

Univention Developer Reference



Manual for developers

Alle Rechte vorbehalten./ All rights reserved.

The mentioned brand names and registered trademarks are owned by the respective legal owners in each case.

Linux® is a registered trademark of Linus Torvalds.

Table of Contents

Foreword	9
1. Packaging software	11
1.1. Introduction	11
1.2. Preparations	11
1.3. Example: Re-building an UCS package	11
1.4. Example: Creating a new UCS package	12
1.5. Setup repository	17
1.6. Building packages through the openSUSE Build Service	17
2. Univention Config Registry	19
2.1. Using UCR	19
2.1.1. Using UCR from shell	19
2.1.2. Using UCR from Python	21
2.2. Configuration files	22
2.2.1. <code>debian/package.univention-config-registry</code>	22
2.2.1.1. File	23
2.2.1.2. Multifile	24
2.2.1.3. Script	25
2.2.1.4. Module	25
2.2.2. <code>debian/package.univention-config-registry-variables</code>	25
2.2.3. <code>debian/package.univention-config-registry-categories</code>	26
2.2.4. <code>debian/package.univention-config-registry-services</code>	27
2.3. UCR Template files <code>conffiles/path/to/file</code>	28
2.4. Build integration	29
2.5. Examples	29
2.5.1. Minimal File example	29
2.5.2. Multifile example	31
2.5.3. Services	32
3. Domain Join	35
3.1. Join scripts	35
3.2. Join status	35
3.3. Running join scripts	35
3.4. Writing join scripts	36
3.4.1. Basic join script example	36
3.4.2. Join script exit codes	38
3.4.3. Join script libraries	38
3.4.3.1. <i>univention-join</i>	38
3.4.3.2. <i>shell-univention-lib</i>	40
3.5. Writing unjoin scripts	43
4. Lightweight Directory Access Protocol (LDAP) in UCS	47
4.1. General	47
4.2. Packaging LDAP Schema Extensions	47
4.3. Packaging LDAP ACL Extensions	48
4.4. LDAP secrets	50
4.4.1. Password change	50
5. Univention Directory Listener	53
5.1. Structure of Listener Modules	53
5.2. Listener Tasks and Examples	56
5.2.1. Basic Example	56
5.2.2. Rename and Move	57
5.2.3. Full Example with Packaging	58
5.2.4. A Little Bit more Object Oriented	62
5.3. Technical Details	65

5.3.1. User-ID and Credentials	65
5.3.2. Internal Cache	65
5.3.2.1. univention-directory-listener-ctrl	65
5.3.2.2. univention-directory-listener-dump	65
5.3.2.3. univention-directory-listener-verify	66
5.3.2.4. get_notifier_id.py	66
5.3.3. Internal working	66
6. Univention Directory Manager (UDM)	69
6.1. Introduction	69
6.2. Packaging Extended Attributes	70
6.2.1. Selection lists	74
6.2.1.1. Static selections	74
6.2.1.2. Dynamic selections	74
6.2.2. Known issues	76
6.2.3. Extended Options	76
6.2.4. Extended Attribute Hooks	77
6.3. UDM Modules	79
6.4. UDM Syntax	79
6.4.1. UDM Syntax Override	80
6.4.2. UDM LDAP search	81
6.5. Packaging UDM Hooks	84
6.6. Packaging UDM Extension Modules	85
6.7. Packaging UDM Syntax Extension	86
7. Univention Management Console (UMC)	89
7.1. Architecture	89
7.2. Asynchronous Framework	90
7.3. Protocol UMCP 2.0	91
7.3.1. Data flow	91
7.3.2. Authentication	91
7.3.3. Message format	91
7.3.3.1. Message header	91
7.3.3.2. Message body	92
7.3.4. Examples	92
7.4. Protocol HTTP for UMC	93
7.4.1. Examples	93
7.5. UMC files	94
7.5.1. <i>debian/package.umc-modules</i>	94
7.5.2. UMC Module Declaration File	95
7.6. Local System Module	95
7.6.1. Python API	95
7.6.2. UMC module API (Python and JavaScript)	95
7.6.2.1. XML definition	96
7.6.2.2. Python module	97
7.6.2.3. UMC store API	99
7.6.3. Packaging	100
7.7. Domain LDAP Module	102
7.8. Disabling a Module	103
8. Web services	105
8.1. Extending the overview page	105
9. App Center	107
9.1. Requirements	107
9.2. Packaging for the App Center	108
9.3. Next steps	108
9.4. Application meta file	109

9.5. Optional application files	116
9.6. Uploading the application	116
9.7. Notifications	117
9.8. Updates for the application	117
9.9. Integrating the Application in UCS	118
9.9.1. Automatic integration done by the App Center	118
9.9.2. Scope of the vendor	118
9.10. Best practices	119
9.10.1. Registration of LDAP and UDM Extensions	119
10. Integration of external repositories	121
10.1. Integration of repository components via Univention Management Console	121
10.2. Integration of repository components via Univention Configuration Registry	122
11. Translating UCS	123
11.1. Univention Management Console translations	123
11.1.1. Install needed tools	123
11.1.2. Obtain a current checkout of the UCS Subversion repository	123
11.1.3. Create a new translation package	123
11.1.4. Edit translation files	123
11.1.5. Update the translation package	124
11.1.6. Build the translation package	124
12. Univention Updater	127
12.1. Separate repositories	127
12.2. Updater scripts	127
12.2.1. Digital signature	128
12.3. Release update walkthrough	128
13. <i>Single Sign-On</i> : Integrating a service provider into UCS	129
13.1. Register new service provider via <i>udm</i>	129
13.2. Get information required by the service provider	129
13.3. Add direct login link to <i>ucs-overview</i> page	130
14. Miscellaneous	131
14.1. Databases	131
14.1.1. PostgreSQL	131
14.1.2. MySQL	131
14.2. UCS lint	131
14.3. Function Libraries	133
14.3.1. <i>shell-univention-lib</i>	133
14.3.2. <i>python-univention-lib</i>	133
14.4. Login Access Control	134
14.5. Network Packet Filter	135
14.5.1. Filter rules by Univention Configuration Registry	135
14.5.2. Local filter rules via <i>iptables</i> commands	136
14.5.3. Testing Univention Firewall settings	136
A. Bug reporting	137
B. Debian packaging	139
B.1. Prerequisites and preparation	139
B.2. <i>dh_make</i>	139
B.3. Debian control files	140
B.3.1. <i>debian/control</i>	142
B.3.2. <i>debian/copyright</i>	145
B.3.3. <i>debian/changelog</i>	146
B.3.4. <i>debian/rules</i>	146
B.3.5. <i>debian/preinst</i> , <i>debian/prerm</i> , <i>debian/postinst</i> , <i>debian/postrm</i>	148
B.4. Building	149
B.5. Further reading	149

Bibliography	151
Index	153

List of Examples

2.1. Use of <code>ucr set</code>	19
2.2. Use of <code>ucr get</code>	20
2.3. Use of <code>is_ucr_true</code>	20
2.4. Use of <code>ucr unset</code>	20
2.5. Use of <code>ucr shell</code>	20
2.6. Reading a Univention Configuration Registry variable in Python	21
2.7. Reading boolean Univention Configuration Registry variables in Python	21
2.8. Changing Univention Configuration Registry variables in Python	21
2.9. Setting and unsetting Univention Configuration Registry variables in Python	22
3.1. Service registration in join script	40
3.2. Service unregistration in unjoin script	41
3.3. Check for unused service in unjoin script	41
3.4. Extension registration in join script	42
3.5. Schema unregistration in unjoin script	43
4.1. Schema registration in join script	48
4.2. LDAP ACL registration in join script	49
4.3. Server password change example	51
6.1. Extended Attribute for custom LDAP schema	73
6.2. Dynamic selection list for Extended Attributes	75
6.3. Extended Option	77
7.1. Authentication request	92
7.2. Search for users	93
7.3. Authentication request	93
7.4. search for users	94
7.5. UMC module category examples	97
14.1. Local firewall rule	135
14.2. Using <code>nmap</code> for firewall port testing	136

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 Appendix A) or send an e-mail to [<feedback@univention.de>](mailto:feedback@univention.de).

Chapter 1. Packaging software

1.1. Introduction	11
1.2. Preparations	11
1.3. Example: Re-building an UCS package	11
1.4. Example: Creating a new UCS package	12
1.5. Setup repository	17
1.6. Building packages through the openSUSE Build Service	17

This chapter describes how software for UCS is packaged. For more details on packaging software in the Debian format, see Appendix B

1.1. Introduction

Feedback 

UCS is based on the Debian distribution, which is using 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)`. 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.

1.2. Preparations

Feedback 

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

```
apt-get install subversion build-essential devscripts
```

1.3. Example: Re-building an UCS package

Feedback 

Source code: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/packaging/testdeb/>

Procedure 1.1. 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 Subversion version control system or download the source code of the currently packaged version.
 - Checkout example package from Univention Subversion

Example: Creating a new UCS package

```
SERVER=http://forge.univention.org
svn co $SERVER/svn/dev/branches/ucs-4.1/ucs-4.1-0/base/
univention-ssh
cd univention-ssh/
```

- Fetch the source code from the Univention Repository server
 - a. Enable unmaintained and source repository once

```
ucr set repository/online/unmaintained=yes \
      repository/online/sources=yes
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. Build the binary package

```
dpkg-buildpackage -uc -us -b
```

5. Locally install the new binary package

```
sudo dpkg -i ../univention-ssh_*_*.deb
```

1.4. Example: Creating a new UCS package

Feedback 

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/env python
"""
Example for creating UCS packages.
"""
import time

now = time.localtime()
filename = '/tmp/testdeb-%s' % time.strftime('%y%m%d%H%M', now)
tmpfile = open(filename, 'a')
tmpfile.close()
```

In addition to the files to be installed some meta-data 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(8)` command provided by the *dh-make* packet:

```
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 Section B.2.

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:

```
Source: testdeb
Section: univention
Priority: optional
Maintainer: John Doe <user@example.com>
Build-Depends: debhelper (>= 7)
Standards-Version: 3.7.2

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
```

Example: Creating a new UCS package

```
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:
http://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 that tabulators must be used for indentation in this file.

debian/testdeb.install

To compensate the missing Makefile, `dh_install(1)` 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 that 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 (see Section 2.1.1):

```
#!/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 redistributable. 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/compat, debian/source/format

These files control some internal aspects of the package build process. They can be ignored for the moment and are further described in Section B.3.

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: source package testdeb
dpkg-buildpackage: source version 0.1-1
dpkg-buildpackage: source changed by John Doe <user@example.com>
dpkg-buildpackage: host architecture amd64
dpkg-source --before-build testdeb
fakeroot debian/rules clean
dh clean
dh_testdir
dh_auto_clean
dh_clean
dpkg-source -b testdeb
dpkg-source: Information: Quellformat »3.0 (native)« wird verwendet
dpkg-source: Information: testdeb wird in testdeb_0.1-1.tar.gz gebaut
dpkg-source: Information: testdeb wird in testdeb_0.1-1.dsc gebaut
debian/rules build
dh build
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
fakeroot debian/rules binary
dh binary
```


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

1.5. Setup repository

 Feedback 

Until now the binary package is only available locally, thus for installation it needs to be copied manually to each host and must be installed manually using `dpkg -i`. If the package required additional dependencies, the installation process will abort, since packages are not downloaded by `dpkg`, but by `apt`. To support automatic installation and dependency resolution, the package must be put into an `apt` repository, which needs to be made available through `http` or some other mechanism.

For this example the repository is created below `/var/www/repository/`, which is exported by default on all UCS systems, where `apache2` is installed. Below that directory several other sub-directories and files must be created to be compatible with the UCS Updater. The following example commands create a repository for UCS version 4.1 with the component name `testcomp`:

```
WWW_BASE="/var/www/repository/4.1/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" )
```

This repository can be included on any UCS system by appending the following line to `/etc/apt/sources.list`, assuming the FQDN of the host providing the repository is named `repository.server`:

```
deb http://repository.server/repository/4.1/maintained/component
testcomp/all/
```

Note

It is important that the directory, from where the `apt-ftparchive` command is invoked, matches 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 is also used as the base URL for all packages listed in the `Packages` file.

Instead of editing the `sources.list` file directly, the repository can also be included as a component, which can be configured by setting several UCR variables. As UCR variables can also be configured through UDM policies, this simplifies the task of installing packages from such a repository on many hosts. For the repository above the following variables need to be set:

```
ucr set \
  repository/online/component/testcomp=yes \
  repository/online/component/testcomp/server=repository.server \
  repository/online/component/testcomp/prefix=repository
```

1.6. Building packages through the openSUSE Build Service

 Feedback 

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

Building packages through the openSUSE Build Service

for other distributions, it can also be used for Univention Corporate Server builds. The build target for UCS 4.1 is called *Univention UCS 4.1*. Note that OBS doesn't handle the integration steps described in later chapters (e.g. the use of Univention Configuration Registry templates).

Chapter 2. Univention Config Registry

2.1. Using UCR	19
2.1.1. Using UCR from shell	19
2.1.2. Using UCR from Python	21
2.2. Configuration files	22
2.2.1. <code>debian/package.univention-config-registry</code>	22
2.2.1.1. File	23
2.2.1.2. Multifile	24
2.2.1.3. Script	25
2.2.1.4. Module	25
2.2.2. <code>debian/package.univention-config-registry-variables</code>	25
2.2.3. <code>debian/package.univention-config-registry-categories</code>	26
2.2.4. <code>debian/package.univention-config-registry-services</code>	27
2.3. UCR Template files <code>conffiles/path/to/file</code>	28
2.4. Build integration	29
2.5. Examples	29
2.5.1. Minimal File example	29
2.5.2. Multifile example	31
2.5.3. Services	32

The Univention Config 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.

2.1. Using UCR

Feedback 

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

2.1.1. Using UCR from shell

Feedback 

`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.

Example 2.1. 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`.

Example 2.2. Use of `ucr get`

```
case "$(ucr get system/role)" in
  domaincontroller_*)
    echo "Running on a UCS Domain Controller"
    ;;
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.

Example 2.3. Use of `is_ucr_true`

```
. /usr/share/univention-lib/ucr.sh
if is_ucr_true repository/online/unmaintained
then
  echo "Unmaintained is enabled"
fi
```

```
ucr unset key ...
```

Unset the Univention Configuration Registry variable `key`.

Example 2.4. Use of `ucr unset`

```
ucr unset print/papersize variable/namme
```

```
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 (`_`).

Example 2.5. Use of `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 than 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.

2.1.2. 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:

Example 2.6. 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>')
```

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 latter 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.

Example 2.7. Reading boolean Univention Configuration Registry variables in Python

```
if ucr.is_true('repository/online/unmaintained'):
    print "unmaintained is explicitly enabled"
if ucr.is_true('repository/online/unmaintained', True):
    print "unmaintained is enabled"
if ucr.is_false('repository/online/unmaintained'):
    print "unmaintained is explicitly disabled"
if ucr.is_false('repository/online/unmaintained', True):
    print "unmaintained is disabled"
```

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

Example 2.8. 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` 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.

Example 2.9. 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'])
```

2.2. Configuration files

 Feedback 

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: A *single file template* consists of only a single file, from which the target file is created, while a *multi file template* can consist of multiple file fragments, which are concatenated to form the target file. See Section 2.3 below for more information.

`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. See Section 2.2.1 below for more information.

`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. See Section 2.2.2 below for more information.

`debian/package.univention-config-registry-categories`

This optional file can add additional categories to group UCR variables. See Section 2.2.3 below for more information.

`debian/package.univention-config-registry-services`

This optional file is used to define long running services. See Section 2.2.4 below for more information.

In addition to these files code needs to be inserted into the package maintainer scripts (see Section B.3.5), which registers and unregisters 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`.

2.2.1. `debian/package.univention-config-registry`

 Feedback 

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.

2.2.1.1. File

[Feedback](#) 

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 Section 2.3.

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, changes):
    pass
def postinst(config_registry, changes):
    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)`.

debian/package.univention-config-registry

`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 list of changed variables as documented in Section 2.2.1.3.

2.2.1.2. Multifile

Feedback 

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 Section 2.3. 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.

2.2.1.3. Script

Feedback 

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.

2.2.1.4. Module

Feedback 

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 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, changes):
    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.

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

Feedback 

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

`debian/package.univention-config-registry-categories`

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>
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 [ISO639].

`Type` (required)

The syntax type for the value. This is unused in UCS-3.1, but future versions might use this for validating the input. Valid values include `str` for strings, `bool` for boolean values, and `int` for integers.

`ReadOnly` (optional)

This declares a variable as read-only and prohibits changing the value through UMC. The restriction is not 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 Section 2.2.3.

2.2.3. `debian/package.univention-config-registry-categories`

Feedback 

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 Section 2.2.2). 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 [ISO639].

icon (required)

The file name 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.

2.2.4. *debian/package.univention-config-registry-services*

Feedback 

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  
icon=<service/icon_name>  
programs=<executable>
```

[*service-name*]

For each daemon one section needs to be created. The *service-name* should match the name of the init-script in */etc/init.d/*.

description[*language*] (required)

A descriptive text for the service. The description can be given in multiple languages, using the two-letter-code following [ISO639].

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 should not be started automatically, but invoking the init-script directly with *start* should still start the service.

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.

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

`icon` (optional)

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

2.3. UCR Template files `conffiles/path/to/file`

Feedback 

For each file, which should be written, one or more template files need be to created below the `conffiles/` directory. For a single-File template (see Section 2.2.1.1), the filename must match the filename given in the `File:` stanza of the `file` entry itself. For a Multifile template (see Section 2.2.1.2), 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`¹ 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 Multifile templates, it should only be on the top the first subfile.

¹ Historically Univention Configuration Registry was named “Univention Base Config”. For backward compatibility the alias `baseConfig` is still provided. It should not be used anymore and will be removed in a future version of UCS.

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 `#`, 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.

2.4. Build integration

Feedback 

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

```
override_dh_auto_install:
univention-install-config-registry
dh_auto_install
```

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 un-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}, ...
```

2.5. Examples

Feedback 

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.

2.5.1. Minimal File example

Feedback 

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 papersize.

Source code: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/ucr/papersize/>

Minimal File example

conffiles/etc/papersize

The template file only contains one line. Please note that this file does not start with the “UCR-WARNING”, 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

#DEBHELPER#

ucr set print/papersize?a4

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 Section B.3.5) to register and unregister the templates during package installation and removal.

```
#!/usr/bin/make -f

override_dh_auto_install:
    dh_auto_install
    univention-install-config-registry

%:
    dh $@
```

Note that 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` via `debian/papersize.substvars`.

```
Source: papersize
Section: univention
Priority: optional
```

```
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends: debhelper (>= 7),
  univention-config-dev,
Standards-Version: 3.7.2

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:
http://www.univention.de/
```

2.5.2. Multifile example

Feedback 

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)` for more details.

Source code: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/ucr/hosts/>

```
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 subfiles.

```
@@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
@!@
```

```
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 verbotende Zugriffsregel für Systemdienste,
  z.B. "ALL: ALL".
Type=str
Categories=service-net

```

2.5.3. Services

Feedback 

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

Source code: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/ucr/service/>

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.

```

#! /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
#
# 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 "$1" in
    start)
        # check ucr autostart setting
        IAL="/usr/share/univention-config-registry/init-autostart.lib"
        if [ -f "$IAL" ]; then
            . "$IAL"
            check_autostart atd atd/autostart
        fi
        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

#DEBHELPER#

ucr set atd/autostart=yes

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 (>= 7),
  univention-config-dev,
Standards-Version: 3.7.2

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:
  http://www.univention.de/
```

Chapter 3. Domain Join

3.1. Join scripts	35
3.2. Join status	35
3.3. Running join scripts	35
3.4. Writing join scripts	36
3.4.1. Basic join script example	36
3.4.2. Join script exit codes	38
3.4.3. Join script libraries	38
3.4.3.1. <i>univention-join</i>	38
3.4.3.2. <i>shell-univention-lib</i>	40
3.5. Writing unjoin scripts	43

An 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.

3.1. Join scripts

Feedback 

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 Section 3.5). The file must have the executable permission bits set.

3.2. Join status

Feedback 

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.

3.3. Running join scripts

Feedback 

There exist three commands related to running join scripts:

univention-join

When `univention-join` is invoked, a machine account is created. 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 master and backup domain controller 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.

3.4. Writing join scripts

 Feedback 

Similar to the Debian maintainer scripts (see Section B.3.5) they should be idem-potent: 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

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!*

3.4.1. Basic join script example

 Feedback 

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: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/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
VERSION=1
. /usr/share/univention-join/joinscripthelper.lib
joinscript_init

SERVICE="MyService"
```

```
eval "$(ucr shell)"

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

udm "computers/$server_role" modify "$@" \
  --dn "$ldap_hostdn" \
  --set reinstall=0 || 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
```

Please note the essential argument "\$@" when udm is invoked, which passes on the required LDAP credentials described in Section 4.4.

debian/join-template.install

The scripts need to be installed into /usr/lib/univention-install/, which is achieved by the following lines:

```
50join-template.inst usr/lib/univention-install/
50join-template-uninstall.uinst usr/lib/univention-uninstall/
```

Note that this package also installs an unjoin script.

debian/join-template.postinst

The join script should be invoked automatically on master and backup domain controller systems. On all other system roles an administrator must run the join script manually through univention-run-join-scripts.

```
#!/bin/sh

#DEBHELPER#

if [ "$1" = "configure" ]
then
  uinst=/usr/lib/univention-install/50join-template-uninstall.uinst
  [ -e "$uinst" ] && rm "$uinst"
fi

. /usr/share/univention-lib/base.sh
call_joinscript 50join-template.inst

exit 0
```

debian/control

The package uses two shell libraries, which are described in more detail in Section 3.4.3. Both packages providing them must be added as additional runtime dependencies.

The unjoin functions were added to UCS 3.1-0 only as erratum update 81¹. Because of this the minimum versions must be specified explicitly.

```
Source: join-template
Section: univention
Priority: optional
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends: debhelper (>= 7)
Standards-Version: 3.7.2

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:
http://www.univention.de/
```

3.4.2. Join script exit codes

 Feedback 

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 task to fully join the system into the domain are still pending. Some entries could not 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.

The script needs to be run again.

3.4.3. Join script libraries

 Feedback 

There exist two shell libraries, which provide functions which help in writing join scripts:

3.4.3.1. *univention-join*

 Feedback 

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

¹ <https://errata.software-univention.de/ucs/>

`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`. 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`.

`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 execute `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.

3.4.3.2. *shell-univention-lib*

 Feedback 

The package contains the shell library `/usr/share/univention-lib/base.sh`. Since package version `>= 2.0.17-1` it provides the following functions:

```
call_joinscript [--binddn bind-dn --bindpwd bind-password] [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 `bind-password` are passed on as-is.

```
call_joinscript_on_dcmaster [--binddn bind-dn --bindpwd bind-password] [XXjoin-script.inst]
```

Similar to `call_joinscript`, but also checks the system role and only executes the script on the master domain controller.

```
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 Section 3.4.3.1).

```
call_unjoinscript [--binddn bind-dn --bindpwd bind-password] [XXunjoin-script.uinst]
```

Calls the given unjoin script `unjoin-script` on master and backup domain controller systems. The file name must be relative to the directory `/usr/lib/univention-install/`. The optional LDAP credentials `bind-dn` and `bind-password` 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 file name 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 (e.g. when adding or removing extended attributes) the server should be stopped manually.

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 --bindpwd bind-password]
```

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

Example 3.1. Service registration in join script

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

```
ucs_removeServiceFromLocalhost servicename [--binddn bind-dn --bindpwd
bind-password]
```

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` are passed on as-is.

Example 3.2. Service unregistration in unjoin script

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

```
ucs_isServiceUnused servicename [--binddn bind-dn --bindpwd bind-pass-
word]
```

Returns 0 if no LDAP host object exists where the service *servicename* is registered with.

Example 3.3. Check for unused service in unjoin script

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

```
ucs_registerLDAPExtension [--binddn bind-dn { --bindpwd bind-password | --
bindpwdfile filename }
{{ --schema filename | --acl filename | --udm_syntax filename | --udm_hook file-
name ...}
| --udm_module filename [--messagecatalog filename...] [--umcregistration filename]
[--icon filename...] }
[--packagename packagename] [--packageversion packageversion] [--ucsversionstart
ucsversion] [--ucsversionend ucsversion]
```

The shell function `ucs_registerLDAPExtension` from the Univention shell function library (see Section 14.3) 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 Section 4.2)

--acl filename.acl
  Register one or more LDAP access control list (see Section 4.3)

--udm_syntax filename.py
  Register one or more UDM syntax extension (see Section 6.4)

--udm_hook filename.py
  Register one or more UDM hook (see Section 6.2.4)

--udm_module filename.py
  Register a single UDM module (see Section 6.3)
```

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.

The following options can be given multiple times, but only after the option `--udm_module`:

```
--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/.
```

`--umregistration filename.xml`

The option can be used to supply an UMC registration file (see Section 7.5.2) to make the UDM module accessible via 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).

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 master domain controller 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 filesystem. Failure conditions may occur e.g. in case the new LDAP schema extension collides with the schema currently active. The master domain controller only activates a new LDAP schema or ACL extension if the configuration check succeeded.

Note

The corresponding UDM modules are documented in Chapter 4 and Chapter 6.

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 subprocesses). 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.

Example 3.4. Extension registration in join script

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

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 \
  --umregistration /path/to/module-object.xml \
  --icon /path/to/moduleicon16x16.png \
  --icon /path/to/moduleicon50x50.png
```

```
ucs_unregisterLDAPExtension [--binddn bind-dn { --bindpwd bind-password |
--bindpwdfile 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 unregister 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.

Example 3.5. Schema unregistration in unjoin script

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

3.5. Writing unjoin scripts

 Feedback 

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 which 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 `-uninstall.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 `prerm` 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-uninstall.uinst`.

As of UCS 3.1 `.inst` and `.uinst` are not distinguishable in the *UMC Join module* by the user. Therefore it is important to use the `-uninstall` suffix to give users a visual hint. 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 unregister 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.

Writing unjoin scripts

50join-template-uninstall.uinst

This unjoin script reverts the changes of the join script from Section 3.4.1.

```
#!/bin/sh

# 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
```

debian/join-template.prerm

The unjoin script has to be copied to the join script directory before it gets removed:

```
#!/bin/sh

#DEBHELPER#

if [ "$1" = "remove" ]
then
  cp /usr/lib/univention-uninstall/50join-template-uninstall.uinst \
    /usr/lib/univention-install/
fi

exit 0
```

debian/join-template.postrm

The unjoin script should be invoked automatically on master and backup domain controller systems after the package is removed. On all other system roles an administrator must run the join script manually through `univention-run-join-scripts`.

```
#!/bin/sh
```

```
#DEBHELPER#

if [ "$1" = "remove" ]
then
  . /usr/share/univention-lib/all.sh
  call_unjoinscript 50join-template-uninstall.uinst
fi

exit 0
```

debian/join-template.postinst

In case the package is installed again and the unjoin script still exists, because it was never executed, the unjoin script must be removed:

```
#!/bin/sh

#DEBHELPER#

if [ "$1" = "configure" ]
then
  uinst=/usr/lib/univention-install/50join-template-uninstall.uinst
  [ -e "$uinst" ] && rm "$uinst"
fi

. /usr/share/univention-lib/base.sh
call_joinscript 50join-template.inst

exit 0
```


Chapter 4. Lightweight Directory Access Protocol (LDAP) in UCS

4.1. General	47
4.2. Packaging LDAP Schema Extensions	47
4.3. Packaging LDAP ACL Extensions	48
4.4. LDAP secrets	50
4.4.1. Password change	50

4.1. General

Feedback 

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 sources cover this topic exhaustively, e.g. <http://www.zytrax.com/books/ldap/> or the man pages (`slapd.conf`, `slapd.access`).

At least it should be noted that OpenLDAP offers two fundamentally different tool sets for direct access or modification of LDAP data: 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. 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 Chapter 6). That is the level that 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 builtin mechanism for automatic schema replication to all UCS hosted LDAP servers in the UCS domain (see Chapter 5). The schema replication mechanism is triggered by installation of a new schema extension package on the UCS master. For redundancy it is strongly recommended to install schema extension packages also on the UCS backup systems. This way, a UCS backup can replace a UCS master in case the master 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.

4.2. Packaging LDAP Schema Extensions

Feedback 

For some purposes, e.g. 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 master domain controller and backup domain controller systems listen for modifications of these objects and integrate them into the local LDAP schema directory. As noted above, this simplifies the task of keeping the schema on the backup domain controller systems up to date with that on the master domain controller.

The commands to create the LDAP schema extension objects in UDM may be put into any join script (see Chapter 3). A LDAP schema extension object is created by using the UDM command line interface `univen-`

tion-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 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 master domain controller and backup domain controller. The file name 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.

`appididentifier` (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 master domain controller to signal availability of the schema extension (default: `FALSE`).

Since many of these parameters are determined automatically by the `ucs_registerLDAPExtension` shell library function, it is recommended to use the shell library function to create these objects (see Section 9.10.1).

Example 4.1. Schema registration in join script

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

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

4.3. Packaging LDAP ACL Extensions

Feedback 

For some purposes, e.g. 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/ldap-acl`. As these objects are replicated throughout the UCS domain, the master domain controller, backup domain controller and slave domain controller 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 domain controller systems up to date with those on the master domain controller.

The commands to create the LDAP ACL extension objects in UDM may be put into any join script (see Chapter 3). 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 file name the ACL UCR template data should be written to on master domain controller, backup domain controller and slave domain controller. The file name 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.

`appidifier` (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 master domain controller to signal availability of the ACL extension on the master domain controller (default: `FALSE`).

Since many of these parameters are determined automatically by the `ucs_registerLDAPExtension` shell library function, it is recommended to use the shell library function to create these objects (see Section 9.10.1).

Example 4.2. 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
```

4.4. LDAP secrets

 Feedback 

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 master and backup domain controller 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. 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 Section B.3.5) on master and backup domain controller 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.

4.4.1. Password change

 Feedback 

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`. 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(8)`. The interval is controlled through a second Univention Configuration Registry variable `server/password/interval`, 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 via `run-parts(8)`. It receives one argument, which is used to distinguish three phases:

Procedure 4.1. Server password change procedure

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(1)`. It is changed in the Univention directory service via 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.

Example 4.3. Server password change example

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

```
#!/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
  invoke-rc.d my-daemon restart
  ;;
esac
```

`init-scripts` should only be invoked indirectly through `invoke-rc.d(8)`. 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.

Chapter 5. Univention Directory Listener

5.1. Structure of Listener Modules	53
5.2. Listener Tasks and Examples	56
5.2.1. Basic Example	56
5.2.2. Rename and Move	57
5.2.3. Full Example with Packaging	58
5.2.4. A Little Bit more Object Oriented	62
5.3. Technical Details	65
5.3.1. User-ID and Credentials	65
5.3.2. Internal Cache	65
5.3.2.1. univention-directory-listener-ctrl	65
5.3.2.2. univention-directory-listener-dump	65
5.3.2.3. univention-directory-listener-verify	66
5.3.2.4. get_notifier_id.py	66
5.3.3. Internal working	66

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 master domain controller (and possibly existing backup domain controller 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 master domain controller (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.

5.1. Structure of Listener Modules

 Feedback 

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.

Each listener module must declare several string constants. They are required by the Univention Directory Listener to handle each module. They should be defined at the beginning of the module.

```
name = "module_name"
description = "Module description"
```

```
filter = "(objectClass=*)"
attribute = ["objectClass"]
modrdn = "1"
```

name (required)

This name is used to uniquely identify the module. This should match with 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/`.

description (required)

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

filter (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, but more restricted:

- it is case sensitive
- it only supports equal matches

Note

The name `filter` has the drawback that it shadows the Python built-in function `filter()`, 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 via the Python `__builtin__` module.

attributes (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.

modrdn (optional)

Setting this variable to the string `1` changes the signature of the function `handler()`. It receives an additional 4th argument, which specifies the LDAP operation triggering the change.

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

```
def initialize(): pass
def handler(dn, new, old[, command='']): pass
def clean(): pass
def prerun(): pass
def postrun(): pass
def setdata(key, value): pass
```

handler(dn, old, new, command='') (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 (master domain controller or backup domain controller servers in the domain).

<http://tools.ietf.org/html/rfc2254>

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 type of modification. This can be used to 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 cached values and `new` contains the new values as retrieved from the LDAP directory service.

`a` (add)

Signals the addition of a new object. `old` is `None` and `new` contains the 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 via `modrdn`)

Signals a change in the distinguished name, which may be caused by renaming an 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 initialization of a new object, which should be handled just like a normal `add` operation.

`initialize()` (optional),

`clean()` (optional)

The function `initialize()` 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 command `univention-directory-listener-ctrl resync name`. In that case the function `initialize()` will be called again.

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

`prerun()` (optional),

`postrun()` (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()` and the closing by `postrun()`.

The function `postrun()` 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()` function.

Warning

The function `postrun()` 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!

`setdata(key, value)` (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 (via `simple bind`).

`bindpw`

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

`ldapserver`

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.

5.2. Listener Tasks and Examples

 Feedback 

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

5.2.1. Basic Example

 Feedback 

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: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/listener/simple.py>

```
def handler(dn, new, old):
    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, new):
    """Handle addition of object."""
    pass # replace this
```

```
def handler_modify(dn, old, new):
    """Handle modification of object."""
    pass # replace this

def handler_remove(dn, old):
    """Handle removal of object."""
    pass # replace this
```

5.2.2. Rename and Move

Feedback 

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

Source code: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/listener/modrdrn.py>

```
modrdrn = "1"

_delay = None

def handler(dn, new, old, command):
    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_move(old_dn, old, new_dn, dn):
    """Handle rename or move of object."""
    pass # replace this

def handler_schema(old, new):
    """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.

5.2.3. Full Example with Packaging

 Feedback 

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

Source code: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/listener/printusers/>

```
"""
Example for a listener module, which logs changes to users.
"""
__package__ = "" # workaround for PEP 366
import listener
import os
import errno
import univention.debug as ud
from collections import namedtuple

name = 'printusers'
description = 'print all names/users/uidNumbers into a file'
filter = """\
(&
 (|
  (&
   (objectClass=posixAccount)
   (objectClass=shadowAccount)
  )
 (objectClass=univentionMail)
 (objectClass=sambaSamAccount)
 (objectClass=simpleSecurityObject)
 (objectClass=inetOrgPerson)
 )
 (!(objectClass=univentionHost))
 (!(uidNumber=0))
 (!(uid=*$))
)""".translate(None, '\t\n\r')
attributes = ['uid', 'uidNumber', 'cn']
_Rec = namedtuple('Rec', ' '.join(attributes))

USER_LIST = '/root/UserList.txt'

def handler(dn, new, old):
    """
    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, new, old):
    """
    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, None)

def _handle_add(dn, new):
    """
    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, old):
    """
    Called when an previously existing object is removed.
    """
    o_rec = _rec(old)
    ud.debug(ud.LISTENER, ud.INFO, 'Removed user "%s"' % (o_rec.uid,))
    _writeit(o_rec, u'removed')

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

class AsRoot(object):
    """
    Temporarily change effective UID to 'root'.
    """
    def __enter__(self):
        listener.setuid(0)
    def __exit__(self, exc_type, exc_value, traceback):
```

```

    listener.unsetuid()

def _writeit(rec, comment):
    """
    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 AsRoot():
            with open(USER_LIST, 'a') as out:
                print >> out, u'%sName: "%s"' % (indent, rec.cn)
                print >> out, u'%sUser: "%s"' % (indent, rec.uid)
                print >> out, u'%sUID: "%s"' % (indent, nuid)
                if comment:
                    print >> out, u'%s%s' % (indent, comment,)
    except IOError as ex:
        ud.debug(
            ud.LISTENER, ud.ERROR,
            'Failed to write "%s": %s' % (USER_LIST, ex))

def initialize():
    """
    Remove the log file.
    This function is called when the module is forcefully reset.
    """
    try:
        with AsRoot():
            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:

- Overwriting `__package__` is currently necessary, as the Univention Directory Listener imports the listener module by its own mechanism, which is incompatible with the mechanism normally used by Python itself. Be aware, that this might cause problems when using *pickle*.
- 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)
    invoke-rc.d univention-directory-listener restart
    ;;
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)
    invoke-rc.d univention-directory-listener restart
    ;;
purge|upgrade|failed-upgrade|abort-install|abort-upgrade|disappear)
    ;;
*)
    echo "postrm called with unknown argument \`$1'" >&2
    exit 1
    ;;
esac

#DEBHELPER#

exit 0
```

5.2.4. 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: <https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/doc/developer-reference/listener/obj.py>

```

__package__ = "" # workaround for PEP 366
name = "refcheck"
description = "Check referential integrity of uniqueMember relations"
filter = "(uniqueMember=*)"
attribute = ["uniqueMember"]
modrdn = "1"

import os
import ldap
import listener
import univention.debug as ud
from pwd import getpwnam

class LocalLdap(object):
    PORT = 7389

    def __init__(self):
        self.data = {}
        self.con = None

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

    def prerun(self):
        try:
            self.con = ldap.open(self.data["ldapservers"], port=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):
        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):
        try:
            ent = getpwnam(self.USER)
            with AsRoot():

```

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

def log(self, msg):
    with open(self.LOG, 'ab') as log:
        print >> log, msg

def clean(self):
    try:
        with AsRoot():
            os.remove(self.LOG)
    except OSError as ex:
        ud.debug(ud.LISTENER, ud.ERROR, str(ex))

class AsRoot(object):
    """
    Temporarily change effective UID to 'root'.
    """
    def __enter__(self):
        listener.setuid(0)
    def __exit__(self, exc_type, exc_value, traceback):
        listener.unsetuid()

class ReferentialIntegrityCheck(LocalLdap, LocalFile):
    MESSAGES = {
        (False, False): "Still invalid: ",
        (False, True): "Now valid: ",
        (True, False): "Now invalid: ",
        (True, True): "Still valid: ",
    }
    def __init__(self):
        super(ReferentialIntegrityCheck, self).__init__()
        self._delay = None
    def handler(self, dn, new, old, command=''):
        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()
        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)
    else:
        pass # ignore, reserved for future use

def handler_add(self, dn, new):
    if not self._validate(new):
        self.log("New invalid object: " + dn)

def handler_modify(self, dn, old, new):
    valid = (self._validate(old), self._validate(new))
    msg = self.MESSAGES[valid]
    self.log(msg + dn)

def handler_remove(self, dn, old):
    if not self._validate(old):
        self.log("Removed invalid: " + dn)

def handler_move(self, old_dn, old, new_dn, new):
    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):
    self.log("Schema change")

def _validate(self, data):
    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
clean = _instance.clean
prerun = _instance.prerun
postrun = _instance.postrun
setdata = _instance.setdata
```

5.3. Technical Details

Feedback 

5.3.1. User-ID and Credentials

Feedback 

The listener runs with the effective permissions of the user `listener`. If root-privileges are required, `listener.setuid()` can be used to switch the effective UID. When done, `listener.unsetuid()` should be called to drop back to the `listener` UID. It's best practice to code this as `try/finally` clauses in Python.

5.3.2. Internal Cache

Feedback 

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

`cache.db`, `cache.db.lock`

The cache file 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 `handle()` 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 member server do not 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.

`notifier_id`

This 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()`. 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.

5.3.2.1. univention-directory-listener-ctrl

Feedback 

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()` and `initialize()` to be called one after the other.

5.3.2.2. univention-directory-listener-dump

Feedback 

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 `service univention-directory-listener stop`. It outputs the cache in format compatible to the LDAP Data Interchange Format (LDIF).

5.3.2.3. univention-directory-listener-verify

Feedback 

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 `service univention-directory-listener stop`. 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)" \
-w "$(cat /etc/machine.secret)"
```

5.3.2.4. get_notifier_id.py

Feedback 

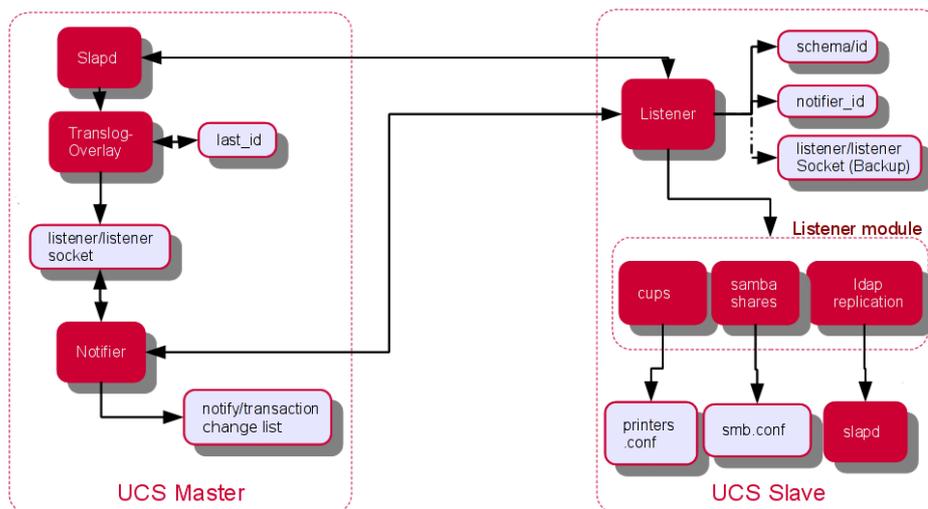
The command `/usr/share/univention-directory-listener/get_notifier_id.py` can be used to get the latest ID from the notifier. This is done by querying the Univention Directory Notifier running on the LDAP server configured through the Univention Configuration Registry variable `ldap/master`. The returned 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

5.3.3. Internal working

Feedback 

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.

Figure 5.1. Listener/Notifier mechanism



On startup the listener connects to the notifier and opens a persistent TCP connection. 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 master domain controller and on all backup domain controllers, only the host named in `ldap/master` is used.
- Otherwise on all other system roles a host is chosen randomly from the combined list of names in `ldap/master` and `ldap/backup`¹.

The following steps occur on changes:

Procedure 5.1. Listener/Notifier procedure

1. An LDAP object is modified on the master domain controller. Changes initiated on all other system roles are re-directed to the master.
2. The UCS-specific overlay-module `translog` appends the DN to the file `/var/lib/univention-ldap/listener/listener`².
3. The Univention Directory Notifier watches that file, picks up and removes each line it processed. It assigns the next transaction number and writes it into the file `/var/lib/univention-ldap/notify/transaction`³, including the DN and change type. For efficient access by transaction ID the index `transaction.index` is updated.
4. All listeners get notified of the new transaction.
5. Each listener triggered in this way queries the Notifier for the latest transaction ID, DN and change type.
6. Each listener opens a connection to the LDAP server running on the UCS system which was used to query the Notifier. 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.
7. The old state of the object is fetched from the local listener cache.
8. 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.
9. If the function `prerun()` of module was not called yet, this is done to signal the start of changes.
10. The function `handler()` specified in the module is called, passing in the DN and the old and new state.
11. 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.
12. On a backup domain controller the Univention Directory Listener writes the transaction data to the file `/var/lib/univention-ldap/listener/listener`⁴ 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.
13. The transaction ID is written into the local file `/var/lib/univention-directory-listener/notifier_id`.
14. After 15 seconds of inactivity the function `postrun()` 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.

¹This list of backup domain controllers stored in the Univention Configuration Registry variable `ldap/backup` is automatically updated by the listener module `ldap_server.py`.

²Referred to as `FILE_NAME_LISTENER`, `TRANSACTION_FILE` in the source code

³Referred to as `FILE_NAME_TF` in the source code

⁴Referred to as `FILE_NAME_LISTENER`, `TRANSACTION_FILE` in the source code

Chapter 6. Univention Directory Manager (UDM)

6.1. Introduction	69
6.2. Packaging Extended Attributes	70
6.2.1. Selection lists	74
6.2.1.1. Static selections	74
6.2.1.2. Dynamic selections	74
6.2.2. Known issues	76
6.2.3. Extended Options	76
6.2.4. Extended Attribute Hooks	77
6.3. UDM Modules	79
6.4. UDM Syntax	79
6.4.1. UDM Syntax Override	80
6.4.2. UDM LDAP search	81
6.5. Packaging UDM Hooks	84
6.6. Packaging UDM Extension Modules	85
6.7. Packaging UDM Syntax Extension	86

The Univention Directory Manager (UDM) is a wrapper for LDAP objects. Traditionally LDAP stores objects as a collection of attributes, which are defines 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.

6.1. Introduction

Feedback 

The command line client `udm` provides different modes of operation.

```
udm [--binddn bind-dn --bindpwd bind-password] [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 multivalued property.

The equivalent LDAP command would look like this:

```
eval "$(ucr shell)"
ldapadd -D "cn=admin,$ldap_base" -y /etc/ldap.secret <<__LDIF__
dn: uid=xxx,$ldap_base
objectClass: organizationalRole
cn: xxx
__LDIF__
```

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 multivalued 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.

6.2. Packaging Extended Attributes

 Feedback 

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 master domain controller, which is replicated from there to all other Domaincontrollers via the Listener/Notifier mechanism (see Chapter 5). This is best done through a separate schema package, which should be installed on the master domain controller and backup domain controller. Since Extended Attributes are declared in LDAP, the commands to create them can be put into any join script (see Chapter 3). 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 ????

`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. It is also used when for an object if the option is enabled, which only then activates the property.
- `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.
- `hook` (optional)
The name of a Python class implementing hook functions. See Section 6.2.4 for more information.
- `doNotSearch` (optional)
If this is enabled, the property is not shown 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. If no name is given, ???
- `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.

Packaging Extended Attributes

`overwriteTab` (optional)

If enabled, the tab declared by the UDM module with the name from the `tabName` settings is replaced 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.

`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.

```

#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 univentionExamplesUdm 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
)
#!/bin/sh

#DEBHELPER#

. /usr/share/univention-lib/base.sh
#!/bin/bash
VERSION=1
. /usr/share/univention-join/joinscripthelper.lib
joinscript_init

# 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" \
  --set translationShortDescription='"de_DE" "Beispielattribut"' \
  --set longDescription="An example attribute" \
  --set translationLongDescription='"de_DE" "Ein Beispielattribut"' \
  --set tabAdvanced=1 \
  --set tabName="Examples" \
  --set translationTabName='"de_DE" "Beispiele"' \
  --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
This example is deliberately missing an unjoin script (see Section 3.5) to keep this example simple. It should
check if the Extended Attribute is no longer used in the domain and then remove it.

joinscript_save_current_version
exit 0

```

6.2.1. Selection lists

Feedback 

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.

6.2.1.1. Static selections

Feedback 

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_registerLDAPSchema` which is described in Section 3.4.3.2.

As an alternative the file can be put into the directory `/usr/share/pyshared/univention/admin/syntax.d/` and linked into the directory `/usr/lib/pymodules/python2.6/univention/admin/syntax.d/`. When included into a Debian package, the linking is normally done by `dh_python`.

The following example is comparable to the default example in file `/usr/share/pyshared/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'),
    ]
```

6.2.1.2. Dynamic selections

Feedback 

A dynamic list is implemented as an LDAP search, which is described in Section 6.4.2. For performance reason it is recommended to implement a class derived from `UDM_Attribute` or `UDM_Objects` instead of using `LDAP_Search`. The file `/usr/share/pyshared/univention/admin/syntax.py` contains several examples.

Example 6.2. Dynamic selection list for Extended Attributes

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.

```
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" \
  --set data="$(bzip2 <DynamicSelection.py | base64)" \
  --set package="$syntax" --set packageversion="1"

udm settings/extended_attribute create --position "cn=custom attributes,
$base" \
  --set name='Profession' \
  --set module='users/user' \
  --set tabName='General' \
  --set translationTabName='"de_DE" "Allgemein"' \
  --set groupName='Personal information' \
  --set translationGroupName='"de_DE" "Persönliche Informationen"' \
  --set shortDescription='UCS profession level' \
  --set translationShortDescription='"de_DE" "UCS Erfahrung"' \
  --set longDescription='Select a level of UCS experience' \
  --set translationLongDescription='"de_DE" "Wählen Sie den Level der
Erfahrung mit UCS"' \
  --set objectClass='univentionFreeAttributes' \
  --set ldapMapping='univentionFreeAttributel' \
  --set syntax="$syntax" --set mayChange=1 --set valueRequired=0
```

6.2.2. Known issues

 Feedback 

- 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/share/pyshared/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.

6.2.3. Extended Options

 Feedback 

UDM properties can be enabled and disabled via *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.

Example 6.3. 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
```

6.2.4. Extended Attribute Hooks

 Feedback 

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/share/pyshared/univention/admin/hooks.d/`¹. The file name must end with `.PY`.

The module `univention.admin.hook` 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.

```
hook_open(self,
```

```
module);
```

This method is called by the default open handler just before the current state of all properties is saved.

```
hook_ldap_pre_create(self,
```

```
module);
```

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.

```
list hook_ldap_addlist(self,
```

```
module,
al= []);
```

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` will also be called next.

```
hook_ldap_post_create(self,
```

```
module);
```

This method is called after the object was created in LDAP.

¹ This assumes that the hook file is packaged and linked by `dh_pysupport` to `/usr/lib/pymodules/python2.6/univention/admin/hooks.d/` for Python 2.6 or whatever Python version is used. If the file is installed manually, it must be placed on a path listed in `sys.path`.

```
hook_ldap_pre_modify(self,
                      module);
```

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.

```
list hook_ldap_modlist(self,
                       module,
                       ml= []);
```

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_post_modify(self,
                      module);
```

This method is called after the object was modified in LDAP.

```
hook_ldap_pre_remove(self,
                     module);
```

This method is called before a UDM object is removed.

```
hook_ldap_post_remove(self,
                      module);
```

This method is called after the object was removed from LDAP.

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'

    def hook_ldap_post_modify(self, module):
        """Remove unused objectClass."""
        ext_attr_name = 'isSampleUser'
        class_name = 'univentionFreeAttributes'

        if module.oldinfo.get(ext_attr_name) in ('1',) and \
            module.info.get(ext_attr_name) in ('0', None):
            if class_name in module.oldattr.get('objectClass', []):
                module.lo.modify(module.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
```

6.3. UDM Modules

Feedback 

The development of Univention Directory Manager modules is currently only documented in Univention Wiki (currently only available in German): http://wiki.univention.de/index.php?title=Entwicklung_und_Integration_eigener_Module_in_Univention_Directory_Manager

6.4. UDM Syntax

Feedback 

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/share/pyshared/univention/admin/syntax.py`. The following list of syntaxes is not complete, for a complete overview the file should be consulted 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 Section 6.4.2.

Additional syntax classes can be added by placing a Python file in `/usr/share/pyshared/univention/admin/syntax.d/`. They're automatically imported by UDM.

6.4.1. UDM Syntax Override

Feedback 

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.

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 (see ????) or by setting the variable in the `.postinst` script of some package, which is installed on all hosts.

6.4.2. UDM LDAP search

It is often required to present a list of entries to the user, from which she 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/share/pyshared/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.

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 `attribute$` is replaced by the value of variable `attribute` declared above.

regex

This defines an optional regular expression, which is used in the frontend 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 DelegateTelephoneNumber(UDM_Attribute):
    udm_module = 'users/user'
    attribute = 'phone'
    label_format = '%(displayName)s: %($attribute$s)'
```

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 `multivalued` 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`.

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'
```

LDAP_Search

This is the old implementation, which should only be used, if `UDM_Attribute` and `UDM_Objects` 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` and providing that implementation in `/usr/share/pyshared/univention/admin/syntax.d/`.

UDM API

By creating a UDM object in LDAP using the module `settings/syntax`.

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
```

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, multivalued),
 ldapattribute (optional, multivalued)
 The name of UDM properties, which are display as a label to the user. Alternatively LDAP attribute names may be used directly.

value (optional),
 ldapvalue (optional)
 The name of the UDM property, which is used to reference the object. Alternatively an LDAP attribute name may be used directly.

viewonly (optional)
 If set to 1 the values can not be changed.

addEmptyValue (optional)
 If set to 1 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'
```

6.5. Packaging UDM Hooks

 Feedback 

For some purposes, e.g. 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 Chapter 3). 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 Section 9.10.1).

name (required)
 Name of the UDM hook.

`data` (required)

The actual UDM hook data in bzip2 and base64 encoded format.

`filename` (required)

The file name the UDM hook data should be written to by the listening servers. The file name 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 4.1-x release range a value like 4.1-99 may be used.

`active` (internal)

A boolean flag used internally by the master domain controller to signal availability of the new UDM hook on the master domain controller (default: `FALSE`).

6.6. Packaging UDM Extension Modules

Feedback 

For some purposes, e.g. 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 Chapter 3). 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 Section 9.10.1).

`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 file name the UDM module data should be written to by the listening servers. The file name may contain path elements and should conform to the name of the UDM module (e.g. `newapp/someobject.py`).

messagecatalog (optional)

Multivalued property to supply message translation files (syntax: `<language tag> <base64 encoded GNU message catalog>`).

umcregistration (optional)

XML definition required to make the UDM module available through the Univention Management Console (base64 encoded XML)

icon (optional)

Multivalued 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 4.1-x release range a value like 4.1-99 may be used.

active (internal)

A boolean flag used internally by the master domain controller to signal availability of the new UDM module on the master domain controller (default: `FALSE`).

6.7. Packaging UDM Syntax Extension

Feedback 

For some purposes, e.g. 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 Chapter 3). 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 Section 9.10.1).

`name` (required)

Name of the UDM syntax.

`data` (required)

The actual UDM syntax data in bzip2 and base64 encoded format.

`filename` (required)

The file name the UDM syntax data should be written to by the listening servers. The file name 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 4.1-x release range a value like 4.1-99 may be used.

`active` (internal)

A boolean flag used internally by the master domain controller to signal availability of the new UDM syntax on the master domain controller (default: `FALSE`).

Chapter 7. Univention Management Console (UMC)

7.1. Architecture	89
7.2. Asynchronous Framework	90
7.3. Protocol UMCP 2.0	91
7.3.1. Data flow	91
7.3.2. Authentication	91
7.3.3. Message format	91
7.3.3.1. Message header	91
7.3.3.2. Message body	92
7.3.4. Examples	92
7.4. Protocol HTTP for UMC	93
7.4.1. Examples	93
7.5. UMC files	94
7.5.1. <code>debian/package.umc-modules</code>	94
7.5.2. UMC Module Declaration File	95
7.6. Local System Module	95
7.6.1. Python API	95
7.6.2. UMC module API (Python and JavaScript)	95
7.6.2.1. XML definition	96
7.6.2.2. Python module	97
7.6.2.3. UMC store API	99
7.6.3. Packaging	100
7.7. Domain LDAP Module	102
7.8. Disabling a Module	103

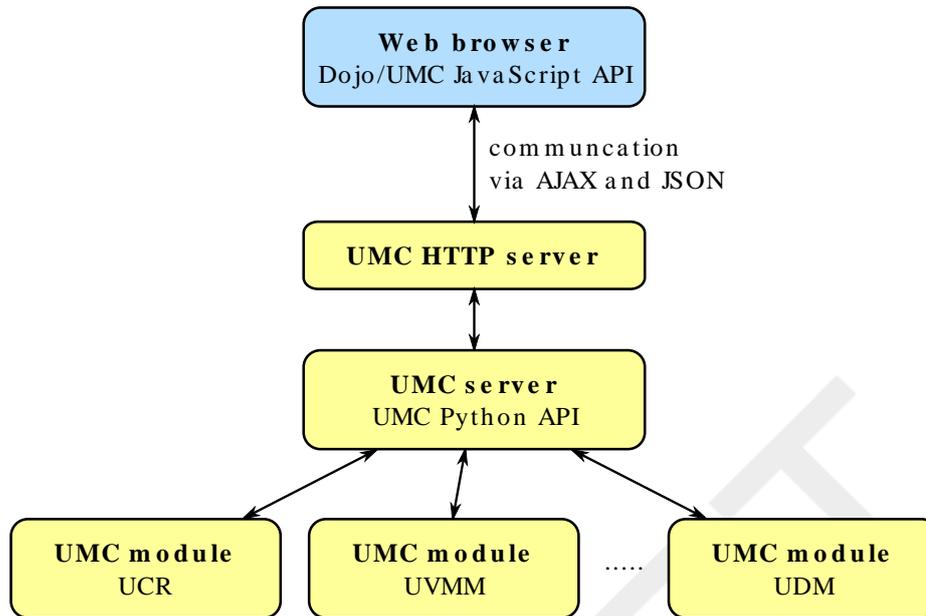
The Univention Management Console (UMC) is a service that runs on all UCS systems by default. This service provides access to several system information and implements modules for management tasks. What modules are available on a UCS system depends on the system role and the installed components. Each domain user can log on to the service via 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.

7.1. Architecture

Feedback 

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 Figure 7.1.

Figure 7.1. UMC architecture and communication channels

- The *UMC server* is the core component. It provides access to the modules and manages the connection and verifies that only authorized users gets access. The protocol used to communicate is the *Univention Management Console Protocol* (UMCP) in version 2.0.
- The *UMC HTTP server* is a small web server that provides HTTP access to the UMC server. 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.

7.2. Asynchronous Framework

Feedback

All server-side components of the UMC service are based on the asynchronous framework Python Notifier, that provides techniques for handling quasi parallel tasks based on events. The framework follows three basic concepts:

Non-blocking sockets

For servers that should handling several communication channels at a time have to use so called non-blocking sockets. This is an option that needs to be set for each socket, that should be management by the server. This is necessary to avoid blocking on read or write operations on the sockets.

Timer

To perform tasks after a defined amount of time the framework provides an API to manage timer (one shot or periodically).

Signals

To inform components within a process of a specific a events the framework provide the possibility to define signals. Components being interested in events may place a registration.

Further details, examples and a complete API documentation for Python Notifier can be found at the website of Python Notifier¹.

¹ <https://github.com/crunchy-github/python-notifier>

7.3. Protocol UMCP 2.0

Feedback 

This protocol is used by the UMC server for external clients and between the UMC server and its UMC module processes.

7.3.1. Data flow

Feedback 

The protocol is based on a server/client model. The client sends requests to the server that will be answered with a response message by the server.

With a status code in the response message the client can determine the type of result of its request:

- An error occurred during the processing of the request. The status code contains details of the error.
- The command was processed successfully. A status message may contain details about the performed task.

7.3.2. Authentication

Feedback 

Before a client may send request messages to the server that contain commands to execute, the client has to authenticate. After a successful authentication the UMC server determines the permissions for the user defined by policies in the LDAP directory. If the LDAP server is not reachable a local cache is checked for previously discovered permissions. If none of these sources is available the user is prohibited to use any command.

The authentication process within the UMC server uses the PAM service `univention-management-console`. By default, this service uses a cache for credentials if the LDAP server is not available to provide the possibility to access the UMC server also in case of problems with the LDAP server.

7.3.3. Message format

Feedback 

The messages, request and response, have the same format that consists of a single header line, one empty line and the body.

The header line contains control information that allows the UMC server to verify the correctness of the message without reading the rest of the message.

7.3.3.1. Message header

Feedback 

The header defines the message type, a unique identifier, the length of the message body in bytes, the command and the mime type of the body.

```
(REQUEST|RESPONSE)/<id>/<length of body>[/<mime-type>]:
<command>[ <arguments>]
```

By the first keyword the message type is defined. Supported message types are `REQUEST` and `RESPONSE`. Any other type will be ignored.

Separated by a / the message id follows, that must be unique within a communication channel. By default it consists of a timestamp and a counter.

The next field is a number, defining the length of the body in bytes, starting to count after the empty line.

Since UMCP 2.0 there is as another field specifying the mime type of the body. If not given then the guessed value for the mime type is `application/json`. If the body can not be decoded using a JSON parser the message is invalid.

The last two fields define the UMCP command that should be executed by the server. The following commands are supported:

AUTH

This command sends an authentication request. It must be the first command sent by the client. All commands sent before a successful authentication are rejected.

GET

This command is used to retrieve information from the UMC server, e.g. a list of all UMC modules available in this session.

SET

This command is used to define settings for the session, e.g. the language.

COMMAND

This command is used to pass requests to UMC modules. Each module defines a set of commands, that it implements. The UMC module command is defined by the first argument in the UMCP header, e.g. a request like `REQUEST/123423423-01/42/application/json: COMMAND ucr/query` passes on the module command `ucr/query` to a UMC module.

7.3.3.2. Message body

Feedback 

The message body may contain one object of any type, e.g. an image, an OpenOffice document or a JSON object. The JSON object is the default type and is the only supported mime type for request messages. It contains a dictionary that has a few predefined keys (for both message types):

options

Contains the arguments for the command.

status

Defines the status code in response messages. The codes are similar to the HTTP status codes, e.g. 200 defines a successful execution of the command.

message

May contain a human readable description of the status code. This may contain details to explain the user the situation.

flavor

An optional field. If given in a request message the module may act differently than without the flavor.

7.3.4. Examples

Feedback 

This section contains a few example messages of UMCP 2.0.

Example 7.1. Authentication request

```
REQUEST/130928961341733-1/147/application/json: AUTH
{"username": "root", "password": "univention"}
```

Example 7.2. Search for users

Request:

```
REQUEST/130928961341726-0/125/application/json: COMMAND udm/query
{
  "flavor": "users/user",
  "options": {
    "objectProperty": "name",
    "objectPropertyValue": "test1*1",
    "objectType": "users/user"
  }
}
```

Response:

```
RESPONSE/130928961341726-0/1639/application/json: COMMAND udm/query
{
  "status": 200,
  "message": null,
  "options": {
    "objectProperty": "name",
    "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"
    }
  ]
}
```

7.4. Protocol HTTP for UMC

Feedback 

With the new generation of UMC there is also an HTTP server available that can be used to access the UMC server. The web server is implemented as a frontend to the UMC server and translates HTTP POST requests to UMCP commands.

7.4.1. Examples

Feedback 

Example 7.3. Authentication request

```
POST http://10.200.15.31/umcp/auth HTTP/1.1
{"options": {"username": "root", "password": "univention"}}
```

Example 7.4. search for users

Request

```
POST http://10.200.15.31/umcp/command/udm/query HTTP/1.1

{"options": {"container": "all",
             "objectType": "users/user",
             "objectProperty": "username",
             "objectPropertyValue": "test1*1"},
 "flavor": "users/user"}
```

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"}]}
```

7.5. UMC files

Feedback 

Files for building a UMC module.

7.5.1. `debian/package.umc-modules`

Feedback 

`dh-umc-module-build` builds translation files. `dh-umc-module-install` installs files. Configured through `debian/package.umc-modules`.

```
Module: module-name
Python: umc
Definition: umc/module-name.xml
Javascript: umc
Icons: umc/icons
```

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 Section 7.5.2 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.

7.5.2. UMC Module Declaration File

Feedback 

`umc/module.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!--DOCTYPE umc SYSTEM "branches/ucs-4.1/ucs-4.1-0/management/
univention-management-console/doc/module.dtd"-->
<umc version="2.0">
  <module id="udm" icon="udm-MODULE" version="1.0"
translationId="MODULE">
    <name>...</name>
    <description>...</description>
    <flavor>...</flavor>
    <categories>
      <category name="domain"/>
    </categories>
    <command>...</command>
  </module>
</umc>
```

`umc/categories/category.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<umc version="2.0">
  <categories>
    <category id="category" priority="..." icon="....svg" color="#xxxxxx"/>
  >
</categories>
</umc>
```

7.6. Local System Module

Feedback 

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 Section 7.1.

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.

7.6.1. Python API

Feedback 

The Python API for the UMCP is defined in the python module `univention.management.console.protocol`.

7.6.2. UMC module API (Python and JavaScript)

Feedback 

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.

7.6.2.1. XML definition

Feedback 

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 UMCP 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

7.6.2.1.1. Module definition

Feedback 

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"?>
<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 Section 7.6.3.

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 Section 7.6.3.

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 identify 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 UMCP message it is the first argument after the UMCP COMMAND.

`function`

defines the method to be invoked within the python module when the command is called.

7.6.2.1.2. Category definition

Feedback 

The predefined set of categories can be extended by each module.

Example 7.5. UMC module category examples

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

7.6.2.2. Python module

Feedback 

The Python API for UMC modules primary consists of one base class that must be implemented. As an addition to python API provides some helper:

- exception classes
- translation support
- logging functions
- UCR access

In the definition file for 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:

```

from univention.management.console import Translation
from univention.management.console.base import Base, UMC_Error
from univention.management.console.log import MODULE
from univention.management.console.config import ucr
from univention.management.console.modules.sanitizers import
    IntegerSanitizer
from univention.management.console.modules.decorators import sanitize

_ = Translation('univention-management-console-modules-udm').translate

class Instance(Base):

    def init(self):
        """Initialize the module with some values"""
        super(Instance, self).init()
        self.data = [int(x) for x in ucr.get('some/examle/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 UMCP 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 e.g. be used to initialize a database connection. The other methods in the example will serve spe-

cific request. 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.

7.6.2.3. UMC store API

Feedback 

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 API². This allows the JavaScript code to easily 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.

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

² <http://dojotoolkit.org/reference-guide/dojo/store.html>

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*.

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 guide³
- Object store tutorial⁴
- HTML5 IndexedDB object store API⁵

7.6.3. Packaging

Feedback 

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`
- `univention-management-console-module-ucr/debian/compat`

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:

```
Module: ucr
Python: umc/python
Definition: umc/ucr.xml
Syntax: umc/syntax/ucr.xml
```

³ <http://dojotoolkit.org/reference-guide/dojo/store.html>

⁴ <http://www.sitepen.com/blog/2011/02/15/dojo-object-stores/>

⁵ <http://www.w3.org/TR/IndexedDB/#object-store>

```
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/compat

debian/rules

```
%:
dh $@

override_dh_auto_build:
dh-umc-module-build
dh_auto_build

override_dh_auto_install:
dh-umc-module-install
dh_auto_install
```

debian/control

```
Source: univention-management-console-module-ucr
Section: univention
Priority: optional
Maintainer: Univention GmbH <packages@univention.de>
Build-Depends: debhelper (>= 7.0.50~),
  python-support,
  univention-management-console-dev,
  python-all
Standards-Version: 3.5.2
XS-Python-Version: all

Package: univention-management-console-module-ucr
Architecture: all
Depends: univention-management-console-server
Description: UMC module for UCR
  This package contains the UMC module for Univention Configuration
  Registry
```

7.7. Domain LDAP Module

Feedback 

Done through *flavor*.

```
<?xml version="1.0" encoding="UTF-8"?>
<umc version="2.0">
  <module id="udm" icon="udm-MODULE" version="1.0"
  translationId="MODULE">
    <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`.

7.8. Disabling a Module

To disabling a module use the following XML file as a template:

```
<?xml version="1.0" encoding="UTF-8"?>
<umc version="2.0">
  <module id="udm" icon="udm/module" version="1.0"
  translationId="MODULE">
    <name/>
    <description/>
    <flavor id="MODULE/SUBMODULE" deactivated="yes" />
  </module>
</umc>
```

DRAFT

DRAFT

Chapter 8. Web services

8.1. Extending the overview page 105

8.1. Extending the overview page

Feedback 

When a user opens `http://localhost/` or `http://hostname/` in a browser, she is 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; e.g. `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; e.g. `description/de` can be used for a description in German. Should not exceed 60 characters, because of space limitations of the rendered box.
- 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`, i.e. with a leading `/`. All file formats typically displayed by browsers can be used (e.g. PNG/JPG). All icons must be scaled to 50x50 pixels.
- The display order can be specified using `priority`. Depending on the values the entries are displayed in *lexicographical* order (i.e. $100 < 50$).

The following example configures the link to the Nagios web interface:

```
ucs/web/overview/entries/admin/nagios/description/de: Netzwerk-, Host-  
und Serviceüberwachung  
ucs/web/overview/entries/admin/nagios/description: Network, host and  
service monitoring system  
ucs/web/overview/entries/admin/nagios/icon: /icon/50x50/nagios.png  
ucs/web/overview/entries/admin/nagios/label/de: Univention Nagios  
ucs/web/overview/entries/admin/nagios/label: Univention Nagios  
ucs/web/overview/entries/admin/nagios/link: /nagios/  
ucs/web/overview/entries/admin/nagios/priority: 50
```


Chapter 9. App Center

9.1. Requirements	107
9.2. Packaging for the App Center	108
9.3. Next steps	108
9.4. Application meta file	109
9.5. Optional application files	116
9.6. Uploading the application	116
9.7. Notifications	117
9.8. Updates for the application	117
9.9. Integrating the Application in UCS	118
9.9.1. Automatic integration done by the App Center	118
9.9.2. Scope of the vendor	118
9.10. Best practices	119
9.10.1. Registration of LDAP and UDM Extensions	119

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.

Univention App Center is the simplest form to install applications on Univention Corporate Server. In the Univention Management Console it is the first entry point for their installation. Univention App Center is no app store in a classical point of view as known from iOS or Android and no license management for applications. The sale of licenses, maintenance or support for applications follows the default procedure of the respective vendor.

This chapter mainly aims at software vendors that want to provide their software in Univention App Center and describes the necessary steps.

A catalog with the available applications can be viewed on the Univention website¹.

9.1. Requirements

 Feedback 

The requirements for providing a software solution in Univention App Center are:

- Formally:
 - The certification of the software solution for UCS and sign the Univention Logoprogram Certified for Univention Corporate Server²
 - Sign the contract to use "Univention App Center" as platform to provide applications to users
- Technically: The packaging of the software solution in the Debian package format (see Section 9.2), provided for the architectures i386 and/or amd64
- Out-of-the-Box functionality: The app has to work already after installation

The following items are *no* disqualifying criteria:

- Missing documentation
- No or little integration of the app with Univention Corporate Server

The following items are important for packaging:

¹ <https://www.univention.com/products/univention-app-center/>

² <https://www.univention.com/products/univention-app-center/for-software-vendors/>

- The application has to be installed without any user interaction (unattended). The installation process is not allowed to be interrupted or canceled which would result in an inconsistent state of the package management.
- The application has to be installed and uninstalled cleanly. "Uninstalled cleanly" especially aims for removing Extended Attributes the application created during installation. See Section 3.5.
- The application does not (neither after installation nor after uninstallation) break Univention Corporate Server functionality.
- Recommended: The application is integrated in UCS and the UCS management system (see Section 9.9).

Note

The software vendor is responsible for the maintenance of his application(s). Package updates are provided for Univention App Center in a contemporary way (see also Section 9.8).

9.2. Packaging for the App Center

Feedback 

The App Center is based on the Debian package system. From a technical point of view, the installation of an application via the App Center consists roughly of two steps: (1) adding a new repository to the system which resides on the App Center server and (2) `apt-get` which installs the required package. This means that 3rd party vendors need to package their application as a `.deb` file and send that to Univention.

For deeper integration in Univention Corporate Server it is necessary to add at least one more file, a Join Script (see Section 9.9.2). This file can be included in the core package of the application or in a dedicated package for Univention Corporate Server integration, say, ***univention-appname*** and have a dependency on the core application (on `dpkg` level).

See Appendix B for further details on how to build Debian packages.

In most cases the core application depends on several other packages, e.g. ***apache***, ***python***, etc. Those packages are automatically installed by the App Center (via `apt`). All packages (except games) from Debian were rebuilt for Univention Corporate Server, but not all packages are maintained by Univention. The maintained repository for Univention Corporate Server contains the most common packages, but it may be that the application needs some specific packages not found there. If this packages exists for Debian, it can be found in the unmaintained repository. But as the unmaintained repository is not activated by default the App Center cannot install the application without user configuration if some packages are only available through the unmaintained repository. But this is a requirement for any application! To workaround this issue, the vendor has to do the following: In a first step he has to grab the packages from the unmaintained repository (see for example <https://updates.software-univention.de/4.1/unmaintained/>) and bundle it with his own packages so that they are put into the app's own repository. In a second step he should inform Univention about these packages. We will probably move them into maintained so that this procedure will become obsolete in future versions of the application.

9.3. Next steps

Feedback 

As soon as the software complies to the requirements the following steps are to be taken:

- Collect the meta data for the software for the App Center (see Section 9.4 and Section 9.5).
- Upload the packages along with the ini file, some README files, an icon and (maybe) a screenshot in a file archive (everything should go in one "directory"). See Section 9.6.
- Test the software solution with Univention Test App Center (more information is given by Univention after the upload).

Important

Univention will not test if the application works as expected. We have a test suite that is run before we publish the application but it only covers base functionality of Univention Corporate Server itself (e.g. LDAP, Join, etc.)

- Give approval for provisioning in the productive Univention App Center towards Univention.
- Show with the badge "Available in Univention App Center" that the App is ready to use. The HTML snippet will be provided by Univention.

See also Section 9.8 for how to proceed if an update is to be released.

9.4. Application meta file

Feedback 

The App Center needs some meta information about the application to handle the installation and to present information to the user. The application needs a so-called `.ini` file for that. The particulars in the meta data have to be in English at least.

An example along with some comments on the variables is attached:

```
[Application]
# internal id
ID=application

# display name of the application
Name=The Application

# application version
Version=7.1

# the logo that is used in the App Center overview
# scalable, square
Logo=myapplication.svg

# the logo shown in the App's details
# scalable, no ratio restriction (e.g. with the app's claim)
# if not set, defaults to Logo
LogoDetailPage=myapplication-big.svg

# whether the user's email address will be transmitted to the vendor
NotifyVendor=False

# a short description of the application (max. 90 characters)
Description>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, se.

# a more complete description of the application (max. 1400 characters)
# in order to use multiple lines, precede lines with one or more
# white spaces
LongDescription>Lorem ipsum dolor sit amet, consetetur sadipscing
elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore
magna aliquyam erat, sed diam voluptua. At vero eos et accusam et
justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea
takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor
sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor
```

Application meta file

```
invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
At vero eos et accusam et justo duo dolores et.

# reference to screenshot files (.png or .jpg file) or youtube videos
(optional)
Thumbnails=application_screenshot.jpg

# applicable category
Categories=System services

# website for more information about the product (e.g. landing page)
Website=http://www.software.com/products/application/welcome

# website for getting support (or information about how to buy a
license)
SupportURL=http://www.software.com/products/application/buy

# display name of the vendor
Vendor=Software GmbH

# contact email address for the customer
#Contact=

# email address that should be used to send notifications.
# If none is provided the address from "Contact" will be used
# Note: An empty email (NotificationEmail=) is not valid! Remove the
line (or
# put in comments) in this case
#NotificationEmail=

# optional: website of the vendor for more information
WebsiteVendor=http://www.software.com/

# optional: display name of the maintainer
# If the vendor does not maintain the application, this is the place,
# where the maintainer can be named. This value is optional.
Maintainer=Maintain GmbH

# optional: website of the maintainer for more information
WebsiteMaintainer=http://www.maintain.com

# optional: If the Application provides its own web interface
# it can be specified. The App Center then points to it if installed.
WebInterface=/application-webinterface

# optional: A dedicated name for the web interface may be given.
# If not, Name is taken
WebInterfaceName=Web interface of The Application

# WebInterface will be added to UCS startsite.
# Possible values: service, admin, False. If not given "service" is
used.
# Only useful when a WebInterface is defined
UCSOverviewCategory=admin
```

```
# The minimal amount of memory in MB. This value is compared with the
# currently available memory (without Swap) when trying to install the
# application. A value of 0 disables the check
MinPhysicalRAM=1024

# other applications which cannot be installed at the same time (comma
# separated list) (optional)
ConflictedApps=fooapp,barapp

# other applications which need to be installed first (comma separated
# list)
# (optional)
RequiredApps=bazapp,quxapp

# system packages which conflict with the application (comma separated
# list) ... these are essential similar to those conflicts specified in
# the debian packages themselves, however, in order to show conflicts
# directly in the app center and without querying the debian package
# information, these can be specified in the .ini file, as well
# (optional)
ConflictedSystemPackages=mysql5,python2.5

# application packages to be installed (comma separated list)
DefaultPackages=univention-application

# optional: If the application adds its own UMC Module, it can be added
# here. The App Center can then directly point to that module if it is
# installed. Name as specified in the UMC XML File of that module.
UMCModuleName=univention-application-module

# optional: A module may have a multiple Flavors. UDM uses this
# functionality. If the application does not add a dedicated UMC Module
# but extends UDM, the UMCModuleName should be "udm" and the flavor
# should be specified:
#UMCModuleFlavor=users/user

# optional: If domain users have to be somehow modified ("activated") to
# use the application, the following line should be included so that the
# App Center can give a hint and point to the Users module of UDM.
UserActivationRequired=True

# allow installation on these server roles
ServerRole=domaincontroller_master,domaincontroller_backup,domaincontroller_slave,

# allow installation on these architectures
# possible values are (comma separated): amd64, i386
# if not set, both architectures are supported
#SupportedArchitectures=amd64,i386

# whether a "Shop" button is displayed in the App details, leading to
# the ShopURL
UseShop=True
```

Application meta file

```
# URL that the user is directed to. Only makes sense in combination with
UseShop=True.
# Default is https://shop.univention.com
ShopURL=https://shop.maintain.com/application

# When UCS is not managing the domain but instead is only part of a
Windows
# controlled Active Directory domain, the environment in which the app
runs is
# different and certain services that this app relies may not not be
running.
# Thus, there are issues and incompatibilities that should be stated:
# * App should not be installed in an AD (results in not being shown in
the App
# Center in such environments):
#ADMemberIssueHide=True
# * App needs a password service running on the Windows domain
controller, e.g.
# because it needs the samba hashes to authenticate users (results in
a
# warning and a link to the documentation how to set up that service
in such
# environments)
#ADMemberIssuePassword=True

# German translations
[de]
Description>Lorem ipsum dolor sit amet, consetetur sadipscing elitr, se.
LongDescription>Lorem ipsum dolor sit amet, consetetur sadipscing
elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore
magna aliquyam erat, sed diam voluptua. At vero eos et accusam et
justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea
takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor
sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor
invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
At vero eos et accusam et justo duo dolores et.

# Sizing information about a single server running the app that manages
X users (computers...)
# Optional
#[Sizing: 10]
#CPU=1
#RAM=1 GB
#Disk=20 GB
#
#[Sizing: 100]
#CPU=2
#RAM=2 GB
#Disk=30 GB
#
#[Sizing: 5000]
#CPU=8
#RAM=32 GB
#Disk=500 GB
```

This file needs to be shipped with the application and should be updated with each new version of the application at least changing *Version*.

Besides the text file and the above mentioned statements, a product logo and a screenshot, if given, are needed:

- Product logo in SVG format, ratio: square. Required.
- "Big" product logo in SVG format (e.g. with claim, app name in logo, etc), ratio: no restrictions. Optional.
- Screenshot(s) of the application. Format can be JPG or (preferably) PNG. Optional.

A few variables of the ini file need further explanation:

ID

Needs to stay the same all the time. It is used to connect different versions.

Categories

Comma separated list of categories. In order to keep the overall list clean and simple an application should use categories from this list:

- Administration
- Business
- Collaboration
- Education
- System services
- UCS components
- Virtualization

Version

Has to rise every time. Needs to be comparable, i.e. versions consisting only of a codename are not useful. In the App Center, only the newest version (or the currently installed one) is shown. It is possible to install one specific version but this needs knowledge of internal IDs (beyond control even for the application vendor).

Logo

The file name of the logo of the App. It is used in the App Center overview when all Apps are shown in a gallery. As the gallery items are squared, the logo should be squared, too.

LogoDetailPage

The file name of a "bigger" logo. It is shown in the detail page of the App Center. Useful when there is a stretched version with the logo, the name, maybe a claim. If not given, the *Logo* is used on the detail page, too.

LongDescription

A (more exhaustive) description that will be displayed when viewing the details of the application. This description needs to be in HTML (this also means that characters like "&" need to be escaped to "&"). Other variables are escaped automatically if not stated otherwise (meaning that "&" may be used).

MinPhysicalRAM

The minimal amount of free memory in MB when trying to install the application. The application should work fluently above that limit at least for smaller environments. If this requirement is not met, it only shows a warning that can be overridden by the user.

Application meta file

RequiredApps

Application IDs for apps that need to be installed in order to run this one correctly. These applications will not be installed automatically. Multiple IDs may be split by comma.

Note

Application IDs are the *ID* values in the ini file. To find the ID, use `univention-app list`. The command is new in UCS 4.1.

RequiredAppsInDomain

Like *RequiredApps*, but the Apps may be installed anywhere in the domain, not necessarily on this very server. Uses Application IDs, multiple IDs may be split by comma.

Note

Application IDs are the *ID* values in the ini file. To find the ID, use `univention-app list`. The command is new in UCS 4.1.

ConflictedApps

Application IDs for apps that not may to be installed along with this App on the same server. Multiple IDs may be split by comma.

Note

Application IDs are the *ID* values in the ini file. To find the ID, use `univention-app list`. The command is new in UCS 4.1.

PortsExclusive

Comma separated list of ports that the App will use exclusively. For conventional Apps, this has no immediate effects but will build an implicit list of *ConflictedApps*. For Docker Apps, this will set up the bridge to the container accordingly and change the Firewall rules.

PortsRedirection

Same as *PortsExclusive*, but only for Docker Apps: The port in the Docker Container will be redirected to a port on the Docker Host. Syntax: `PortsRedirection=8080:80` will make the container's port 80 accessible on port 8080. Note that in this case it may be a better idea to use *WebInterface*.

ConflictedSystemPackages

Package names that may not be installed along with the application. It is highly recommended that this list is reflected on `dpkg` level in some other package of the application. This list is just to show the user conflicts *before* the application is to be installed. Conflicts on package level are resolved automatically and the user has to confirm if some packages will be uninstalled.

DefaultPackages

A list of packages that will be installed. It is recommended to keep this list as small as possible and install other packages as dependencies as this is less error prone. Preferably only one package like ***univention-appname*** is given here.

DefaultPackagesMaster (deprecated)

Warning

Old apps will have to be migrated eventually. Please see below for some easy replacements. Otherwise get in touch with Univention if you feel that you need `DefaultPackagesMaster` functionality.

Installing an application may require an extension of the LDAP schema. This can be done by providing a separate package for this schema file and install it on the master domain controller (and all backup domain controller servers). The App Center will install these packages automatically throughout the domain — in opposite to the *DefaultPackages* which are only installed locally. Another difference is that these packages will not be uninstalled even when the whole application is removed because a schema that was used before it is removed again will break the whole LDAP server.

It is possible (and preferable) to register schema extensions via the Join Script (see Section 9.10.1). In most cases this makes *DefaultPackagesMaster* unnecessary.

DefaultPackagesMaster is not allowed in Docker Apps.

Thumbnails

A comma separated list of screenshots and/or *YouTube* videos. Will be listed in the App's detail page of the UMC module. For screenshots, filenames need to be specified (and they need to be sent along with the ini file), for *YouTube* videos, a URL to that video has to be given. The URL needs to point directly to the video on <https://www.youtube.com/>

NotifyVendor

See Section 9.7 for further explanation. Has to be set to `True` to be activated.

SupportURL

Points to a website where the user can get/buy support for the app. If not set, the user will instead be encouraged to contact the app vendor (*Contact*). If set to `None`, the user will see something like "No support option provided".

ShopURL

If set, a button appears in the App Center: "Buy". This button leads to the URL specified here. The following parameters are sent to this URL via GET:

- `key_id` (Key ID of the License)
- `ucs_version` (major and minor version of UCS, e.g. 4 . 1)
- `app_id` (internal ID of the Application)
- `app_version` (current version of the Application)
- `locale` (locale used in the frontend, e.g. `de`)

WebInterface

If a web frontend is installed by the application, one may state this here. Should probably start with a `"/`.

UCSOverviewCategory

If there is a *WebInterface* the App Center automatically registers a link in the overview page of the UCS server. By default these are added in the "service" section of the overview page. If *UCSOverviewCategory* is set to `admin` the link will be generated there. Can be disabled completely if set to `False`. See also Section 9.9.1.

ServerRole

Roles that may use this application, separate by comma. Possible roles are `domaincontroller_master`, `domaincontroller_backup`, `domaincontroller_slave`, and `memberserver`. If not given, every role is allowed.

[Sizing: xx]

(Optional) Sections starting with `Sizing` are not evaluated by the App Center itself but give Cloud Service Providers a hint how to tailor a server for a customer. The number inside the section name means users (or something else depending on the app's functionality) working with the app (not simultaneously, of course). The following variables can be defined inside this section:

Optional application files

CPU: The number of cores, e.g., 2

RAM: The amount of memory and the unit, e.g., 2 GB

Disk: The amount of disk space and the unit, e.g., 20 GB

The numbers should be minimal, enabling a *reasonable* performance. The following sections are common: [Sizing: 10],[Sizing: 50],[Sizing: 100],[Sizing: 500],[Sizing: 1000],[Sizing: 5000].

9.5. Optional application files

Feedback 

Along with the `.ini` file, there are more files that can be shipped along with the Application, each of them optionally (at least in English if given):

LICENSE_AGREEMENT

A file the user has to accept before the installation starts. HTML is allowed in this file and it may be localized by using the files `LICENSE_AGREEMENT_EN` resp. `LICENSE_AGREEMENT_DE`.

README_INSTALL

A file that is presented to the user before installation (but after `LICENSE_AGREEMENT`). HTML is allowed. It may be localized by using the files `README_INSTALL_EN` resp. `README_INSTALL_DE`.

README_POST_INSTALL

A file that is presented to the user after installation. HTML is allowed. It may be localized by using the files `README_POST_INSTALL_EN` resp. `README_POST_INSTALL_DE`.

README_UPDATE

A file that is presented to the user before upgrading, listing changes. Note that a user may only upgrade to the newest version and does so in one step. Only the latest `README_UPDATE` is shown. It may be localized by using the files `README_UPDATE_EN` resp. `README_UPDATE_DE`.

README_POST_UPDATE

A file that is presented to the user after upgrading. Note that a user may only upgrade to the newest version and does so in one step. Only the latest `README_POST_UPDATE` is shown. HTML is allowed. It may be localized by using the files `README_POST_UPDATE_EN` resp. `README_POST_UPDATE_DE`.

README

A README file that the user may see in a dedicated Univention Management Console module for the installed application. Should contain something like first steps with the installed application and explain what can be done with it and how. HTML is allowed. It may be localized by using the files `README_EN` resp `README_DE`.

README_UNINSTALL

A file that is presented to the user before uninstallation. HTML is allowed. It may be localized by using the files `README_UNINSTALL_EN` resp. `README_UNINSTALL_DE`.

README_POST_UNINSTALL

A file that is presented to the user after uninstallation. HTML is allowed. It may be localized by using the files `README_POST_UNINSTALL_EN` resp. `README_POST_UNINSTALL_DE`.

9.6. Uploading the application

Feedback 

If the meta data (text file, logo and screenshot) and the packages are ready for upload, they can be put into a file archive (`.zip` or (preferably) `.tar.gz`) and uploaded to `http://upload.univention.de/`. If the archive is larger than 150 MB it has to be split into single parts not larger than 150 MB.

A unique upload ID will be displayed after the download that has to be forwarded to Univention. Please send the following information via e-mail to <appcenter@univention.de>:

- Upload ID for the file archive. Please provide all upload IDs if the archive had to be split.
- Grant permission to Univention to distribute the software through Univention App Center.

All further communication will be done through the ticket system after the upload.

9.7. Notifications

Feedback 

Univention always receives an estranged notification for statistical purposes upon installation and uninstallation of an application in Univention App Center that is only saved at Univention for data processing and will not be forwarded to any third party.

Depending on the guideline of the respective application vendor an updated UCS license key with so-called key identification (Key ID) is required for the installation of an application. In this case, the Key ID will be sent to Univention together with the notification. As a result the application vendor regularly receives a message from Univention with the following information:

- Name of the installed application
- Registered email address

The description of every application includes a respective indication for such cases and is shown before installation.

If a UCS environment does not have such a key at its disposal (e.g. UCS Core Edition) and the vendor requires a Key ID, the user will be asked to request an updated license key directly from Univention. Afterwards the new key can be applied and the application installed.

9.8. Updates for the application

Feedback 

If an update for an application is available, the software vendor has to provide the recent packages to Univention. The updated packages go in general through the same procedure as with the initial process described in Section 9.3.

This holds for minor updates (i.e. just small bug fixes) as well as major updates (added functionality, overhauled interface, etc.). The vendor is responsible for a smooth transition from one version to another.

Important

The App Center always installs the newest version, i.e. we do not incrementally upgrade an application. So each newer version needs to make sure that the update from *any* older version works.

For the App Center, there is no difference between a minor and a major update. The old version is completely replaced by the newer one. We do not natively support two (more or less independent) "versions lines" of the same application (e.g. one for those who want the newest features and one for those who wish to have a stable platform with nothing but conservative bug fixes). If this is desired, two separate applications need to be uploaded to the App Center that will show up next to each other. In this case the ID in the ini file needs to be changed and the application somehow needs to make sure that they cannot both be installed in parallel (e.g. a version dependency on dpkg level of some meta package). Univention may help you with the details.

Updates for the application description (.ini file), the product logo, the screenshot, readme files (especially README_UPDATE) and the packages are all submitted to <appcenter@univention.de>. Only those

files that changed need to be uploaded again, everything else can be copied by Univention. The ini file needs to change every time, because the *Version* has to be increased. If this is the only change in the ini file, this may be stated in the email instead of shipping the file and we will take care of that during copying. Please use the upload if a higher amount of data has to be transferred (see Section 9.6).

As the application has already gone through the initial approval process and passed, Univention will offer to "semi-automatically" publish the application. I.e. we will run our test suite while giving you the chance to test your application. If our tests go well and we do not hear anything contrary from you, we will publish the application by then.

9.9. Integrating the Application in UCS

 Feedback 

9.9.1. Automatic integration done by the App Center

 Feedback 

Some tasks can be done automatically by the App Center and do not need any (or much) help from the vendor.

The overview of any UCS system (i.e. plain `https://IP/`) can link to various web frontends, e.g. the Univention Management Console. If the application provides a *WebInterface* in the ini file, a link is built automatically, with the ini's *Name* and *Description* as text.

This feature can be disabled by stating `UCSOverviewCategory=False`. This may be a good idea if there are multiple web frontends and they should be maintained manually and in one place. In this case the following UCR variables should be set in `postinst` and removed in `postrm` (see also Section 8.1):

```
ucs/web/overview/entries/service/appid/icon
```

Path to the icon for the link.

```
ucs/web/overview/entries/service/appid/label
```

English version of the headline. German version is `ucs/web/overview/entries/service/appid/label/de`

```
ucs/web/overview/entries/service/appid/description
```

English version of the short text below the headline. German version is `ucs/web/overview/entries/service/appid/description/de`

```
ucs/web/overview/entries/service/appid/link
```

Link to the web interface. Probably something like `/appid`.

Each application registers itself in the LDAP directory and sets the hosts on which it is installed automatically. These information can be accessed by any *machine connection*. The path is `univentionAppID=appid_appversion,cn=appid,cn=apps,cn=univention,ldap_base`. See also Chapter 6 if this data shall be accessed.

9.9.2. Scope of the vendor

 Feedback 

Reuse users, groups, computers already administered in domain.

An Application may need more attributes than there are present. For example something like *This user is allowed to use the app*. For this it needs to add Extended Attributes (see Section 6.2) in a Join Script (see Chapter 3) in a package from *DefaultPackages*. It will also have to extend the LDAP schema. This can also be done in the Join Script (see Section 9.10.1) or, alternatively, in a package of *DefaultPackagesMaster*.

If the application wants to sync certain attributes (for example passwords) there are listener modules. See Chapter 5.

9.10. Best practices

Feedback 

9.10.1. Registration of LDAP and UDM Extensions

Feedback 

LDAP schema and ACL extensions as well as UDM syntax, UDM hook and UDM extension modules can be installed via the library function `ucs_registerLDAPExtension`, see Section 3.4.3.2.

Chapter 10. Integration of external repositories

10.1. Integration of repository components via Univention Management Console	121
10.2. Integration of repository components via Univention Configuration Registry	122

Sometimes it might be necessary to add external repositories, e.g. when testing an application which is developed for the UCS@school. Such components can be registered via 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

All versions of the same major number will be used. If for example UCS-4.2 is installed, all repositories of the component with version numbers 4.0, 4.1 and 4.2 will be used if available.

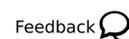
current

current Using the keyword *current* will likewise include all versions of the same major version. Additionally it will block all minor and major upgrades of the installed UCS system until the respective component is also available for the new release. Patch level and errata updates are not affected. If for example UCS-3.1 is currently installed and UCS-3.2 or UCS-4.0 is already available, the release updated will be postponed until the component is also available for version 3.2 and 4.0 respectively.

major.minor

By specifying an explicit version number only the specified version of the component will be used. Release updates of the system will not be hindered by such components. Multiple versions can be given using commas as delimiters, for example *3.2,4.0*.

10.1. Integration of repository components via Univention Management Console



A list of the integrated repository components is in the UMC module **Repository Settings**. Applications which have been added via the Univention App Center are still listed here, but should be managed via 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 host name of the download server is to be entered in the input field **Repository server**, and, if necessary, an additional file path in **Repository prefix**.

A **Username** and **Password** can be configured for repository servers which require authentication.

A software component is only available once **Enable this component** has been activated.

A differentiation is also made for components between *maintained* and *unmaintained* components.

10.2. Integration of repository components via Univention Configuration Registry

The following Univention Configuration Registry variables can be used to register a repository component. It is also possible to activate further functions here which cannot be configured via the UMC module. *NAME* stands for the component's name:

`repository/online/component/NAME/server`

The repository server on which the components are available. If this variable is not set, the server from the Univention Configuration Registry variable `repository/online/server` uses.

`repository/online/component/NAME`

This variable must be set to *enabled* if the components are to be mounted.

`repository/online/component/NAME/localmirror`

This variable can be used to configure whether the component is mirrored locally. In combination with the Univention Configuration Registry variable `repository/online/component/NAME/server`, a configuration can be set up so that the component is mirrored, but not activated, or that it is activated, but not mirrored.

`repository/online/component/NAME/description`

A descriptive name for the repository.

`repository/online/component/NAME/prefix`

Defines the URL prefix which is used on the repository server. This variable is usually not set.

`repository/online/component/NAME/username`

If the repository server requires authentication, the user name can be entered in this variable.

`repository/online/component/NAME/password`

If the repository server requires authentication, the password can be entered in this variable.

`repository/online/component/NAME/version`

This variable controls the versions to include, see Chapter 10 for details.

`repository/online/component/NAME/defaultpackages`

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 is not installed. Specifying the package list eases the subsequent installation of components.

Chapter 11. Translating UCS

11.1. Univention Management Console translations	123
11.1.1. Install needed tools	123
11.1.2. Obtain a current checkout of the UCS Subversion repository	123
11.1.3. Create a new translation package	123
11.1.4. Edit translation files	123
11.1.5. Update the translation package	124
11.1.6. Build the translation package	124

11.1. Univention Management Console translations

Feedback 

By default UCS includes English and German localizations. Univention provides a set of tools that facilitates the process of creating translations for Univention Management Console.

This section describes all steps necessary to create a working translation package for UCS. We recommend having a running UCS installation where the tools can be set up in an easy manner. Further more a current Subversion checkout of the UCS source code is required.

11.1.1. Install needed tools

Feedback 

The package *univention-ucs-translation-template* contains all tools required to setup and update a translation package. It requires some additional Debian tools to build the package. Run the following command on your UCS to install all needed packages.

```
univention-install univention-ucs-translation-template dpkg-dev
subversion
```

11.1.2. Obtain a current checkout of the UCS Subversion repository

Feedback 

The Subversion 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
svn co http://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-3
```

11.1.3. Create a new translation package

Feedback 

To create a new translation package for, e.g., French in the current working directory, the following command must be executed:

```
cd ~/translation
univention-ucs-translation-build-package \
-s ~/translation/ucs-4.1-3 -c fr -l fr_FR.UTF-8:UTF-8 -n French
```

This creates a new directory `~/translation/univention-ucs-translation-fr/` which contains a Debian source package of the same name. It includes all source and target files for the translation.

11.1.4. Edit translation files

Feedback 

The translation source files (.po files) are located below the directory `~/translation/univention-ucs-translation-fr/fr`. Each file should be edited to create the translation.

Update the translation package

These files are generated by the package *gettext*. The manual can be found at <http://www.gnu.org/software/gettext/manual/gettext.html>. Translation files created by *gettext* consist of a header and various entries of the form

```
#: 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 string `fuzzy` indicates an entry, which was copied by *gettext* from a previous version and needs to be updated.

The line starting with `msgid` contains the original text. The translation has to be placed on the line containing `msgstr`. 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.

After a file has been translated completely, the line containing `fuzzy` at the beginning of the entry should 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.

11.1.5. Update the translation package

Feedback 

First update your Subversion checkout:

```
cd ~/translation/ucs-4.1-3
svn update
```

If changes affecting translations are made in the Subversion repository, existing translation packages need to be updated to reflect those changes. Given a path to an updated Subversion checkout, `univention-ucs-translation-merge` can update a previously created translation source package. The following example will update the translation package *univention-ucs-translation-fr*:

```
univention-ucs-translation-merge ~/translation/ucs-4.1-3 \
~/translation/univention-ucs-translation-fr
```

11.1.6. Build the translation package

Feedback 

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-ucs-translation-fr  
dpkg-buildpackage -uc -us -b  
dpkg -i ../univention-ucs-translation-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 will be still shown in their original language, i.e. in English.

Chapter 12. Univention Updater

12.1. Separate repositories	127
12.2. Updater scripts	127
12.2.1. Digital signature	128
12.3. Release update walkthrough	128

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.

12.1. Separate repositories

Feedback 

UCS releases are provided either via DVD images or via online repositories. For each major, minor and patch-level 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 encourage 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_errata__components_update.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 *????* manual. Setting the variable `.../version=current` can be used to mark a component as required, which blocks an upgrade until the component is available for the new release.

12.2. Updater scripts

Feedback 

In addition to the regular Debian Maintainer Scripts (see Section B.3.5) 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 is 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: Once before the main release `preup.sh` script is called and once more 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 is 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-4.1 this would be `4.1/maintained/4.1-0/all/preup.sh` and `4.1/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 Procedure 12.1.

12.2.1. Digital signature

Feedback 

From UCS 3.2 on the scripts must be digitally signed by an PGP (Pretty Good Privacy) key stored in the keyring of apt-key(8). The detached signature must be placed in a separate file next to each updater scripts with the additional file name 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 keyphrase-file -o script.sh.gpg -b script.sh`

Signatures can be checked manually using the following command: `gpgv --keyring /etc/apt/trusted.gpg script.sh.gpg script.sh`

12.3. Release update walkthrough

Feedback 

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.

Procedure 12.1. Update process steps

1. Create temporary source list file `00_ucs_temporary_errata__components_update.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. Execute `component-postup.sh pre $version`
9. Execute `release-postup.sh $version`
10. Execute `component-postup.sh post $version`
11. Set the release related Univention Configuration Registry variables to the new version

Chapter 13. *Single Sign-On*: Integrating a service provider into UCS

13.1. Register new service provider via <i>udm</i>	129
13.2. Get information required by the service provider	129
13.3. Add direct login link to <i>ucs-overview</i> page	130

UCS provides *Single Sign-On* functionality with a SAML 2.0 compatible identity provider based on *simplesamlphp*. The identity provider is by default installed on the DC Master and all DC Backup servers. A DNS Record for all systems providing *Single Sign-On* services is registered for failover, usually `ucs-sso.domainname`. Clients are required to be able to resolve the *Single Sign-On* DNS name.

13.1. Register new service provider via *udm*

 Feedback 

New service providers can be registered by using the *UDM* module `saml/serviceprovider`. To create a new service provider entry in a *joinscript*, see the following example:

```
eval "$(ucr shell)"
udm saml/serviceprovider create "$@" \
  --ignore_exists \
  --position "cn=saml-serviceprovider,cn=univention,$ldap_base" \
  --set isActivated=TRUE \
  --set Identifier="MyServiceProviderIdentifier" \
  --set NameIDFormat="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified" \
  --set simplesamlAttributes="false" \
  --set AssertionConsumerService="https://$hostname.$domainname/sso-login-page" \
  --set simplesamlNameIDAttribute="uid" \
  --set privacyPolicyURL="https://example.com/policy.html" \
  --set serviceProviderOrganizationName="My Service Name" \
  --set serviceProviderdescription="A long description shown to the user on the Single Sign-On page." || die
```

13.2. Get information required by the service provider

 Feedback 

The service provider usually requires at least a public certificate or XML metadata about the identity provider. The certificate can e.g. be downloaded with the following call:

```
eval "$(ucr shell)"
wget --ca-certificate /etc/univention/ssl/ucsCA/CAcert.pem \
  https://"${ucs_server_sso_fqdn:-ucs-sso.$domainname}"/simplesamlphp/saml2/idp/certificate \
  -O /etc/idp.cert
```

The XML metadata is available e.g. from

```
eval $(ucr shell)
wget --ca-certificate /etc/univention/ssl/ucsCA/CAcert.pem \
  https://"${ucs_server_sso_fqdn:-ucs-sso.$domainname}"/simplesamlphp/saml2/idp/metadata.php \
  -O /etc/idp.metadata
```

Add direct login link to ucs-overview page

The *Single Sign-On* Login page to be configured in the service provider is <https://ucs-sso.domainname/simplesamlphp/saml2/idp/SSOService.php>

13.3. Add direct login link to *ucs-overview* page

Feedback 

To provide users with a convenient link to an identity provider initiated login, the following ucr command may be used

```
fqdn="ucs-sso.domainname"
myspi="MyServiceProviderIdentifier"
ucr set ucs/web/overview/entries/service/SP/description="External
Service Login" \
ucs/web/overview/entries/service/SP/label="External Service SSO" \
ucs/web/overview/entries/service/SP/link="https://$fqdn/simplesamlphp/
saml2/idp/SSOService.php?spentityid=$myspi" \
ucs/web/overview/entries/service/SP/description/de="Externer Dienst
Login" \
ucs/web/overview/entries/service/SP/label/de="Externer Dienst SSO" \
ucs/web/overview/entries/service/SP/priority=50
```

Where *MyServiceProviderIdentifier* is the identifier used when creating the UDM service provider object.

Chapter 14. Miscellaneous

14.1. Databases	131
14.1.1. PostgreSQL	131
14.1.2. MySQL	131
14.2. UCS lint	131
14.3. Function Libraries	133
14.3.1. <i>shell-univention-lib</i>	133
14.3.2. <i>python-univention-lib</i>	133
14.4. Login Access Control	134
14.5. Network Packet Filter	135
14.5.1. Filter rules by Univention Configuration Registry	135
14.5.2. Local filter rules via iptables commands	136
14.5.3. Testing Univention Firewall settings	136

14.1. Databases

Feedback 

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

Feedback 

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. MySQL

Feedback 

By default the MySQL 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

Feedback 

Use `ucslint` to find packaging mistakes. Called best from `debian/rules`, needs build dependency on *ucslint*.

```
override_dh_auto_test:
dh_auto_test
ucslint
```

For each issue, `ucslint` prints one line, which line 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 mis-spellings of *Univention*

0016-Deprecated

Checks files for usage of deprecated functions

0017-Shell

Checks shell scripts for quoting errors

The module and test number may be optionally followed by a file name, 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 that a file `debian/ucslint.overrides` can be created with list of modules and test, which should be ignored. Without specifying the optional filename, line number and column number, the test is globally disabled for all files.

14.3. Function Libraries

Feedback 

The source package *univention-lib* provides two binary packages *shell-univention-lib* and *python-univention-lib*, which contain common library functions usable in shell or Python programs.

14.3.1. *shell-univention-lib*

Feedback 

This package provides several 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/base.sh`

This file contains some helpers to create log files, handle unjoin scripts (see Section 3.5) or query the network configuration.

`/usr/share/univention-lib/ldap.sh`

This file contains some helpers to query data from LDAP, register and unregister 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 contains some helpers to handle boolean Univention Configuration Registry variables and handle UCR files on package removal.

`/usr/share/univention-lib/umc.sh`

This file contains some helpers to handle UMC (see Chapter 7) 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*

Feedback 

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.error

This module provides the function `formatTraceback`, which returns the full stack trace for an exception.

univention.lib.fstab

This module provides some functions for handling the file `/etc/fstab`.

```
/usr/share/pyshared/univention/lib/getMailFromMailOrUid.py { uid |
email }
```

This program returns the distinguished name of the user, which either matches the user identifier or email address given to the command as an argument.

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 Section 9.10.1.

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.shell

This module provides two functions for escaping shell command line arguments and creating at jobs.

univention.lib.ucrLogrotate

This module provides some helper functions internally used for parsing the Univention Configuration Registry variables related to `logrotate(8)`.

univention.lib.ucs

This module provides the class `UCS_Version` to more easily handle UCS version strings.

univention.lib.umc_connection

This module provides the class `UMCConnection` to handle connections to remote UMC servers.

univention.lib.umc_module

This module provides some functions for handling icons.

univention.lib.urllib2_ssl

This module provides a pack-port of `urllib2` from Python-3.3, which implements proper certificate checking.

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

 Feedback 

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

 Feedback 

Besides predefined service definitions, Univention Firewall also allows the implementation of package filter rules via Univention Configuration Registry. These rules are included in `/etc/security/packetfilter.d/` via a Univention Configuration Registry module.

Filter rules can be provided via 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.

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:

Example 14.1. Local firewall rule

```
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 via `iptables` commands

Feedback 

Besides the existing possibilities for settings via Univention Configuration Registry, there is also the possibility of integrating user-defined enhanced configurations in `/etc/security/packetfilter.d/`, e.g. 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 Section 2.2.1.1.

Full documentation for IPTables can be found at <http://www.netfilter.org/>.

14.5.3. Testing Univention Firewall settings

Feedback 

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)`

Example 14.2. Using `nmap` for firewall port testing

```
nmap 192.168.1.100 -p 400
nmap 192.168.1.110 -sU -p 400-500
```

Appendix A. Bug reporting

UCS is neither error free nor feature complete. Issues are tracked using Bugzilla at <https://forge.univention.org/bugzilla/>.

Create an account.

Search for existing entries before opening new reports.

Include the version info: `ucr search --brief ^version/`.

Provide enough information to help us reproduce the bug.

Search <http://sdb.univention.de/>

Search <http://wiki.univention.de/>

Search <http://forum.univention.de/> 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.

Appendix B. 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*.

B.1. Prerequisites and preparation

Feedback 

Some packages are required for creating and building packages.

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 [ucs-handbuch].

B.2. `dh_make`

Feedback 

`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 package 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)` 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:

```
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.

B.3. Debian control files

Feedback 

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 [Debian FAQ].

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,¹

The following files are required:

`changelog`

Changes related to packaging, not the upstream package. See Section B.3.3 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)` for more details.

¹ 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.

control

Contains control information about the source and all its binary packages. This mostly includes package name and dependency informations. See Section B.3.1 below for more information.

copyright

This file contains the copyright and license information for all files contained in the package. See Section B.3.2 below for more information.

rules

This is a Makefile style file, which controls the package build process. See Section B.3.4 below for more information.

source/format

This file configures how `dpkg-source(1)` 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)`.

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)` for more details.

package.dirs

List of extra directories to be created. See `dh_installdirs(1)` for more details.²

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)` for more details.

package.docs

List of documentation files to be installed in `/usr/share/doc/package/`. See `dh_installdocs(1)` 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)` for more details.

package.doc-base*

Control files to install and register extended HTML and PDF documentation. See `dh_installdocs(1)` for more details.

² May other `dh_` tools automatically create directories themselves, so in most cases this file is unneeded.

package.init.d,
package.default

Start-/stop script to manage a system daemon or service. See `dh_installinit(1)` 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)` for more details.

package.menu

Control file to register programs with the Debian menu system. See `dh_installmenu(1)` 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)` for more details.

package.preinst,
package.postinst,
package.prerm,
package.postrm

Scripts to be executed before and after package installation and removal. See Section B.3.5 below for more information.

package.maintscript

Control file to simplify the handling of *conffiles*. See `dpkg-maintscript-helper(1)` and `dh_installdeb(1)` for more information.

Other debhelper programs use additional files, which are described in the respective manual pages.

B.3.1. debian/control

Feedback 

The `control` file contains information about the packages and their dependencies, which is 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`³

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.

³ <http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections>

Priority⁴

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.

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 build 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.

⁴ <http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities>

Most packages are not self-contained but need other packages for proper function. Debian supports different kinds of dependencies.

Depends

A 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 packets, 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.

```
dh_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.

```
dh_python:Depends
```

`dh_python` detects similar dependencies for Python modules.

```
dh_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
```

An Example:

```
Depends: libexample1 (>= ${binary:Version}),
  exim4 | mail-transport-agent,
  ${shlibs:Depends}, ${misc:Depends}
Conflicts: libgg0, libgg1
Recommends: libncurses5 (>> 5.3)
Suggests: libgii0-target-x (= 1:0.8.5-2)
Replaces: vim-python (<< 6.0), vim-tcl (<= 6.0)
Provides: www-browser, news-reader
```

B.3.2. debian/copyright

Feedback 

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>
```

```
License:
```

```
<Must follow here>
```

The file does not require any specific format. Debian now recommends to use a machine-readable format, but this is not required for UCS. The format is described in <http://dep.debian.net/deps/dep5/> 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: 2013-2015 Univentio GmbH
License: AGPL
```

B.3.3. debian/changelog

Feedback 

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 time stamp must follow the format described in RFC 2822⁴. `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 Appendix A) to reference additional details of the bug fixed. Other parties are encouraged to devise similar comments, e.g. URLs to other bug tracking systems.

B.3.4. debian/rules

Feedback 

The file `rules` describes the commands needed to build the package. It must use the Make syntax `[make]`. It consists of several rules, which have the following structure:

```
target: dependencies
command
...
```

⁴ <http://tools.ietf.org/html/rfc2822>

Each rule starts with the target name, which can be a file name 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 file names 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 Section B.3. 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.

B.3.5. `debian/preinst`, `debian/prerm`, `debian/postinst`, `debian/postrm` Feedback

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

It is important that these scripts handle error conditions properly: Maintainer scripts should terminate 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.

B.4. Building

Feedback 

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.

B.5. Further reading

Feedback 

- [Debian FAQ]
- [Debian Guide]
- [Debian Policy]
- [Debian Reference]

Bibliography

[ucs-handbuch] Univention GmbH. 2015. *Univention Corporate Server - Manual for users and administrators*. <https://docs.software-univention.de/manual-4.1.html>.

[make] Free Software Foundation. 2010. *The GNU Make manual*¹.

[ISO639] International Organization for Standardization. 2002. *ISO 639-1: Alpha-2 code*².

[Debian FAQ] Debian. 2012. *The Debian GNU/Linux FAQ - Basics of the Debian package management system*³.

[Debian Guide] Debian. 2013. *Debian New Maintainers' Guide*⁴.

[Debian Policy] Debian. 2012. *Debian Policy Manual*⁵.

[Debian Reference] Debian. 2012. *Debian Developer's Reference*⁶.

¹ <http://www.gnu.org/software/make/manual/>

² <http://www.loc.gov/standards/iso639-2/>

³ http://www.debian.org/doc/manuals/debian-faq/ch-pkg_basics

⁴ <http://www.debian.org/doc/devel-manuals#maint-guide>

⁵ <http://www.debian.org/doc/debian-policy/>

⁶ <http://www.debian.org/doc/manuals/developers-reference/>

Index

A

Apache (see Web Services)
App center, 107

B

Bug (see Bugzilla)
Bugzilla, 137

C

Config Registry, 19
 Categories, 26
 Configuration files, 22
 Descriptions, 25
 Examples, 29
 Multifile, 31
 Services, 32
 Single File, 29
 Repository, 17
 Services, 27
 Template
 Module, 25
 Multi file, 24
 Script, 25
 Single file, 23
 Template file, 28
Custom Attributes (see Extended Attributes)

D

Database, 131
 MySQL, 131
 PostgreSQL, 131
Directory Listener, 53
 Cache, 65
 Credentials, 65
 Debug, 65
 Example module, 56
 modrdrn, 57
 Notifier ID, 66
 Verify, 66
Directory Manager, 69
 Custom Modules, 79
 Extended Attributes (see Extended Attributes)
 Hook extension, 41
 Hooks
 Packaging, 84
 LDAP search, 81
 Module extension, 41
 Syntax extension, 41
 Syntax override, 80
Domain join, 35

Domain credentials, 50
 Machine credential change, 50
Join script (see Join script)
Join status, 35
Running, 35

E

Example
 Config Registry, 29
Extended Attributes, 70
 Hooks, 77
 Options, 76
 Selection list, 74

J

Join (see Domain join)
Join script
 Exit codes, 38
 Helpers (see Library)
 Library, 38
 Return codes (see Exit codes)
 Writing, 36

L

LDAP
 Access control list extension, 41
 Schema extension, 41
Listener (see Directory Listener)
Localisation (see Translation)

M

Management Console, 89
 Files, 94
 Module
 Disable, 103
 LDAP, 102
 System, 95
 umc-modules, 94
 XML, 95

P

Package
 binary-, 139
 source-, 139
Packaging, 11
 Build dependencies, 139
 Checking for errors, 131
 Debian, 139
 Library functions, 133
 Modifying existing package, 11
 New package, 12
 Package repository, 17
postup (see Updater)

preup (see Updater)

R

Registry (see Config Registry)

Repository (see Packaging)

S

Server password change (see Domain join)

Single Sign-On

 SAML, 129

SSO (see Single Sign-On)

T

Translation, 123

U

UCR (see Config Registry)

UDM (see Directory Manager)

UMC (see Management Console)

Univention Directory Listener (see Directory Listener)

Univention Directory Manager (see Directory Manager)

Univention Management Console (see Management Console)

Update (see Updater)

Updater

 Repositories, 127

 Scripts, 127

 System update, 127

Upgrade (see Updater)

W

Web Services, 105