrator', 'univention')
module = udm.get('users/user')

# 1. create a user
obj = module.new()
obj.properties['username'] = 'foo'
obj.properties['password'] = 'univention'
obj.properties['lastname'] = 'foo'
obj.save()

# 2. search for users (first user)
obj = next(module.search('uid=*'))
if obj:
    obj = obj.open()
print('Object {}'.format(obj))

# 3. get by dn
ldap_base = udm.get_ldap_base()
obj = module.get('uid=foo,cn=users,%s' % (ldap_base,))

# 4. get referenced objects e.g. groups
pg = obj.objects['primaryGroup'][0].open()
print(pg.dn, pg.properties)
print(obj.objects['groups'])

# 5. modify
obj.properties['description'] = 'foo'
obj.save()

# 6. move to the ldap base
obj.move(ldap_base)

# 7. remove
obj.delete()

```

- **python3-univention-directory-manager-rest-async-client:**

After installing the Debian package on a UCS system, you can use it in the following way:

Listing 7.5: Example for using Python asynchronous UDM REST API client

```

import asyncio
from univention.admin.rest.async_client import UDM

uri = 'https://ucs-primary.example.com/univention/udm/'

async def main():
    async with UDM.http(uri, 'Administrator', 'univention') as udm:
        module = await udm.get('users/user')

        # 1. create a user
        obj = await module.new()

```

(continues on next page)

(continued from previous page)

```
obj.properties['username'] = 'foo'
obj.properties['password'] = 'univention'
obj.properties['lastname'] = 'foo'
await obj.save()

# 2. search for users (first user)
objs = module.search()
async for obj in objs:
    if not obj:
        continue
    obj = await obj.open()
    print('Object {}'.format(obj))

# 3. get by dn
ldap_base = await udm.get_ldap_base()
obj = await module.get('uid=foo,cn=users,%s' % (ldap_base,))

# 4. get referenced objects e.g. groups
pg = await obj.objects['primaryGroup'][0].open()
print(pg.dn, pg.properties)
print(obj.objects['groups'])

# 5. modify
obj.properties['description'] = 'foo'
await obj.save()

# 6. move to the ldap base
await obj.move(ldap_base)

# 7. remove
await obj.delete()
```

- Python UDM HTTP REST API Client:

- Package at PyPI<sup>217</sup>
- Documentation<sup>218</sup>

<sup>217</sup> <https://pypi.org/project/udm-rest-api-client/>

<sup>218</sup> <https://udm-rest-client.readthedocs.io/en/latest/index.html>

## UNIVENTION MANAGEMENT CONSOLE (UMC)

The Univention Management Console (UMC) is a service that runs on all UCS systems by default. This service provides access to several system information and implements modules for management tasks. What modules are available on a UCS system depends on the system role and the installed components. Each domain user can log on to the service through a web interface. Depending on the access policies for the user the visible modules for management tasks will differ.

In the following the technical details of the architecture and the Python and JavaScript API for modules are described. This chapter has the following content:

### 8.1 Architecture

The Univention Management Console service consists of four components. The communication between these components is encrypted using SSL. The architecture and the communication channels is shown in Fig. 8.1.

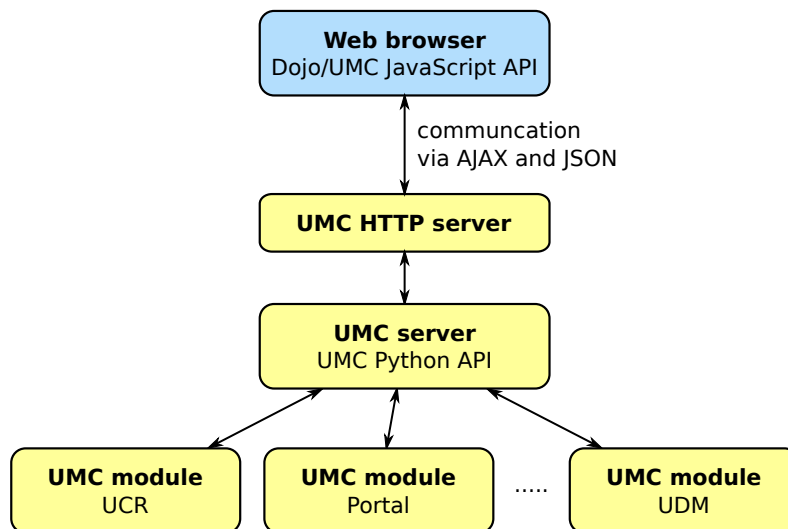


Fig. 8.1: UMC architecture and communication channels

- The *UMC server* is a small web server that provides HTTP access to the modules and manages the connection and verifies that only authorized users get access. It is used by the web frontend.
- The *UMC module* processes are forked by the UMC server to provide a specific area of management tasks within a session.

### 8.2 Protocol HTTP for UMC

With the new generation of UMC there is also an HTTP server available that can be used to access the UMC server.

Listing 8.1: Authentication request

```
POST http://192.0.2.31/univention/auth HTTP/1.1
{"options": {"username": "root", "password": "univention"}}
```

Request:

Listing 8.2: Search for users

```
POST http://192.0.2.31/univention/command/udm/query HTTP/1.1
{"options": {"container": "all",
            "objectType": "users/user",
            "objectProperty": "username",
            "objectPropertyValue": "test1*1"},
 "flavor": "users/user"}
```

Response:

Listing 8.3: Response

```
{"status": 200,
 "message": null,
 "options": {"objectProperty": "username",
            "container": "all",
            "objectPropertyValue": "test1*1",
            "objectType": "users/user"},
 "result": [{"ldap-dn": "uid=test11,cn=users,dc=univention,dc=qa",
            "path": "univention.qa:/users",
            "name": "test11",
            "objectType": "users/user"},
 ...
            {"ldap-dn": "uid=test191,cn=users,dc=univention,dc=qa",
            "path": "univention.qa:/users",
            "name": "test191",
            "objectType": "users/user"}]}
```

## 8.3 UMC files

Files for building a UMC module.

### 8.3.1 `debian/package.umc-modules`

- `univention-110n-build` builds translation files.
- `dh-umc-module-install` installs files.

Configured through `debian/package.umc-modules`.

**Module**

Internal (?) name of the module.

**Python**

Directory containing the Python code relative to top-level directory.

**Definition**

Path to an XML file, which describes the module. See *UMC module declaration file* (page 105) for more information.

**Javascript**

Directory containing the Java-Script code relative to top-level directory.

**Icons (deprecated)**

Directory containing the Icons relative to top-level directory. Must provide icons in sizes 16×16 (`umc/icons/16x16/udm-module.png`) and 50×50 (`umc/icons/50x50/udm-module.png`) pixels.

## 8.3.2 UMC module declaration file

`umc/module.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
SPDX-FileCopyrightText: 2025-2026 Univention GmbH
SPDX-License-Identifier: AGPL-3.0-only
-->

<!--DOCTYPE umc SYSTEM "management/univention-management-console/data/umc-module.
↪dtd"-->
<umc version="2.0">
  <module id="udm" icon="udm-MODULE" priority="50" version="1.0">
    <name>...</name>
    <description>...</description>
    <keywords>...</keywords>
    <flavor>...</flavor>
    <categories>
      <category name="domain"/>
    </categories>
    <command>...</command>
  </module>
</umc>
```

`umc/categories/category.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
SPDX-FileCopyrightText: 2025-2026 Univention GmbH
SPDX-License-Identifier: AGPL-3.0-only
-->

<umc version="2.0">
  <categories>
    <category id="category" priority="..." icon="....svg" color="
↪#xxxxxx"/>
  </categories>
</umc>
```

## 8.4 Local system module

The UMC server provides management services that are provided by so called UMC modules. These modules are implemented in Python (backend) and in JavaScript (web frontend). The following page provides information about developing and packaging of UMC modules. It is important to know the details of *Architecture* (page 103).

The package `univention-management-console-dev` provides the command `umc-create-module`, which can be used to create a template for a custom UMC module.

## 8.4.1 Python API

The Python API for the UMC is defined in the Python module `univention.management.console.base`.

## 8.4.2 UMC module API (Python and JavaScript)

A UMC module consists of three components

- A XML document containing the definition.
- The Python module defining the command functions.
- The JavaScript frontend providing the web frontend.

### XML definition

The UMC server knows three types of resources that define the functionality it can provide:

#### UMC modules

provide commands that can be executed if the required permission is given.

#### Syntax types

can be used to verify the correctness of command attributes defined by the UMC client in the request message or return values provided by the UMC modules.

#### Categories

help to define a structure and to sort the UMC modules by its type of functionality.

All these resources are defined in XML files. The details are described in the following sections

### Module definition

The UMC server does not load the Python modules to get the details about the modules name, description and functionality. Therefore, each UMC module must provide an XML file containing this kind of information. The following example defines a module with the id `udm`:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
SPDX-FileCopyrightText: 2025-2026 Univention GmbH
SPDX-License-Identifier: AGPL-3.0-only
-->

<umc version="2.0">
  <module id="udm" icon="udm/module" version="1.0">
    <name>Univention Directory Manager</name>
    <description>Manages all UDM modules</description>
    <flavor icon="udm-users" id="users/user">
      <name>Users</name>
      <description>Managing users</description>
    </flavor>
    <categories>
      <category name="domain"/>
    </categories>
    <command name="udm/query" function="query"/>
    <command name="udm/containers" function="containers"/>
  </module>
</umc>
```

The element `module` defines the basic details of a UMC module.

**id**

This identifier must be unique among the modules of an UMC server. Other files may extend the definition of a module by adding more flavors or categories.

**icon**

The value of this attribute defines an identifier for the icon that should be used for the module. Details for installing icons can be found in the *Packaging* (page 110).

The child elements `name` and `description` define the English human readable name and description of the module. For other translations the build tools will create translation files. Details can be found in the *Packaging* (page 110).

This example defines a so called *flavor*. A flavor defines a new name, description and icon for the same UMC module. This can be used to show several virtual modules in the overview of the web frontend. Additionally, the flavor is passed to the UMC server with each request i.e. the UMC module has the possibility to act differently for a specific flavor.

As the next element `categories` is defined in the example. The child elements `category` set the categories within the overview where the module should be shown. Each module can be part of multiple categories. The attribute `name` is the internal identifier of a category.

At the end of the definition file a list of commands is specified. The UMC server only passes commands to a UMC module that are defined. A command definition has two attributes:

**name**

is the name of the command that is passed to the UMC module. Within the HTTP request it is the URL path after `/univention/command/`.

**function**

defines the method to be invoked within the Python module when the command is called.

## Category definition

The predefined set of categories can be extended by each module.

Listing 8.4: UMC module category examples

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
SPDX-FileCopyrightText: 2025-2026 Univention GmbH
SPDX-License-Identifier: AGPL-3.0-only
-->

<umc version="2.0">
  <categories>
    <category id="favorites">
      <name>Favorites</name>
    </category>
    <category id="system">
      <name>System</name>
    </category>
    <category id="wizards">
      <name>Wizards</name>
    </category>
    <category id="monitor">
      <name>Surveillance</name>
    </category>
  </categories>
</umc>
```

## Python module

The Python API for UMC modules primarily consists of one base class that must be implemented. As an addition the Python API provides some helpers:

- Exception classes
- Translation support
- Logging functions
- UCR access

In the definition file, the UMC module specifies functions for the commands provided by the module. These functions must be implemented as methods of the class `Instance` that inherits from `univention.management.console.base.Base`.

The following Python code example matches the definition in the previous section:

```
# SPDX-FileCopyrightText: 2021-2026 Univention GmbH
#
# SPDX-License-Identifier: AGPL-3.0-only

from univention.management.console import Translation
from univention.management.console.base import Base, UMC_Error
from univention.management.console.config import ucr
from univention.management.console.log import MODULE
from univention.management.console.modules.decorators import sanitize
from univention.management.console.modules.sanitizers import IntegerSanitizer

_ = Translation('univention-management-console-modules-udm').translate

class Instance(Base):
    def init(self):
        """Initialize the module with some values"""
        super().init()
        self.data = [int(x) for x in ucr.get('some/example/ucr/variable', '1,2,3').
↳split(',')]

    def query(self, request):
        """get all values of self.data"""
        self.finished(request.id, self.data)

    @sanitize(item=IntegerSanitizer(required=True))
    def get(self, request):
        """get a specific item of self.data"""
        try:
            item = self.data[request.options['item']]
        except IndexError:
            MODULE.error('A invalid item was accessed.')
            raise UMC_Error(_('The item %d does not exists.') % (request.options[
↳'item'],), status=400)
        self.finished(request.id, self.data[item])

    @sanitize(IntegerSanitizer(required=True))
    def put(self, request):
        """replace all data with the list provided in request.options"""
        self.data = request.options
        self.finished(request.id, None)
```

Each command methods has one parameter that contains the UMC request. Such an object has the following properties:

**id**  
the unique identifier of the request.

**options**  
contains the arguments for the command. For most commands it is a dictionary.

**flavor**  
the name of the flavor that was used to invoke the command. This might be `None`.

The method `init()` in the example is invoked when the module process starts. It could for example be used to initialize a database connection.

The other methods in the example will serve specific requests. To respond to a request the function `finished()` must be invoked. To validate the request body the decorator `@sanitize` might be used with various sanitizers defined in `univention.management.console.modules.sanitizers`.

For a way to send an error message back to the client the `UMC_Error` can be raised with the error message as argument and an optional HTTP status code. The base class for modules provides some properties and methods that could be useful when writing UMC modules:

**username**  
The username of the owner of this session.

**user\_dn**  
The DN of the user or `None` if the user is only a local user.

**password**  
The password of the user.

**init()**  
Is invoked after the module process has been initialized. At that moment, the settings, like locale and username and password are available.

**destroy()**  
Is invoked before the module process shuts down.

## UMC store API

In order to encapsulate and ease the access to module data from the JavaScript side, a store object offers a unified way to query and modify module data.

The UMC JavaScript API comes with an object store implementation of the [Dojo store API](#)<sup>219</sup>. This allows the JavaScript code to access/modify module data and to observe changes on the data in order to react immediately. The following methods are supported:

**get(*id*)**  
Returns a dictionary of all properties for the object with the specified identifier.

**put(*dictionary*, *options*)**  
modifies an object with the corresponding properties and an optional dictionary of options.

**add(*dictionary*, *options*)**  
Adds a new object with the corresponding properties and an optional dictionary of options.

**remove(*id*)**  
Removes the object with the specified identifier.

**query(*dictionary*)**  
Queries a list of objects (returned as list of dictionaries) corresponding to the given query which is represented as dictionary. Note that not all object properties need to be returned in order to save bandwidth.

<sup>219</sup> <https://dojotoolkit.org/reference-guide/1.10/dojo/store.html>

The UMC object store class in JavaScript will be able to communicate directly with the Python module if the following methods are implemented:

**module/get ()**

Expects as input a list of unique IDs (as strings) and returns a list of dictionaries as result. Each dictionary entry holds all object properties.

**module/put ()**

Expects as input a list of dictionaries where each entry has the properties `object` and `options`. The property `object` holds all object properties to be set (i.e., this may also be a subset of all possible properties) as a dictionary. The second property `options` is an optional dictionary that holds additional options as a dictionary.

**module/add ()**

Expects similar input values as `module/put ()` (page 110).

**module/remove ()**

Expects as input a list of dictionaries where each entry has the properties `object` (containing the object's unique ID (as string)) and `options`. The `options` property can be necessary as a removal might be executed in different ways (recursively, shallow removal etc.).

**module/query ()**

Expects as input a dictionary with entries that specify the query parameters and returns a list of dictionaries. Each entry may hold only a subset of all possible object properties.

Further references:

- [Dojo object store reference guide](#)<sup>220</sup>
- [Object store tutorial](#)<sup>221</sup>
- [HTML5 IndexedDB object store API](#)<sup>222</sup>

### 8.4.3 Packaging

A UMC module consists of several files that must be installed at a specific location. As this mechanism is always the same there are `debhelper` tools making package creation for UMC modules very easy.

The following example is based on the package for the UMC module UCR.

A UMC module may be part of a source package with multiple binary packages. The examples uses a own source package for the module.

As a first step create a source package with the following directories and files:

- `univention-management-console-module-ucr/`
- `univention-management-console-module-ucr/debian/`
- `univention-management-console-module-ucr/debian/univention-management-console-module-ucr.umc-modules`
- `univention-management-console-module-ucr/debian/rules`
- `univention-management-console-module-ucr/debian/changelog`
- `univention-management-console-module-ucr/debian/control`
- `univention-management-console-module-ucr/debian/copyright`

All these files are standard Debian packaging files except `univention-management-console-module-ucr.umc-modules`. This file contains information about the locations of the UMC module source files:

---

<sup>220</sup> <https://dojotoolkit.org/reference-guide/1.10/dojo/store.html>

<sup>221</sup> <https://www.sitepen.com/blog/2011/02/15/dojo-object-stores/>

<sup>222</sup> <https://www.w3.org/TR/IndexedDB/#object-store>

```

Module: ucr
Python: umc/python
Definition: umc/ucr.xml
Syntax: umc/syntax/ucr.xml
Javascript: umc/js
Icons: umc/icons

```

The keys in this file of the following meaning:

**Module**

The internal name of the module

**Python**

A directory that contains the Python package for the UMC module

**Definition**

The filename of the XML file with the module definition

**Javascript**

A directory containing the JavaScript source code

**Icons**

A directory containing the icons required by the modules web frontend

**Syntax (optional)**

The filename of the XML file with the syntax definitions

**Category (optional)**

The filename of the XML file with the category definitions

The directory structure for such a UMC module file would look like this:

- univention-management-console-module-ucr/umc/
- univention-management-console-module-ucr/umc/syntax/
- univention-management-console-module-ucr/umc/syntax/ucr.xml
- univention-management-console-module-ucr/umc/js/
- univention-management-console-module-ucr/umc/js/ucr.js
- univention-management-console-module-ucr/umc/js/de.po
- univention-management-console-module-ucr/umc/de.po
- univention-management-console-module-ucr/umc/icons/
- univention-management-console-module-ucr/umc/icons/16x16/
- univention-management-console-module-ucr/umc/icons/16x16/ucr.png
- univention-management-console-module-ucr/umc/icons/24x24/
- univention-management-console-module-ucr/umc/icons/24x24/ucr.png
- univention-management-console-module-ucr/umc/icons/64x64/
- univention-management-console-module-ucr/umc/icons/64x64/ucr.png
- univention-management-console-module-ucr/umc/icons/32x32/
- univention-management-console-module-ucr/umc/icons/32x32/ucr.png
- univention-management-console-module-ucr/umc/ucr.xml
- univention-management-console-module-ucr/umc/python/
- univention-management-console-module-ucr/umc/python/ucr/
- univention-management-console-module-ucr/umc/python/ucr/de.po

- univention-management-console-module-ucr/umc/python/ucr/\_\_init\_\_.py

If such a package has been created a few things need to be adjusted

### debian/rules

```
#!/usr/bin/make -f
%:
    dh $@ --with umc,python3
```

### debian/control

```
Source: univention-management-console-module-ucr
Section: univention
Priority: optional
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends:
    debhelper-compat (= 13),
    dh-python,
    python3-all,
    univention-management-console-dev (>= 12.0.2),
Standards-Version: 4.3.0.3

Package: univention-management-console-module-ucr
Architecture: all
Depends:
    univention-management-console-server,
Provides:
    ${python3:Provides},
Description: UMC module for UCR
    This package contains the UMC module for Univention Configuration Registry
```

## 8.5 Domain LDAP module

Done through flavor.

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
SPDX-FileCopyrightText: 2025-2026 Univention GmbH
SPDX-License-Identifier: AGPL-3.0-only
-->

<umc version="2.0">
    <module id="udm" icon="udm-MODULE" version="1.0">
        <flavor priority="25" icon="udm-MODULE-SUBMODULE" id="MODULE/
→SUBMODULE">
            <name>MODULE name</name>
            <description>MODULE description</description>
        </flavor>
        <categories>
            <category name="domain"/>
        </categories>
    </module>
</umc>
```

Must use /umc/module/category/@name="domain"!

Must use `/umc/module/@translationId` to specify alternative translation file, which must be installed as `/usr/share/univention-management-console/i18n/language/module.mo`.

## 8.6 Disabling a module

To disable a module, use the following XML file as a template:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
SPDX-FileCopyrightText: 2025-2026 Univention GmbH
SPDX-License-Identifier: AGPL-3.0-only
-->

<umc version="2.0">
  <module id="udm" icon="udm/module" version="1.0">
    <name/>
    <description/>
    <flavor id="MODULE/SUBMODULE" deactivated="yes" />
  </module>
</umc>
```



## WEB SERVICES

### 9.1 Extending the overview page

When users open `http://localhost/` or `http://hostname/` in a browser, they are redirected to the UCS overview page.

Depending on the preferred language negotiated by the web browser the user is either redirected to the German or English version. The overview page is split between *Installed web services* and *Administration* entries.

The start page can be extended using Univention Configuration Registry variables. *PACKAGE* refers to a unique identifier, typically the name of the package shipping the extensions to the overview page. The configurable options are explained below:

- `ucs/web/overview/entries/admin/PACKAGE/OPTION` variables extend the administrative section.
- `ucs/web/overview/entries/service/PACKAGE/OPTION` variables extend the web services section.

To configure an extension of the overview page the following options must/can be set using the pattern `ucs/web/overview/entries/admin/PACKAGE/OPTION=*VALUE*` (and likewise for services).

**link**

defines a link to a URL representing the service (usually a web interface).

**label**

specifies a title for an overview entry. The title can also be translated; for example `label/de` can be used for a title in German.

**description**

configures a longer description of an overview entry. The description can also be translated; for example `description/de` can be used for a description in German. Should not exceed 60 characters, because of space limitations of the rendered box.

**icon**

Optionally an icon can be displayed. Using `icon`, either a filename or a URI can be provided. When specifying a filename, the name must be relative to the directory `/var/www`, that is with a leading `/`. All file formats typically displayed by browsers can be used (for example PNG/JPG). All icons must be scaled to `50x50` pixels.

**priority**

The display order can be specified using `priority`. Depending on the values the entries are displayed in *lexicographical* order (i.e. `100 < 50`).



## APP CENTER

The Univention App Center provides a platform for software vendors and an easy-to-use entry point for Univention Corporate Server users to extend their environment with business software.

The documentation how to develop Apps for Univention App Center can be found in the [Univention App Center for App Providers<sup>223</sup>](#) guide.

---

<sup>223</sup> <https://docs.software-univention.de/app-center/5.2/en/index.html>



## INTEGRATION OF EXTERNAL REPOSITORIES

Sometimes it might be necessary to add external repositories, for example when testing an application which is developed for UCS@school. Such components can be registered through Univention Management Console or in Univention Configuration Registry.

Components can be versioned. This ensures that only components are installed that are compatible with a UCS version.

### **empty or unset or current**

The current major-minor version will be used.

If for example UCS 5.2 is currently in use, only the 5.2 repository will be used. Please note that all major and minor updates will be blocked until the component is available for the new release. Patch level and errata updates are not affected.

If for example UCS 5.1 is currently installed. When UCS 5.2 or UCS 6.0 become available, the release updated will be postponed until the component is also available for version 5.2 and 6.0 respectively.

### **major.minor**

By specifying an explicit version number only the specified version of the component will be used if it matches the current UCS version. Release updates of the system will not be hindered by such components. Multiple versions can be given using comma as delimiter.

For example `5.1 5.2` would only include the component with UCS 5.1 and 5.2 but not if UCS 5.0 or UCS 5.3 is in use.

## 11.1 Integrate with Univention Management Console

A list of the integrated repository components is in the UMC module *Repository Settings*. Applications which have been added through the Univention App Center are still listed here, but should be managed through the *App Center* module.

A further component can be set up with *Add*. The *Component name* identifies the component on the repository server. A free text can be entered under *Description*, for example, for describing the functions of the component in more detail.

The absolute URL of the download server is to be entered in the input field *Repository server*, and can also optionally contain a *Username*, *Password*, *Repository prefix* (file path) and *port* if required.

### **Warning**

The credentials are stored unencrypted and as plain text in Univention Configuration Registry. Every user with access to the local system can read them.

A software component is only available when *Enable this component* has been activated.

Prior to UCS 5 two separate repository branches were provided for *maintained* and *unmaintained* software. While UCS 5 no longer uses this distinction.

## 11.2 Integrate with Univention Configuration Registry

You can use the following Univention Configuration Registry Variables to register a repository component. It's also possible to activate further functions here that you can't configured through the UMC module. *NAME* stands for the component's name.

### **repository/online/component/NAME/server**

The repository server absolute URL on which the components are available. If this variable isn't set, UCS uses the server from Univention Configuration Registry Variable `repository/online/server`<sup>224</sup>.

### **repository/online/component/NAME**

You must set this variable to `enabled`, if UCS should activate and use the component.

### **repository/online/component/NAME/localmirror**

You can use this variable to configure whether UCS mirrors the component locally. In combination with the Univention Configuration Registry Variable `repository/online/component/NAME/server` (page 120), you can set up a configuration so that UCS mirrors the component, but doesn't activate it, or that UCS activates the component, but doesn't mirror it.

### **repository/online/component/NAME/description**

A optional description for the repository.

### **repository/online/component/NAME/prefix**

Deprecated since version 5.0.

Defines the URL path prefix that the repository server uses. Don't use this variable anymore. Instead, specify the path as part of the absolute URL in the UCR variable `repository/online/component/NAME/server` (page 120).

For example: `repository/online/component/NAME/server=https://repository.example.com/prefix`

### **repository/online/component/NAME/layout**

Defines the type of the repository:

- If the variable has the value `arch` or is unset, UCS searches for the `Packages` within the architecture subdirectories `amd64/` and `all/` respectively.
- If the variable has the value `flat`, UCS searches for the `Packages` file within the root directory of the repository.

This variable is usually unset.

### **repository/online/component/NAME/username**

Deprecated since version 5.0.

The variable defines the username if the repository server requires authentication. Don't use this variable anymore. Instead, specify the username as part of the absolute URL in the UCR variable `repository/online/component/NAME/server` (page 120).

For example: `repository/online/component/NAME/server=https://username@repository.example.com`

### **repository/online/component/NAME/password**

Deprecated since version 5.0.

This variable defines the password if the repository server requires authentication. Don't use this variable anymore. Instead, specify the password as part of the absolute URL in the UCR variable `repository/online/component/NAME/server` (page 120).

For example: `repository/online/component/NAME/server=https://username:password@repository.example.com`

---

<sup>224</sup> <https://docs.software-univention.de/manual/5.2/en/appendix/variables.html#envvar-repository-online-server>

**repository/online/component/NAME/version**

This variable controls the versions to include. For more information, see *Integration of external repositories* (page 119).

**repository/online/component/NAME/defaultpackages**

Defines a list of package names separated by blanks. The UMC module *Repository Settings* offers the installation of this component if at least one of the packages isn't installed. Specifying the package list eases the subsequent installation of components.



## TRANSLATE UCS

This chapter is for those who want to translate UCS into another language.

- To add or update the translation for a package, refer to *Translating a single Debian package* (page 123).
- To add a translation package for UCS, refer to *Create a translation package for UCS* (page 126).

A guide to editing translation files is provided in *Editing translation files* (page 128). The previously mentioned sections refer to it when necessary.

### 12.1 Translating a single Debian package

When creating a new package or updating an existing one, it is possible to provide a translation for that package by following the workflow described in this section. Examples in this section use the German translation, but they are applicable to any other language as well.

#### 12.1.1 Setup of `univention-110n-build`

The setup depends on the operating system developers use to develop the package. A running UCS installation is recommended, where translators can set up the tools with `univention-install`, see *Setup on a UCS machine* (page 123). Otherwise, follow the instructions in section *Setup on a non-UCS machine* (page 123). Both setup variants provide the command `univention-110n-build`.

##### Setup on a UCS machine

Install the package `univention-110n-dev` as root:

```
$ univention-install univention-110n-dev
```

After the installation of `univention-110n-dev`, the command `univention-110n-build` is available for the following steps.

Skip the next section and continue with *UCS package translation workflow* (page 124).

##### Setup on a non-UCS machine

First, install Git<sup>225</sup>, Python 3.7 or later<sup>226</sup> and pip<sup>227</sup>. For example, run the following command on Ubuntu 20.04:

```
$ sudo apt-get install git python3 python3-pip
```

To checkout the latest version of the UCS Git repository, if not yet available, use the following commands:

---

<sup>225</sup> <https://git-scm.com/downloads>

<sup>226</sup> <https://www.python.org/downloads/>

<sup>227</sup> <https://pip.pypa.io/en/stable/installation/>

```
$ mkdir ~/translation
$ cd ~/translation
$ git clone \
  --single-branch --depth 1 --shallow-submodules \
  https://github.com/univention/univention-corporate-server
```

Install the python package `univention-110n` with pip:

```
$ pip install ~/translation/univention-corporate-server/packaging/univention-110n/
```

Pip installs all required Python packages and the command `univention-110n-build`.

## 12.1.2 UCS package translation workflow

The translation process is divided into the following steps:

1. *Prepare the source code* (page 124) for translation.
2. *Add and/or update supplementary files* (page 124).
3. *Run univention-110n-build* (page 126) for the package.
4. *Translate* (page 126) the strings by editing the `.po` files.

The `.po` files used in this section contain the German language code `de` in the file `de.po`. Use the appropriate language code from the [ISO-639-1 list](https://www.iso.org/standard/50017.html)<sup>228</sup> for other languages.

### Prepare the source code

Mark all strings that need translation within the source code. See the following example for a Python file:

```
from univention.lib.i18n import Translation
_ = Translation("<packagename>").translate
example_string = _("Hello World!")
```

Replace `<packagename>` with the wanted *gettext* domain, for example the name of the UCS Debian package like the existing package `univention-management-console-module-udm`.

For UMC XML files, the translatable XML elements are automatically added to their associated `de.po` file. This includes XML elements like `name`, `description`, `keywords`, and more.

For UMC JavaScript module files, include the translation function `_` in the define function:

```
define([
  "umc/i18n!umc/modules/<module>"
], function(_) {
  var example_string = _("Hello World");
});
```

Replace `<module>` with the module id (examples for existing packages: `appcenter`, `udm`).

### Add and/or update supplementary files

The program `univention-110n-build` needs to know which source files target which `de.po` file. `de.po` files associate translatable strings with their translations and are meant to be edited manually. For more information, see the [gettext](https://www.gnu.org/software/gettext/)<sup>229</sup> framework upon which `univention-110n-build` is based.

For a UMC package, `de.po` files are automatically created for its associated XML file, the JavaScript files and the Python module, see [debian/package.umc-modules](#) (page 104) about UMC modules.

<sup>228</sup> [https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_639-1\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)

<sup>229</sup> <https://www.gnu.org/software/gettext/>

Other source files have to be declared with `.univention-110n` files that are located in the `debian` directory and structured like the following example from the package `univention-appcenter`:

```
[
  {
    "input_files": [
      "udm/*.*"
    ],
    "po_subdir": "udm/handlers/appcenter",
    "target_type": "mo",
    "destination": "usr/share/locale/{lang}/LC_MESSAGES/univention-
↪admin-handlers-appcenter.mo"
  }
]
```

This file instructs `univention-110n` to compile a `de.po` file in the directory `udm/handlers/appcenter` which includes translations for all files below the directory `udm`. The name `univention-admin-handlers-appcenter` has to be replaced with the wanted *gettext* domain, for example the name of the new or updated Debian package. Additionally, if there are one or more `.univention-110n` files, add `univention-110n` to the add-on list in the `debian/rules` file:

```
$ dh --with univention-110n
```

As an example, refer to the following file tree of the `appcenter` package, which displays all relevant files for the translation inside the package:

```
├─ debian
│   └─ rules
│       └─ univention-management-console-module-appcenter.umc-modules
│       └─ univention-management-console-module-appcenter.univention-110n
│       └─ ...
├─ ...
├─ udm
│   └─ handlers
│       └─ appcenter
│           └─ de.po
│           └─ ...
├─ umc
│   └─ appcenter.xml
│   └─ de.po
│   └─ ...
│   └─ js
│       └─ de.po
│       └─ appcenter.js
│       └─ ...
│   └─ python
│       └─ appcenter
│           └─ de.po
│           └─ __init__.py
│           └─ ...
```

#### **debian/rules**

Add `univention-110n` add-on if non-UMC files have to be translated.

#### **debian/univention-management-console-module-appcenter.umc-modules**

See *debian/package.umc-modules* (page 104).

#### **debian/univention-management-console-module-appcenter.univention-110**

Instructions for translatable non-UMC files.

### `udm/handlers/appcenter/de.po`

Only created/updated if defined in `univention-management-console-module-appcenter.univention-l10n`.

### `umc/appcenter.xml`

UMC standard XML file.

### `umc/de.po`

UMC standard `de.po` file for `appcenter.xml`.

### `umc/js/de.po`

UMC standard `de.po` file for all JavaScript files.

### `umc/js/appcenter.js`

One of the JavaScript files with translatable strings.

### `umc/python/appcenter/de.po`

UMC standard `de.po` file for all Python files.

### `umc/python/appcenter/__init__.py`

One of the Python files with translatable strings.

## Run `univention-l10n-build`

Run the command `univention-l10n-build` in the package directory. The program finds all marked strings and either updates or creates the corresponding `de.po` file.

### Warning

`univention-l10n-build` updates every package in the current working directory and below. Make sure to run the program from inside the package directory, if this is not the desired outcome.

## Translate

After `univention-l10n-build` finished, the translation can start. Edit the `de.po` files with a text editor. Find all empty `msgstr` fields and enter the translation of the corresponding `msgid`. See [Editing translation files](#) (page 128) for details.

After the translation step, build and test the package on a UCS installation. Repeat this workflow every time a marked string is changed or a new one is added to the source files.

## 12.2 Create a translation package for UCS

UCS provides builtin English and German localization and a French translation package. Univention provides a set of tools that facilitates the creation of translation packages. Translation packages can provide translations for all translatable strings of UCS for a specific language. The Univention Management Console, more specifically its packages, contains the largest share of translatable strings. This section describes all necessary steps to create a translation package for UCS.

### 12.2.1 Install needed tools

The package `univention-l10n-dev` contains all tools required to set up and update a translation package. It requires some additional Debian tools to build the package. Run the following command on UCS to install all needed packages.

```
$ sudo univention-install univention-l10n-dev dpkg-dev git
```

## 12.2.2 Obtain a current checkout of the UCS Git repository

The Git repository is later processed to get initial files for a new translation (often referred to as PO file or Portable Objects).

```
$ mkdir ~/translation
$ cd ~/translation
$ git clone \
  --single-branch \
  --depth 1 \
  --shallow-submodules \
  https://github.com/univention/univention-corporate-server
```

## 12.2.3 Create translation package

To create a translation package, for example for French, in the current working directory, the following command must be executed:

```
$ cd ~/translation
$ univention-ucs-translation-build-package \
  --source ~/translation/univention-corporate-server \
  --languagecode fr \
  --locale fr_FR.UTF-8:UTF-8 \
  --language-name French
```

This creates a new directory `~/translation/univention-l10n-fr/` which contains a Debian source package of the same name. It includes all source and target files for the translation.

## 12.2.4 Edit translation files

The translation source files (`.po` files) are located below the directory `~/translation/univention-l10n-fr/fr`. Each file should be edited to create the translation. Refer to *Editing translation files* (page 128) for detailed information.

## 12.2.5 Update the translation package

First update the Git repository:

```
$ cd ~/translation/univention-corporate-server
$ git pull --rebase
```

If changes affecting translations are made in the Git repository, existing translation packages need to be updated to reflect those changes. Given a path to an updated Git checkout, `univention-ucs-translation-merge` can update a previously created translation source package.

The following example updates the translation package `univention-l10n-fr`:

```
$ univention-ucs-translation-merge \
  ~/translation/univention-corporate-server \
  ~/translation/univention-l10n-fr
```

## 12.2.6 Build the translation package

Before using the new translation, the Debian package has to be built and installed. This can be done with the following commands:

```
$ cd ~/translation/univention-l10n-fr
$ sudo apt-get build-dep .
```

(continues on next page)

(continued from previous page)

```
$ dpkg-buildpackage -uc -us -b -rfakeroot
$ sudo dpkg -i ../univention-l10n-fr_*.deb
```

After logging out of the Univention Management Console the new language should now be selectable in the Univention Management Console login window. Untranslated strings are still shown in their original language, that is, in English.

## 12.3 Editing translation files

The actual translation process is done by editing translation files which are named `<lang>.po`, where `<lang>` is an ISO-639-1<sup>230</sup> language code. For example, the German code is `de`, which results in `de.po` filenames. When following the instructions in the preceding sections, these files are generated by the package `gettext` behind the scenes. The manual can be found in GNU `gettext utilities`<sup>231</sup>.

### 12.3.1 Editing translation entries

In the following listing shows a simple example of a translation file:

```
#: umc/app.js:637
#, python-format
msgid "The %s will expire in %d days and should be renewed!"
msgstr ""
```

- The first line provides a hint, where the text is used.
- The second line is optional and contains flags, which indicate the type and state of the translation.
- The line starting with `msgid` contains the original text. The translation has to be placed on the line containing `msgstr`.

For more information about the PO file format, for example about the flags, see [The Format of PO Files](#)<sup>232</sup>.

Long texts can be split over multiple lines, where each line must start and end with a double-quote. The following example from the German translation shows a text spanning two lines, with the placeholder present in the original and translated text:

```
#: umc/js/appcenter/AppCenterPage.js:1067
#, python-format
msgid ""
"If everything else went correct and this is just a temporary network "
"problem, you should execute %s as root on that backup system."
msgstr ""
"Wenn keine weiteren Fehler auftraten und dies nur ein temporäres "
"Netzwerkproblem ist, sollten Sie %s als root auf dem Backup System ausführen."
```

Some lines contain parameters, in this example `%s` and `%d`. They are indicated by a flag like `c-format` or `python-format`, which must not be removed. The placeholders have to be carried over to the translated string unmodified and in the same order. Some other files contain placeholders of the form `%(text)s`, which are more flexible and can be reordered. Because of that, programmers should always use the form `%(foo)s`.

After a file has been translated completely, the line containing `fuzzy` at the beginning of the entry must be removed to avoid warnings. If a translation string consists of multiple lines the translated string should roughly contain as many lines as the original string.

When a `msgid` has changed and a translation existed beforehand, it is marked with `#, fuzzy`, as those have to be corrected:

<sup>230</sup> [https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_639-1\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)

<sup>231</sup> <https://www.gnu.org/software/gettext/manual/gettext.html>

<sup>232</sup> [https://www.gnu.org/software/gettext/manual/html\\_node/PO-Files.html](https://www.gnu.org/software/gettext/manual/html_node/PO-Files.html)

```
#: umc/js/appcenter/example.js:42
#, fuzzy
msgid "Hello!"
msgstr "Hallo, Welt!"
```

Correct the translation in the `msgstr` line and remove the line which contains `fuzzy`:

```
#: umc/js/appcenter/example.js:42
msgid "Hello!"
msgstr "Hallo!"
```

### Warning

If a `fuzzy` entry is still in one of the `de.po` files, the package build process fails.

## 12.3.2 Update meta information

The first entry of a `.po` file contains its meta information, with each line consisting of a name-value pair. If the translation work within a file is done, update this information. As an example, see the following excerpt from a `.po` translation file:

```
msgid ""
msgstr ""
"Project-Id-Version: univention-management-console-module-services\n"
"Report-Msgid-Bugs-To: packages@univention.de\n"
"POT-Creation-Date: 2020-09-25 01:15+0200\n"
"PO-Revision-Date: 2020-09-25 11:26+0100\n"
"Last-Translator: Univention GmbH <packages@univention.de>\n"
"Language-Team: Univention GmbH <packages@univention.de>\n"
"Language: de\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

Running `univention-110n-build` updates the `POT-Creation-Date`. The `PO-Revision-Date` should be updated every time the `.po` has been modified. Insert the [ISO 639 language code](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)<sup>233</sup> for the target translation language into `Language`. Enter contact information into `Last-Translator`, `Language-Team` and `Report-Msgid-Bugs-To`.

See the `gettext` manual entry about [header entries](https://www.gnu.org/software/gettext/manual/html_node/Header-Entry.html#Header-Entry)<sup>234</sup> for more information about all fields, including optional fields not listed here. Tools like `poedit`<sup>235</sup> update some of the fields automatically for the user.

<sup>233</sup> [https://en.wikipedia.org/wiki/List\\_of\\_ISO\\_639-1\\_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)

<sup>234</sup> [https://www.gnu.org/software/gettext/manual/html\\_node/Header-Entry.html#Header-Entry](https://www.gnu.org/software/gettext/manual/html_node/Header-Entry.html#Header-Entry)

<sup>235</sup> <https://poedit.net/>



## UNIVENTION UPDATER

The Univention Updater is used for updating the software. It is based on the Debian APT tools. On top of that the updater provides some UCS specific additions.

### 13.1 Separate repositories

UCS releases are provided either through ISO images or through online repositories. For each major, minor and patchlevel release there is a separate online repository. They are automatically added to the files in `/etc/apt/sources.list.d/` depending on the Univention Configuration Registry Variables `version/version` and `version/patchlevel`, which are managed by the updater.

Separate repositories are used to prevent automatic updates of software packages. This is done to encouraged users to thoroughly test a new release before their systems are updated. The only exception from this rule are the errata updates, which are put into a single repository, which is updated incrementally.

Therefore, the updater will include the repositories of a new release in a file called `/etc/apt/sources.list.d/00_ucs_temporary_installation.list` and then do the updates. Only at the end of a successful update are the Univention Configuration Registry Variables updated.

Additional components can be added as separate repositories using Univention Configuration Registry Variables `repository/online/component/...`, which are described in *Integration of external repositories* (page 119) and manual. Setting the variable `.../version` can be used to mark a component as required (for certain UCS versions), which blocks an upgrade until the component is available for the specific release(es).

If configured and enabled, components are considered required if the variable `.../version` is unset or set to `current`.

As an alternative a fixed list of `major.minor` releases can be used to include the component only for a sub-set of releases: such a component is only used locally if the listed component versions include the current version, for example a `5.0 5.1 5.2` component will not be used on a `5.4` system.

### 13.2 Updater scripts

In addition to the regular Debian Maintainer Scripts (see *debian/preinst*, *debian/prerm*, *debian/postinst*, *debian/postrm* (page 148)) the UCS updater supports additional scripts, which are called before and after each release update. Each UCS release and each component can include its own set of scripts.

#### **preup.sh**

These scripts are called before the update is started. If any of the scripts aborts with an exit value unequal zero, the update is canceled and never started. The scripts receives the version number of the next release as an command line argument.

For components their `preup.sh` scripts is called twice:

- Before the main release `preup.sh` script is called
- After the main script was called.

This is indicated by the additional command line argument `pre` respectively `post`, which is *inserted before* the version string.

### **postup.sh**

These scripts are called after the update successfully completed. If any of the scripts aborts with an exit value unequal zero, the update is canceled and does not finish successfully. The scripts receives the same arguments as described above.

The scripts are located in the `all/` component of each release and component. For UCS-5.0 this would be `dists/ucs500/preup.sh` and `5.0/maintained/components/some-component/all/preup.sh` for the `preup.sh` script. The same applies to the `postup.sh` script. The full process is shown in [Release update walk-through](#) (page 132).

### **13.2.1 Digital signature**

From UCS 3.2 on the scripts must be digitally signed by an PGP (Pretty Good Privacy) key stored in the key-ring of `apt-key`. The detached signature must be placed in a separate file next to each updater scripts with the additional filename extension `.gpg`, that is `preup.sh.gpg` and `postup.sh.gpg`. These extra files are downloaded as well and any error in doing so and in the validation process aborts the updater immediately.

The signatures must be updated after each change to the underlying scripts. This can be automated or be done manually with a command like the following: `gpg -a -u key-id --passphrase-file key-phrase-file -o script.sh.gpg -b script.sh`

Signatures can be checked manually using the following command: `apt-key verify script.sh.gpg script.sh`

## **13.3 Release update walk-through**

For an release update, the following steps are performed. It assumes a single component is enabled. If multiple components are enabled, the order in which their scripts are called is unspecified. It shows which scripts are called in which order with which arguments.

1. Create temporary source list file `00_ucs_temporary_installation.list`
2. Download the `preup.sh` and `postup.sh` files for the next release and all components into a temporary directory and validate their PGP signatures
3. Execute `component-preup.sh pre $version`
4. Execute `release-preup.sh $version`
5. Execute `component-preup.sh post $version`
6. Download the new `Packages` and `Release` files. Their PGP signatures validated by **APT** internally.
7. Perform the update
8. Set the release related Univention Configuration Registry Variables to the new version
9. Execute `component-postup.sh pre $version`
10. Execute `release-postup.sh $version`
11. Execute `component-postup.sh post $version`

## MISCELLANEOUS

### 14.1 Databases

UCS ships with two major database management systems, which are used for UCS internal purposes, but can also be used for custom additions.

#### 14.1.1 PostgreSQL

UCS uses PostgreSQL by default for its package tracking database, which collects the state and versions of packages installed on all systems of the domain.

#### 14.1.2 MariaDB

By default the MariaDB root password is set to \_\_\_\_\_. Debian provides the `dbconfig` package, which can be used to create and modify additional databases from maintainer scripts.

### 14.2 UCS lint

Use `ucslint` to find packaging issues.

For each issue one or more lines are printed. The first line per issue always contains several fields separated by `::`:

```
severity:module-id-test-id[:filename[:line-number[:column-number]]]:message
```

For some issues extra context data is printed on the following lines, which are indented with space characters. All other lines start with a letter specifying the severity:

- E** Error: Missing data, conflicting information, real bugs.
- W** Warning: Possible bug, but might be okay in some situations.
- I** Informational: found some issue, which needs further investigation.
- S** Style: There might be some better less error prone way.

The severities are ordered by importance. By default `ucslint` only aborts on errors, but this can be overwritten using the `--exitcode-categories` argument followed by a subset of the characters `EWIS`.

After the severity an identifier follows, which uniquely identifies the module and the test. The module is given as four digits, which is followed by a dash and the number of the test in that module. Currently the following modules exist:

**0001-CheckJoinScript**

Checks join file issues

**0002-CopyPasteErrors**

Checks for copy & paste error from example files

- 0004-CheckUCR**  
Checks UCR info files
- 0006-CheckPostinst**  
Checks Debian maintainer scripts
- 0007-Changelog**  
Checks `debian/changelog` file for conformance with Univention rules
- 0008-Translations**  
Checks translation files for completeness and errors
- 0009-Python**  
Checks Python files for common errors
- 0010-Copyright**  
Checks for Univention copyright
- 0011-Control**  
Checks `debian/control` file for errors
- 0013-bashism**  
Checks files using `/bin/sh` for BASH constructs
- 0014-Depends**  
Checks files for missing runtime dependencies on UCS packages
- 0015-FuzzyNames**  
Checks for spelling of Univention
- 0016-Deprecated**  
Checks files for usage of deprecated functions
- 0017-Shell**  
Checks shell scripts for quoting errors
- 0018-Debian**  
Checks for Debian packaging issues

The module and test number may be optionally followed by a filename, line number in that file, and column number in that line, where the issue was found. After that a message is printed, which describes the issue in more detail.

Since `ucslint` is very Univention centric, many of its tests return false positives for software packages by other parties. Therefore, many tests need to be disabled. For this the file `debian/ucslint.overrides` can be created with list of modules and test to be ignored. Without specifying the optional filename, line number and column number, the test is globally disabled for all files.

## 14.3 Function libraries

The source package `univention-lib` provides the binary packages `shell-univention-lib`, `python3-univention-lib` and `python-univention-lib`, which contain common library functions usable in shell or Python programs.

### 14.3.1 `shell-univention-lib`

This package (and several others) provides shell libraries in `/usr/share/univention-lib/`, which can be used in shell scripts.

`/usr/share/univention-lib/admember.sh`

This file contains some helpers to test for and to manage hosts in AD member mode.

`/usr/share/univention-lib/backup.sh`

This file contains code to remove old backup files from `/var/univention-backup/`.

**/usr/share/univention-lib/base.sh**

This file contains some helpers to create log files, handle unjoin scripts (see *Writing unjoin scripts* (page 37)) or query the network configuration.

**/usr/share/univention-lib/join.sh**

This file is provided by the package **univention-join**. It is used by Debian maintainer scripts to register and call join scripts. See *join.sh* (page 34) for further details.

**/usr/share/univention-lib/ldap.sh**

This file contains some helpers to query data from LDAP, register and un-register service entries, LDAP schema and LDAP ACL extensions.

**/usr/share/univention-lib/samba.sh**

This file contains a helper to check if Samba4 is used.

**/usr/share/univention-lib/ucr.sh**

This file is provided by the package **univention-config**. It contains some helpers to handle boolean Univention Configuration Registry Variables and handle UCR files on package removal. See *Using UCR from shell* (page 11) for further details.

**/usr/share/univention-lib/umc.sh**

This file contains some helpers to handle UMC (see *Univention Management Console (UMC)* (page 103)) related tasks.

**/usr/share/univention-lib/all.sh**

This is a convenient library, which just includes all libraries mentioned above.

### 14.3.2 python-univention-lib

This package provides several Python libraries located in the module **univention.lib**.

**univention.lib.admember**

This module contains functions to test for and to manage hosts in AD member mode.

**univention.lib.atjobs**

This module contains functions to handle **at**-jobs.

**univention.lib.fstab**

This module provides some functions for handling the file `/etc/fstab`.

**univention.lib.i18n**

This module provides some classes to handle texts and their translations.

**univention.lib.ldap\_extension**

This module provides some helper functions internally used to register LDAP extension as described in *join.sh* (page 34).

**univention.lib.listenerSharePath**

This module provides some helper functions internally used by the Directory Listener module handling file shares.

**univention.lib.locking**

This module provides some functions to implement mutual exclusion using file objects as locking objects.

**univention.lib.misc**

This module provides miscellaneous functions to query the set of configured LDAP servers, localized domain user names, and other functions.

**univention.lib.package\_manager**

This module provides some wrappers for **dpkg** and **APT**, which add functions for progress reporting.

**univention.lib.s4**

This module provides some well known SIDs and RIDs.

### `univention.lib.ucrLogrotate`

This module provides some helper functions internally used for parsing the Univention Configuration Registry Variables related to `logrotate.8`<sup>236</sup>.

### `univention.lib.ucs`

This module provides the class `UCS_Version` to more easily handle UCS version strings.

### `univention.lib.umc`

This module provides the class `Client` to handle connections to remote UMC servers.

### `univention.lib.umc_module`

This module provides some functions for handling icons.

## 14.4 Login access control

Access control to services can be configured for individual services by setting certain Univention Configuration Registry Variables. Setting `auth/SERVICE/restrict` to `true` enables access control for that service. This will include the file `/etc/security/access-SERVICE.conf`, which contains the list of allowed users and groups permitted to login to the service. Users and groups can be added to that file by setting `auth/SERVICE/user/USER` and `auth/SERVICE/group/GROUP` to `true` respectively.

## 14.5 Network packet filter

Firewall rules are setup by `univention-firewall` and can be configured through Univention Configuration Registry or by providing additional UCR templates.

### 14.5.1 Filter rules by Univention Configuration Registry

Besides predefined service definitions, Univention Firewall also allows the implementation of package filter rules through Univention Configuration Registry. These rules are included in `/etc/security/packetfilter.d/` through a Univention Configuration Registry module.

Filter rules can be provided through packages or can be configured locally by the administrator. Local rules have a higher priority and overwrite rules provided by packages.

All Univention Configuration Registry settings for filter rules are entered in the following format:

#### Local filter rule

```
security/packetfilter/protocol/>port (s) address=policy
```

#### Package filter rule

```
security/packetfilter/package/package/protocol/port (s) /address=policy
```

The following values need to be filled in:

#### **package (only for packaged rules)**

The name of the package providing the rule.

#### **protocol**

Can be either `tcp` for server services using the *Transmission Control Protocol* or `udp` for services using the stateless *User Datagram Protocol*.

#### **port; min-port: max-port**

Ports can be defined either as a single number between 1 and 65535 or as a range separated by a colon:  
`min-port: max-port`

#### **address**

This can be either `ipv4` for all IPv4 addresses, `ipv6` for all IPv6 addresses, `all` for both IPv4 and IPv6 addresses, or any explicitly specified IPv4 or IPv6 address.

---

<sup>236</sup> <https://manpages.debian.org/bookworm/logrotate/logrotate.8.en.html>

**policy**

If a rule is registered as `DROP`, then packets to this port will be silently discarded; `REJECT` can be used to send back an ICMP message `port unreachable` instead. Using `ACCEPT` explicitly allows such packets. (IPTables rules are executed until one rule applies; thus, if a package is accepted by a rule which is discarded by a later rule, then the rule for discarding the package does not become valid).

Filter rules can optionally be described by setting additional Univention Configuration Registry Variables. For each rule and language, an additional variable suffixed by `/language` can be used to add a descriptive text.

Some examples:

Listing 14.1: Local firewall rules

```
security/packetfilter/tcp/2000/all=DROP
security/packetfilter/tcp/2000/all/en=Drop all packets to TCP port 2000
security/packetfilter/udp/500:600/all=ACCEPT
security/packetfilter/udp/500:600/all/en=Accept UDP port 500 to 600
```

All package rules can be globally disabled by setting the Univention Configuration Registry Variable `security/packetfilter/use_packages` to `false`.

### 14.5.2 Local filter rules through iptables commands

Besides the existing possibilities for settings through Univention Configuration Registry, there is also the possibility of integrating user-defined enhanced configurations in `/etc/security/packetfilter.d/`, for example for realizing a firewall or Network Address Translation. The enhancements should be realized in the form of shell scripts which execute the corresponding `iptables` for IPv4 and `ip6table` for IPv6 calls. For packages this is best done through using a Univention Configuration Registry template as described in *File* (page 15).

Full documentation for IPTables can be found at the [netfilter/iptables project](https://netfilter.org/projects/iptables/)<sup>237</sup>.

### 14.5.3 Testing Univention Firewall settings

Package filter settings should always be thoroughly tested. The network scanner `nmap`, which is integrated in Univention Corporate Server as a standard feature, can be used for testing the status of individual ports.

Since `nmap` requires elevated privileges in the network stack, it should be started as `root` user. A TCP port can be tested with the following command: `nmap HOSTNAME -p PORT(s)`

A UDP port can be tested with the following command: `nmap HOSTNAME -sU -p PORT(s)`

Listing 14.2: Using nmap for firewall port testing

```
$ nmap 192.0.2.100 -p 400
$ nmap 192.0.2.110 -sU -p 400-500
```

## 14.6 Active Directory Connection custom mappings

For general overview about the **Active Directory Connection** app, see [Active Directory Connection](#)<sup>238</sup> in *Univention Corporate Server - Manual for users and administrators* [2].

It is possible to modify and append custom mappings. Administrators need to create the file `/etc/univention/connector/ad/localmapping.py`. Within that file, they must implement the following function:

```
def mapping_hook(ad_mapping):
    return ad_mapping
```

<sup>237</sup> <https://www.netfilter.org/>

<sup>238</sup> <https://docs.software-univention.de/manual/5.2/en/windows/ad-connection.html#ad-connector-general>

The variable `ad_mapping` influences the mapping. The Active Directory Connection app logs the resulting mapping to `/var/log/univention/connector-ad-mapping.log`, when the administrator restarts Univention AD connector.

## 15.1 Bug reporting

UCS is neither error free nor feature complete. Issues are tracked using [Bugzilla](#)<sup>239</sup>.

1. Create an account.
2. Search for existing entries before opening new reports.
3. Include the version info:

```
$ ucr search --brief ^version/
```

4. Provide enough information to help us reproduce the bug.
5. Conduct some research:
  - Search [Univention Help Knowledge Base](#)<sup>240</sup>
  - Search [Univention Help](#)<sup>241</sup> and ask for help. In addition to our support team many of our partners are also present there. Your questions might also help other users while you may profit from issues already solved for other users.

## 15.2 Debian packaging

This chapter describes how software for Univention Corporate Server is packaged in the Debian format. It allows proper dependency handling and guarantees proper tracking of file ownership. Customers can package their own internal software or use the package mechanism to distribute configuration files consistently to different machines.

Software is packaged as a *source package*, from which one or more *binary packages* can be created. This is useful to create different packages from the same source package. For example the **Samba** source package creates multiple binary packages:

- one containing the file server
- one containing the client commands to access the server
- and several other packages containing documentation, libraries, and common files shared between those packages

The directory should be named *package\_name-version*.

### 15.2.1 Prerequisites and preparation

Some packages are required for creating and building packages.

---

<sup>239</sup> <https://forge.univention.org/bugzilla/index.cgi>

<sup>240</sup> <https://help.univention.com/c/knowledge-base/supported/>

<sup>241</sup> <https://help.univention.com/>

### **build-essential**

This meta package depends on several other packages like compilers and tools to extract and build source packages. Packages must not declare an explicit dependency on this and its dependent packages.

### **devscripts**

This package contains additional scripts to modify source package files like for example `debian/changelog`.

### **dh-make**

This program helps to create an initial `debian/` directory, which can be used as a starting point for packaging new software.

These packages must be installed on the development system. If not, missing packages can be installed on the command line using `univention-install` or through UMC, which is described in the *Univention Corporate Server - Manual for users and administrators* [2].

## 15.2.2 `dh_make`

`dh_make` is a tool, which helps creating the initial `debian/` directory. It is interactive by default and asks several questions about the package to be created.

```
Type of package: single binary, indep binary, multiple binary, library, kernel,
↳module, kernel patch?
[s/i/m/l/k/n]
```

### **s, single binary**

A single architecture specific binary package is created from the source package. This is for software which needs to be compiled individually for different CPU architectures like `i386` and `amd64`.

### **i, indep binary**

A single architecture-independent binary package is created from the source package. This is for software which runs unmodified on all CPU architectures.

### **m, multiple binary**

Multiple binary packages are created from the source package, which can be both architecture independent and dependent.

### **l, library**

Two or more binary packages are created for a compiled library package. The runtime package consists of the shared object file, which is required for running programs using that library. The development package contains the header files and other files, which are only needed when compiling and linking programs on a development system.

### **k, kernel module**

A single kernel-dependent binary package is created from the source package. Kernel modules need to be compiled for each kernel flavor. `dkms` should probably be used instead. This type of packages is not described in this manual.

### **n, kernel patch**

A single kernel-independent package is created from the source package, which contains a patch to be applied against an unpacked Linux kernel source tree. `dkms` should probably be used instead. This type of packages is not described in this manual.

In Debian, a package normally consists of an upstream software archive, which is provided by a third party like the Samba team. This collection is extended by a Debian specific second TAR archive or a patch file, which adds the `debian/` directory and might also modify upstream files for better integration into a Debian system.

When a source package is built, `dpkg-source.1`<sup>242</sup> separates the files belonging to the packaging process from files belonging to the upstream package. For this to work, `dpkg-source` needs the original source either provided as a TAR archive or a separate directory containing the unpacked source. If neither of these is found and `--native` is not given, `dh_make` prints the following warning:

<sup>242</sup> <https://manpages.debian.org/bookworm/dpkg-dev/dpkg-source.1.en.html>

```
Could not find my-package_1.0.orig.tar.gz
Either specify an alternate file to use with -f,
or add --createorig to create one.
```

The warning from `dh_make` states that no pristine upstream archive was found, which prohibits the creation of the Debian specific patch, since the Debian packaging tools have no way to separate upstream files from files specific to Debian packaging. The option `--createorig` can be passed to `dh_make` to create a `.orig.tar.gz` archive before creating the `debian/` directory, if such separation is required.

### 15.2.3 Debian control files

The control files in the `debian/` directory control the package creation process. The following sections provide a short description of these files. A more detailed description is available in the *The Debian GNU/Linux FAQ - Basics of the Debian package management system* [5].

Several files will have the `.ex` suffix, which mark them as examples. To activate these files, they must be renamed by stripping this suffix. Otherwise, the files should be deleted to not clutter up the directory by unused files. In case a file was deleted and needs to be restored, the original templates can be found in the `/usr/share/debhelper/dh_make/debian/` directory.

The `debian/` directory contains some global configuration files, which can be put into two categories: The files `changelog`, `control`, `copyright`, `rules` are required and control the build process of all binary packages. Most other files are optional and only affect a single binary package. Their filename is prefixed with the name of the binary package. If only a single binary package is build from the source package, this prefix can be skipped, but it is good practice to always use the prefix.

The following files are required:

#### **changelog**

Changes related to packaging, not the upstream package. See [debian/changelog](#) (page 146) below for more information.

#### **compat**

The `Debhelper` tools support different compatibility levels. For UCS-3.x the file must contain a single line with the value 7. See [debhelper.7](#)<sup>243</sup> for more details.

#### **control**

Contains control information about the source and all its binary packages. This mostly includes package name and dependency information. See [debian/control](#) (page 143) below for more information.

#### **copyright**

This file contains the copyright and license information for all files contained in the package. See [debian/copyright](#) (page 145) below for more information.

#### **rules**

This is a `Makefile` style file, which controls the package build process. See [debian/rules](#) (page 147) below for more information.

#### **source/format**

This file configures how `dpkg-source.1`<sup>244</sup> separates the files belonging to the packaging process from files belonging to the upstream package. Historically, the Debian source format 1.0 shipped packages as a TAR file containing the upstream source plus one patch file, which contained all files of the `debian/` sub-directory in addition to all changes to upstream files.

The new format 3.0 (`quilt`) replaces the patch file with a second TAR archive containing the `debian/` directory. Changes to upstream files are no longer applied as one giant patch, but split into logical changes and applied using a built-in `quilt.1`<sup>245</sup>.

<sup>243</sup> <https://manpages.debian.org/bookworm/debhelper/debhelper.7.en.html>

<sup>244</sup> <https://manpages.debian.org/bookworm/dpkg-dev/dpkg-source.1.en.html>

<sup>245</sup> <https://manpages.debian.org/bookworm/quilt/quilt.1.en.html>

For simple packages, where there is no distinction between upstream and the packaging entity, the 3.0 (native) format can be used instead, were all files including the `debian/` directory are contained in a single TAR file.

The following files are optional and should be deleted if unused, which helps other developers to concentrate on only the files relevant to the packaging process:

### **README.Debian**

Notes regarding package specific changes and differences to default options, for example compiler options. Will be installed into `/usr/share/doc/package_name/README.Debian`.

### **package.cron.d**

Cron tab entries to be installed. See [dh\\_installcron.1](#)<sup>246</sup> for more details.

### **package.dirs**

List of extra directories to be created. See [dh\\_installdirs.1](#)<sup>247</sup> for more details. May other `dh_` tools automatically create directories themselves, so in most cases this file is unneeded.

### **package.install**

List of files and directories to be copied into the package. This is normally used to partition all files to be installed into separate packages, but can also be used to install arbitrary files into packages. See [dh\\_install.1](#)<sup>248</sup> for more details.

### **package.docs**

List of documentation files to be installed in `/usr/share/doc/package/`. See [dh\\_installdocs.1](#)<sup>249</sup> for more details.

### **package.emacsen-install; package.emacsen-remove; package.emacsen-startup**

Emacs specific files to be installed below `/usr/share/emacs-common/package/`. See [dh\\_installemacsen.1](#)<sup>250</sup> for more details.

### **package.doc-base\***

Control files to install and register extended HTML and PDF documentation. See [dh\\_installdocs.1](#)<sup>251</sup> for more details.

### **package.init.d; package.default**

Start/stop script to manage a system daemon or service. See [dh\\_installinit.1](#)<sup>252</sup> for more details.

### **package.manpage.1; package.manpage.sgml**

Manual page for programs, library functions or file formats, either directly in `troff` or SGML. See [dh\\_installman.1](#)<sup>253</sup> for more details.

### **package.menu**

Control file to register programs with the Debian menu system. See [dh\\_installmenu.1](#)<sup>254</sup> for more details.

### **watch**

Control file to specify the download location of this upstream package. This can be used to check for new software versions. See [uscan.1](#)<sup>255</sup> for more details.

### **package.preinst; package.postinst; package.prerm; package.postrm**

Scripts to be executed before and after package installation and removal. See [debian/preinst](#), [debian/prerm](#), [debian/postinst](#), [debian/postrm](#) (page 148) below for more information.

### **package.maintscript**

---

<sup>246</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installcron.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installcron.1.en.html)

<sup>247</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installdirs.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installdirs.1.en.html)

<sup>248</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_install.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_install.1.en.html)

<sup>249</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installdocs.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installdocs.1.en.html)

<sup>250</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installemacsen.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installemacsen.1.en.html)

<sup>251</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installdocs.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installdocs.1.en.html)

<sup>252</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installinit.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installinit.1.en.html)

<sup>253</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installman.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installman.1.en.html)

<sup>254</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installmenu.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installmenu.1.en.html)

<sup>255</sup> <https://manpages.debian.org/bookworm/devscripts/uscan.1.en.html>

Control file to simplify the handling of configuration files. See `dpkg-maintscript-helper.1`<sup>256</sup> and `dh_installdeb.1`<sup>257</sup> for more information.

Other `debhelper` programs use additional files, which are described in the respective manual pages.

### `debian/control`

The `control` file contains information about the packages and their dependencies, which are needed by `dpkg`. The initial `control` file created by `dh_make` looks like this:

```
Source: testdeb
Section: unknown
Priority: optional
Maintainer: John Doe <user@example.com>
Build-Depends: debhelper (>= 5.0.0)
Standards-Version: 3.7.2

Package: testdeb
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: <insert up to 60 chars description>
<insert long description, indented with spaces>
```

The first block beginning with `Source` describes the source package:

#### **Source**

The name of the source package. Must be consistent with the directory name of the package and the information in the `changelog` file.

#### **Section**<sup>258</sup>

A category name, which is used to group packages. There are many predefined categories like `libs`, `editors`, `mail`, but any other string can be used to define a custom group.

#### **Priority**<sup>259</sup>

Defines the priority of the package. This information is only used by some tools to create installation DVD. More important packages are put on earlier CD, while less important packages are put on later CD.

#### **essential**

Packages are installed by default and `dpkg` prevents the user from easily removing it.

#### **required**

Packages which are necessary for the proper functioning of the system. The package is part of the base installation.

#### **important**

Important programs, including those which one would expect to find on any Unix-like system. The package is part of the base installation.

#### **standard**

These packages provide a reasonably small but not too limited character-mode system.

#### **optional**

Package is not installed by default. This level is recommended for most packages.

#### **extra**

This contains all packages that conflict with some other packages.

#### **Maintainer**

The name and email address of a person or group responsible for the packaging.

<sup>256</sup> <https://manpages.debian.org/bookworm/dpkg/dpkg-maintscript-helper.1.en.html>

<sup>257</sup> [https://manpages.debian.org/bookworm/debhelper/dh\\_installdeb.1.en.html](https://manpages.debian.org/bookworm/debhelper/dh_installdeb.1.en.html)

<sup>258</sup> <https://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections>

<sup>259</sup> <https://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities>

### **Build-Depends; Build-Depends-Indep**

A list of packages which are required for building the package.

### **Standards-version**

Specifies the Debian Packaging Standards version, which this package is conforming to. This is not used by UCS, but required by Debian.

All further blocks beginning with `Package` describes a binary package. For each binary package one block is required.

### **Package**

The name of the binary package. The name must only consist of lower case letters, digits and dashes. If only a single binary package is built from a source package, the name is usually the same as the source package name.

### **Architecture**

Basically there are two types of packages:

- Architecture dependent packages must be build for each architecture like `i386` and `amd64`, since binaries created on one architecture do not run on other architectures. A list of architectures can be explicitly given, or `any` can be used, which is then automatically replaced by the architecture of the system where the package is built.
- Architecture independent packages only need to be built once, but can be installed on all architectures. Examples are documentation, scripts and graphics files. They are declared using `all` in the architecture field.

### **Description**

The first line should contain a short description of up to 60 characters, which should describe the purpose of the package sufficiently. A longer description can be given after that, where each line is indented by a single space. An empty line can be inserted by putting a single dot after the leading space.

Most packages are not self-contained but need other packages for proper function. Debian supports different kinds of dependencies.

### **Depends**

An essential dependency on some other packages, which must be already installed and configured before this package is configured.

### **Recommends**

A strong dependency on some other packages, which should normally be co-installed with this package, but can be removed. This is useful for additional software like plug-ins, which extends the functionality of this package, but is not strictly required.

### **Suggests**

A soft dependency on some other packages, which are not installed by default. This is useful for additional software like large add-on packages and documentation, which extends the functionality of this package, but is not strictly required.

### **Pre-Depends**

A strong dependency on some other package, which must be fully operational even before this package is unpacked. This kind of dependency should be used very sparsely. It's mostly only required for software called from the `.preinst` script.

### **Conflicts**

A negative dependency, which prevents the package to be installed while the other package is already installed. This should be used for packages, which contain the same files or use the same resources, for example TCP port numbers.

### **Provides**

This package declares, that it provides the functionality of some other package and can be considered as a replacement for that package.

### **Replaces**

A declaration, that this package overwrites the files contained in some other package. This deactivates the check normally done by `dpkg` to prevent packages from overwriting files belonging to some other package.

**Breaks**

A negative dependency, which requests the other package to be upgraded before this package can be installed. This is a lesser form of `Conflicts`. `Breaks` is almost always used with a version specification in the form `Breaks: package (<< version)`: This forces `package` to be upgraded to a version greater than `version` before this package is installed.

In addition to literal package names, `debhelper` supports a substitution mechanism: Several helper scripts are capable of automatically detecting dependencies, which are stored in variables.

**`${shlibs:Depends}`**

`dh_shlibdeps` automatically determines the shared library used by the programs and libraries of the package and stores the package names providing them in this variable.

**`${python3:Depends}`**

`dh_python` detects similar dependencies for Python modules.

**`${misc:Depends}`**

Several `Debhelper` commands automatically add additional dependencies, which are stored in this variable.

In addition to specifying a single package as a dependency, multiple packages can be separated by using the pipe symbol (`|`). At least one of those packages must be installed to satisfy the dependency. If none of them is installed, the first package is chosen as the default.

A package name can be followed by a version constraint enclosed in parenthesis. The following operators are valid:

```
<<
    is less than

<=
    is less than or equal to

=
    is equal to

>=
    is greater than or equal to

>>
    is greater than
```

For example:

```
Depends: libexample1 (>= ${binary:Version}),
        exim4 | mail-transport-agent,
        ${shlibs:Depends}, ${misc:Depends}
Conflicts: libgg0, libggi1
Recommends: libncurses5 (>> 5.3)
Suggests: libggi0-target-x (= 1:0.8.5-2)
Replaces: vim-python (<< 6.0), vim-tcl (<= 6.0)
Provides: www-browser, news-reader
```

**debian/copyright**

The `copyright` file contains copyright and license information. For a downloaded source package it should include the download location and names of upstream authors.

```
This package was debianized by John Doe <max@example.com> on
Mon, 21 Mar 2009 13:46:39 +0100.

It was downloaded from <fill in ftp site>

Copyright:
Upstream Author(s): <put author(s) name and email here>
```

(continues on next page)

(continued from previous page)

```
License:
<Must follow here>
```

The file does not require any specific format. Debian recommends to use a machine-readable format, but this is not required for UCS. The format is described in [Machine-readable debian/copyright file](#)<sup>260</sup> at looks like this:

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: Univention GmbH
Upstream-Contact: <package>@univention.de>
Source: https://docs.software-univention.de/

Files: *
Copyright: 2013-2026 Univention GmbH
License: AGPL
```

### debian/changelog

The changelog file documents the changes applied to this Debian package. The initial file created by `dh_make` only contains a single entry and looks like this:

```
testdeb (0.1-1) unstable; urgency=low

 * Initial Release.

-- John Doe <user@example.com> Mon, 21 Mar 2013 13:46:39 +0100
```

For each new package release a new entry must be prepended before all previous entries. The version number needs to be incremented and a descriptive text should be added to describe the change.

The command `debchange` from the `devscripts` package can be used for editing the `changelog` file. For example the following command adds a new version:

```
dch -i
```

After that the `changelog` file should look like this:

```
testdeb (0.1-2) unstable; urgency=low

 * Add more details.

-- John Doe <user@example.com> Mon, 21 Mar 2013 17:55:47 +0100

testdeb (0.1-1) unstable; urgency=low

 * Initial Release.

-- John Doe <user@example.com> Mon, 21 Mar 2013 13:46:39 +0100
```

The date and timestamp must follow the format described in [RFC 2822](#)<sup>261</sup>. `debchange` automatically inserts and updates the current date. Alternatively `date -R` can be used on the command line to create the correct format.

For UCS it is best practice to mention the bug ID of the UCS bug tracker (see [Bug reporting](#) (page 139)) to reference additional details of the bug fixed. Other parties are encouraged to devise similar comments, for example URLs to other bug tracking systems.

<sup>260</sup> <https://dep-team.pages.debian.net/deps/dep5/>

<sup>261</sup> <https://datatracker.ietf.org/doc/html/rfc2822.html>

**debian/rules**

The file `rules` describes the commands needed to build the package. It must use the **Make** syntax *The GNU Make manual* [6]. It consists of several rules, which have the following structure:

```
target: dependencies
    command
    ...
```

Each rule starts with the target name, which can be a filename or symbolic name. Debian requires the following targets:

**clean**

This rule must remove all temporary files created during package build and must return the state of all files back to the same state as when the package is freshly extracted.

**build; build-arch; build-indep**

These rules should configure the package and build either all, all architecture dependent or all architecture independent files.

These rules are called without root permissions.

**binary; binary-arch; binary-indep**

These rules should install the package into a temporary staging area. By default this is the directory `debian/tmp/` below the source package root directory. From there files are distributed to individual packages, which are created as the result of these rules.

These rules are called with root permissions.

Each command line must be indented with one tabulator character. Each command is executed in a separate shell, but long command lines can be split over consecutive lines by terminating each line with a backslash (`\`).

Each rule describes a dependency between the target and its dependencies. **make** considers a target to be out-of-date, when a file with that name `target` does not exist or when the file is older than one of the files it depends on. In that case **make** invokes the given commands to re-create the target.

In addition to filenames also any other word can be used for target names and in dependencies. This is most often used to define *phony* targets, which can be given on the command line invocation to trigger some tasks. The above mentioned `clean`, `build` and `binary` targets are examples for that kind of targets.

**dh\_make** only creates a template for the `rules` file. The initial content looks like this:

```
#!/usr/bin/make -f
# -*- makefile -*-
# Sample debian/rules that uses debhelper.
# This file was originally written by Joey Hess and Craig Small.
# As a special exception, when this file is copied by dh-make into a
# dh-make output file, you may use that output file without restriction.
# This special exception was added by Craig Small in version 0.37 of dh-make.

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

%:
    dh $@
```

Since UCS-3.0 the `debian/rules` file is greatly simplified by using the **dh** sequencer. It is a wrapper around all the different **debhelper** tools, which are automatically called in the right order.

**Tip**

To exactly see which commands are executed when `dpkg-buildpackage` builds a package, invoke `dh target --no-act` by hand, for example `dh binary --no-act` lists all commands to configure, build, install and create

the package.

In most cases it's sufficient to just provide additional configuration files for the individual `debhelper` commands as described in *Debian control files* (page 141). If this is not sufficient, any `debhelper` command can be individually overridden by adding an *override* target to the `rules` file.

For example the following snippet disables the automatic detection of the build system used to build the package and passes additional options:

```
override_dh_auto_configure:
    ./setup --prefix=/usr --with-option-foo
```

Without that explicit override `dh_auto_configure` would be called, which normally automatically detects several build systems like `cmake`, `setup.py`, `autoconf` and others. For these `dh` also passes the right options to configure the default prefix `/usr` and use the right compiler flags.

After configuration the package is built and installed to the temporary staging area in `debian/tmp/`. From there `dh_install` partitions individual files and directories to binary packages. This is controlled through the `debian/package.install` files.

This file can also be used for simple packages, where no build system is used. If a path given in the `debian/package.install` file is not found below `debian/tmp/`, the path is interpreted as relative to the source package root directory. This mechanism is sufficient to install simple files, but fails when files must be renamed or file permissions must be modified.

#### `debian/preinst`, `debian/prerm`, `debian/postinst`, `debian/postrm`

In addition to distributing only files, packages can also be used to run arbitrary commands on installation, upgrades or removal. This is handled by the four *Maintainer scripts*, which are called before and after files are unpacked or removed:

`debian/package.preinst`  
called before files are unpacked.

`debian/package.postinst`  
called after files are unpacked. Mostly used to (re-)start services after package installation or upgrades.

`debian/package.prerm`  
called before files are removed. Mostly used to stop services before a package is removed or upgraded.

`debian/package.postrm`  
called after files have been removed.

The scripts themselves must be shell scripts, which should contain a `#DEBHELPER#` marker, where the shell script fragments created by the `dh_` programs are inserted. Each script is invoked with several parameters, from which the script can determine, if the package is freshly installed, upgraded from a previous version, or removed. The exact arguments are described in the template files generated by `dh_make`.

The maintainer scripts can be called multiple times, especially when errors occur. Because of that the scripts should be idempotent, that is they should be written to *achieve a consistent state* instead of blindly doing the same sequence of commands again and again.

A bad example would be to append some lines to a file on each invocation. The right approach would be to add a check, if that line was already added and only do it otherwise.

#### Warning

Make sure to handle package *upgrades* and *removal* correctly: Both tasks will invoke any existing scripts `prerm` and `postrm`, but with different parameters `remove` and `upgrade` only.

It is important that all these scripts handle error conditions properly: Maintainer scripts should exit with `exit 0` on success and `exit 1` on fail, if things go catastrophically wrong.

On the other hand, an exit code unequal to zero usually aborts any package installation, upgrade or removal process. This prevents any automatic package maintenance and usually requires manual intervention of a human administrator. Therefore, it is essential that maintainer scripts handle error conditions properly and are able to recover an inconsistent state.

## 15.2.4 Building

Before the first build is started, remove all unused files from the `debian/` directory. This simplifies maintenance of the package and helps other maintainers to concentrate on only the relevant differences from standard packages.

The build process is started by invoking the following command:

```
$ dpkg-buildpackage -us -uc
```

The options `-us` and `-uc` disable the PGP signing process of the source and changes files. This is only needed for Debian packages, where all files must be cryptographically signed to be uploaded to the Debian infrastructure.

Additionally, the option `-b` can be added to restrict the build process to only build the binary packages. Otherwise a source package will also be created.

## 15.2.5 Further reading

- *The Debian GNU/Linux FAQ - Basics of the Debian package management system* [5]
- *Debian New Maintainers' Guide* [7]
- *Debian Policy Manual* [8]
- *Debian Developer's Reference* [9]

## 15.3 Bibliography



## BIBLIOGRAPHY

- [1] *ISO 639-1: Alpha-2 code*. International Organization for Standardization, 2010. URL: <https://www.loc.gov/standards/iso639-2/>.
- [2] *Univention Corporate Server - Manual for users and administrators*. Univention GmbH, 2021. URL: <https://docs.software-univention.de/manual/5.2/en/>.
- [3] *Nubus Customization and Modification Manual 1.x*. Univention GmbH, 2024-2026. URL: <https://docs.software-univention.de/nubus-customization/latest/en/>.
- [4] *Univention Corporate Server 5.2 Architecture*. Univention GmbH, 2023. URL: <https://docs.software-univention.de/architecture/5.2/en/>.
- [5] *The Debian GNU/Linux FAQ - Basics of the Debian package management system*. Debian, 2019. URL: <https://www.debian.org/doc/manuals/debian-faq/pkg-basics.en.html>.
- [6] *The GNU Make manual*. Free Software Foundation, 2020. URL: <https://www.gnu.org/software/make/manual/>.
- [7] *Debian New Maintainers' Guide*. Debian, 2015. URL: <https://www.debian.org/doc/manuals/debmake-doc/index.en.html>.
- [8] *Debian Policy Manual*. Debian, 2020. URL: <https://www.debian.org/doc/debian-policy/>.
- [9] *Debian Developer's Reference*. Debian, 2021. URL: <https://www.debian.org/doc/manuals/developers-reference/>.



## PYTHON MODULE INDEX

### U

`udm_modules_globals`, [72](#)  
`univention.admin.hook`, [94](#)



## Non-alphabetical

\$PATH, 12  
 |UCSUDL|, *see* directory listener  
 |UCSUDM|, *see* directory manager

## A

--acl  
     ucs\_registerLDAPExtension command  
         line option, 35  
 add() (*built-in function*), 109  
 addEmptyValue (*LDAP\_Search.UDM\_API attribute*), 87  
 addEmptyValue (*Python\_API attribute*), 86  
 apache, *see* web services  
 app center, 117  
 appendEmptyValue (*Python\_API attribute*), 86  
 as\_root() (*in module high\_level*), 50  
 atd/autostart, 25, 27  
 attribute (*LDAP\_Search.UDM\_API attribute*), 87  
 attribute (*Python\_API attribute*), 86  
 attribute (*UDM\_Attribute attribute*), 84  
 attributes (*in module high\_level*), 47  
 attributes (*in module your\_module*), 44

## B

base (*LDAP\_Search.UDM\_API attribute*), 87  
 bug, *see* bugzilla  
 bugzilla, 139

## C

childs (*in module udm\_modules\_globals*), 72  
 clean() (*in module high\_level*), 49  
 clean() (*in module your\_module*), 46  
 config registry, 11, 15  
     categories, 20  
     descriptions, 18  
     examples, 2325  
     repository, 9  
     services, 20  
     template, 1517  
     template file, 21  
 configuration files, 15  
 create() (*in module high\_level*), 48  
 custom attributes, *see* extended attributes

## D

database, 133  
     mariadb, 133  
     mysql, 133  
     postgresql, 133  
 depends (*UDM\_Attribute attribute*), 84  
 description (*in module high\_level*), 47  
 description (*in module your\_module*), 44  
 description (*LDAP\_Search.UDM\_API attribute*), 86  
 die, 33  
 diff() (*in module high\_level*), 50  
 directory listener, 43  
     cache, 65  
     credentials, 65  
     debug, 65  
     example module, 55  
     modrdn, 57  
     notifier ID, 66  
     verify, 66  
 Directory manager  
     module extension, 35  
 directory manager, 71  
     custom modules, 72  
     extended attributes, 87  
     extension modules packaging, 98  
     hook extension, 35  
     hooks packaging, 98  
     LDAP search, 84  
     syntax extension, 35  
     syntax override, 83  
     UDM syntax extension packaging, 99  
 domain join, 29  
     domain credentials, 41, 42  
     join script, 29  
     join status, 29  
     machine credential change, 42  
     running, 29  
     unjoin, 37

## E

empty\_value (*UDM\_Attribute attribute*), 84  
 environment variable  
     \$PATH, 12  
     atd/autostart, 25, 27  
     hosts/allow/, 24  
     hosts/deny/, 24

JS\_LAST\_EXECUTED\_VERSION, 33  
 ldap/backup, 67  
 ldap/hostdn, 29, 41  
 ldap/master, 66, 67  
 listener/cache/filter, 45  
 notifier/server, 66  
 notifier/server/port, 66  
 print/papersize, 23  
 repository/online/component/NAME, 9, 120  
 repository/online/component/NAME/defaultpackages, 121  
 repository/online/component/NAME/description, 120  
 repository/online/component/NAME/layout, 120  
 repository/online/component/NAME/localmirror, 120  
 repository/online/component/NAME/password, 120  
 repository/online/component/NAME/prefix, 120  
 repository/online/component/NAME/server, 9, 120  
 repository/online/component/NAME/username, 120  
 repository/online/component/NAME/version, 120  
 repository/online/server, 120  
 security/packetfilter/use\_packages, 137  
 server/password/change, 42  
 server/password/interval, 42  
 ucs/web/overview/entries/admin/PACKAGE/OPTION, 115  
 ucs/web/overview/entries/service/PACKAGE/OPTION, 115  
 UNIVENTION\_APP\_IDENTIFIER, 36  
 VERSION, 33  
 version/patchlevel, 131  
 version/version, 131  
 Errata updates  
   UCS 4.2 erratum 311, 43  
   UCS 4.3 erratum 85, 31  
   UCS 4.3 erratum 427, 67  
   UCS 5.0 erratum 164, 68  
   UCS 5.2 erratum 416, 94, 95, 97  
 error\_handler() (in module high\_level), 51  
 error\_message (UDM\_Attribute attribute), 84  
 examples  
   config registry, 2325  
   multi file, 24  
   services, 25  
   single file, 23  
 extended attributes, 87  
   hooks, 94  
   options, 93  
   selection list, 91

## F

filter (in module low\_level), 52  
 filter (in module your\_module), 44  
 filter (LDAP\_Search.UDM\_API attribute), 86  
 filter (Python\_API attribute), 86

## G

get () (built-in function), 109  
 get\_active () (in module high\_level), 48  
 get\_attributes () (in module high\_level), 47  
 get\_attributes () (in module your\_module), 44  
 get\_description () (in module high\_level), 47  
 get\_description () (in module your\_module), 44  
 get\_ldap\_filter () (in module high\_level), 47  
 get\_ldap\_filter () (in module your\_module), 44  
 get\_listener\_module\_class () (in module high\_level), 48  
 get\_listener\_module\_instance () (in module high\_level), 47  
 get\_name () (in module high\_level), 47  
 get\_name () (in module your\_module), 43  
 get\_priority () (in module high\_level), 47  
 get\_priority () (in module your\_module), 45

## H

handle\_every\_delete (in module your\_module), 45  
 handler () (in module low\_level), 53  
 hook\_ldap\_addlist () (univention.admin.hook.simpleHook method), 95  
 hook\_ldap\_modlist () (univention.admin.hook.simpleHook method), 96  
 hook\_ldap\_post\_create () (univention.admin.hook.simpleHook method), 95  
 hook\_ldap\_post\_remove () (univention.admin.hook.simpleHook method), 96  
 hook\_ldap\_pre\_create () (univention.admin.hook.simpleHook method), 95  
 hook\_ldap\_pre\_modify () (univention.admin.hook.simpleHook method), 96  
 hook\_ldap\_pre\_remove () (univention.admin.hook.simpleHook method), 96  
 hook\_open () (univention.admin.hook.simpleHook method), 95  
 hosts/allow/, 24  
 hosts/deny/, 24

## I

--icon  
   ucs\_registerLDAPExtension command  
   line option, 36  
 initialize () (in module high\_level), 49  
 initialize () (in module your\_module), 45

## J

join, *see* domain join  
 join script  
   domain join, 29

- exit codes, 32
- helpers, 32
- library, 32
- return codes, 32
- writing, 30

joinscript\_check\_any\_version\_executed, 33

joinscript\_check\_specific\_version\_executed version, 33

joinscript\_check\_version\_in\_range\_executed min max, 33

joinscript\_extern\_init join-script, 33

joinscript\_init, 33

joinscript\_remove\_script\_from\_status\_file name, 33

joinscript\_save\_current\_version, 33

## K

key (*UDM\_Objects* attribute), 85

Knowledge Base  
KB 13149, 65

## L

label (*UDM\_Objects* attribute), 85

label\_format (*UDM\_Attribute* attribute), 84

layout (*in module udm\_modules\_globals*), 74

LDAP

- access control list extension, 35
- schema, 68
- schema extension, 35

ldap/backup, 67

ldap/hostdn, 29, 41

ldap/master, 66, 67

ldap\_filter (*in module high\_level*), 47

ldap\_filter (*in module your\_module*), 44

LDAP\_Search (*built-in class*), 85

LDAP\_Search.UDM\_API (*built-in class*), 86

ldapattribute (*LDAP\_Search.UDM\_API* attribute), 87

ldapvalue (*LDAP\_Search.UDM\_API* attribute), 87

listener, *see* directory listener

- schema replication, 68

listener/cache/filter, 45

listener\_module\_class (*in module high\_level*), 48

lo (*in module high\_level*), 52

localisation, *see* translation

logger (*in module high\_level*), 52

long\_description (*in module udm\_modules\_globals*), 72

## M

management console, 103

- disable, 113
- files, 104
- LDAP, 112
- Module, 112
- module, 105, 113
- system, 105
- umc-modules, 104

- XML, 105

map () (*univention.admin.hook.simpleHook* method), 94

mapping (*in module udm\_modules\_globals*), 75

--messagecatalog

- ucs\_registerLDAPExtension command
- line option, 35

modify () (*in module high\_level*), 48

modrdrn (*in module your\_module*), 44

module

- template, 17
- udm\_modules\_globals, 72
- univention.admin.hook, 94

module (*in module udm\_modules\_globals*), 72

module/add () (*built-in function*), 110

module/get () (*built-in function*), 110

module/put () (*built-in function*), 110

module/query () (*built-in function*), 110

module/remove () (*built-in function*), 110

multi file

- examples, 24
- template, 16

## N

--name

- ucs\_registerLDAPExtension command
- line option, 35

name (*in module high\_level*), 47

name (*in module your\_module*), 43

name (*LDAP\_Search.UDM\_API* attribute), 86

notifier/server, 66

notifier/server/port, 66

## O

object (*class in udm\_modules\_globals*), 76

operations (*in module udm\_modules\_globals*), 72

options (*in module udm\_modules\_globals*), 73

options.default (*in module udm\_modules\_globals*), 73

options.editable (*in module udm\_modules\_globals*), 73

options.long\_description (*in module udm\_modules\_globals*), 73

options.objectClasses (*in module udm\_modules\_globals*), 73

options.short\_description (*in module udm\_modules\_globals*), 73

## P

packaging, 3

- build dependencies, 139
- check for errors, 133
- debian, 139
- library functions, 134
- modify existing package, 3
- new package, 4
- package repository, 8

po (*in module high\_level*), 52

post\_run () (*in module high\_level*), 50

postrun() (in module *your\_module*), 46

postup  
  updater, 131

pre\_run() (in module *high\_level*), 49

prerun() (in module *your\_module*), 46

preup  
  updater, 131

print/papersize, 23

priority (in module *high\_level*), 47

priority (in module *your\_module*), 45

property\_descriptions (in module *udm\_modules\_globals*), 73

property\_descriptions.default (in module *udm\_modules\_globals*), 74

property\_descriptions.dontsearch (in module *udm\_modules\_globals*), 74

property\_descriptions.editable (in module *udm\_modules\_globals*), 74

property\_descriptions.identifies (in module *udm\_modules\_globals*), 74

property\_descriptions.long\_description (in module *udm\_modules\_globals*), 74

property\_descriptions.may\_change (in module *udm\_modules\_globals*), 74

property\_descriptions.multivalue (in module *udm\_modules\_globals*), 74

property\_descriptions.options (in module *udm\_modules\_globals*), 74

property\_descriptions.required (in module *udm\_modules\_globals*), 74

property\_descriptions.short\_description (in module *udm\_modules\_globals*), 74

property\_descriptions.syntax (in module *udm\_modules\_globals*), 74

put() (built-in function), 109

Python 3  
  migration, 69

Python\_API (built-in class), 86

## Q

query() (built-in function), 109

## R

regex (*UDM\_Attribute attribute*), 84

regex (*UDM\_Objects attribute*), 85

registry, *see* config registry

remove() (built-in function), 109

remove() (in module *high\_level*), 49

repositories  
  external, 119

repository, *see* packaging

repository/online/component/NAME, 9

repository/online/component/NAME/server,  
  9, 120

repository/online/server, 120

RFC  
  RFC 2254, 44

  RFC 2822, 146

## S

--schema  
  ucs\_registerLDAPExtension command  
  line option, 35

script  
  template, 17

security/packetfilter/use\_packages, 137

server password change, *see* domain join

server/password/change, 42

server/password/interval, 42

services  
  examples, 25

setdata() (in module *low\_level*), 54

short\_description (in module *udm\_modules\_globals*), 72

simple (*UDM\_Objects attribute*), 85

simpleHook (class in *univention.admin.hook*), 94

single file  
  examples, 23

  template, 15

static\_values (*UDM\_Attribute attribute*), 84

syntax\_name (*Python\_API attribute*), 86

## T

template  
  config registry, 1517

  module, 17

  multi file, 16

  script, 17

  single file, 15

template (in module *udm\_modules\_globals*), 75

translation, 126

## U

UCR, *see* config registry

ucr (in module *high\_level*), 52

UCS source code  
  UCS source: doc/developer-reference/join/join-templ  
  30

  UCS source: doc/developer-reference/listener/modrdr  
  57

  UCS source: doc/developer-reference/listener/obj.py  
  62

  UCS source: doc/developer-reference/listener/printu  
  58

  UCS source: doc/developer-reference/listener/simple  
  56

  UCS source: doc/developer-reference/packaging/testc  
  3

  UCS source: doc/developer-reference/ucr/hosts/  
  24

  UCS source: doc/developer-reference/ucr/papersize/  
  23

  UCS source: doc/developer-reference/ucr/service/  
  25

  UCS source: manage-  
  ment/univention-directory-listener/examples/com  
  47

UCS source: management/univention-directory-listener/examples (univention.admin.hook.simpleHook method), 47, 55

UCS source: packaging/univention-directory-manager-modules/update-example/, 77, 83

ucs/web/overview/entries/admin/PACKAGE/OPTION, 115

ucs/web/overview/entries/service/PACKAGE/OPTION, 115

ucs\_registerLDAPExtension command line option

- acl, 35
- icon, 36
- messagecatalog, 35
- name, 35
- schema, 35
- ucsversionend, 35
- ucsversionstart, 35
- udm\_hook, 35
- udm\_module, 35
- udm\_syntax, 35
- umcmmessagecatalog, 35
- umcregistration, 36

--ucsversionend

ucs\_registerLDAPExtension command line option, 35

--ucsversionstart

ucs\_registerLDAPExtension command line option, 35

UDM, *see* directory manager

UDM\_Attribute (*built-in class*), 84

udm\_filter (*UDM\_Attribute attribute*), 84

--udm\_hook

ucs\_registerLDAPExtension command line option, 35

--udm\_module

ucs\_registerLDAPExtension command line option, 35

udm\_module (*UDM\_Attribute attribute*), 84

udm\_modules (*UDM\_Objects attribute*), 85

udm\_modules\_globals

module, 72

UDM\_Objects (*built-in class*), 85

--udm\_syntax

ucs\_registerLDAPExtension command line option, 35

UMC, *see* management console

--umcmmessagecatalog

ucs\_registerLDAPExtension command line option, 35

--umcregistration

ucs\_registerLDAPExtension command line option, 36

Univention Management Console, *see* management console

UNIVENTION\_APP\_IDENTIFIER, 36

univention.admin.hook

module, 94

update, *see* updater

update-example/, 77, 83

postup, 131

preup, 131

repositories, 131

scripts, 131

system update, 131

upgrade, *see* updater

use\_objects (*UDM\_Objects attribute*), 85

## V

value (*LDAP\_Search.UDM\_API attribute*), 87

value (*Python\_API attribute*), 86

VERSION, 33

version/patchlevel, 131

version/version, 131

viewonly (*LDAP\_Search.UDM\_API attribute*), 87

viewonly (*Python\_API attribute*), 86

virtual (*in module udm\_modules\_globals*), 75

## W

web services, 115