univention
be open.

App Tutorial

Building Apps for the Univention App Center

univention
be open.

Table of Contents

Chapter 1. Apps and Univention App Center

Univention App Center provides a platform for software vendors and an easy-to-use entry point for Univention Corporate Server (UCS) users to extend their IT environment with business software. The App Center is part of the web-based UCS management system and gives an overview of available and installed apps. Its purpose is to present available business applications for UCS and simplify their installation. This allows their easy evaluation and fosters the purchase decision.

Apps are the content of the App Center and they consist of the business software and some meta data about the presentation in the App Center. Most of them come with an integration into UCS, e.g. the management system or the mailstack. The purpose of an app is to provide the business solution in a way that it is ready to use after the installation and that comes with a decent default configuration to offer a satisfying impression of the solution. The installation is non-interactive and is done by just a click. Furthermore, an app utilizes the benefits of UCS and the business solution.

The App Center infrastructure consists of two parts: The already mentioned frontend as part of the web-based UCS management system and the server side component that stores the app meta data and the app software packages in their own respective repositories. The server side infrastructure is operated by Univention. The technological basis for installation and updates of the apps is APT, the well known advanced package tool from Debian. Therefore, the app needs to consist of so-called Debian packages. The App Center frontend is responsible for the app's presentation. As soon as an app is clicked to be installed or updated, the App Center activates the respective repository and the further process is handed over to apt which takes care of the rest like for example dependency resolution.

The next sections explain how to prepare your business solution as app for UCS. It also outlines the integration possibilities and describes what to do by example. Let's go!

Chapter 2. Prepare the environment

Before you can start with the creation of an app for Univention App Center, you'll need to prepare your UCS environment. This section guides you through the necessary steps.

2.1. Download

Feedback

First of all, get yourself a copy of UCS free of charge at the Univention Website[1]. You can choose between an ISO image or a pre-installed virtual machine.

2.2. Initial setup

Feedback

Please refer to the UCS Quickstart Guide[2] for the steps about installation and initial setup.

2.3. Activate the unmaintained repository

Feedback

UCS is a Linux distribution derived from Debian GNU/Linux. It behaves very similar and therefore software is installed from software repositories. UCS comes with the same packages as Debian (except the packages from the games section). The packages are provided through two repositories: maintained and unmaintained. Only the maintained repository is always activated by default.

To install your solution, you may need packages that are in the unmaintained repository. Please activate it:

```
ucr set repository/online/unmaintained='yes'
```

Note

Please remember the packages you need from unmaintained repository and provide the list later with your upload. Univention will copy those packages besides your app packages and make sure that the package dependencies from the unmaintained repository are met without prior activation by the user.

2.4. Install required packages for App development

Feedback

To build your software on UCS you will need to install build tools for Debian packages. The corresponding package can be installed with

```
univention-install build-essential debhelper
```

Depending on your app you may furthermore require several development libraries (e.g. ***libc-dev***, ***php5-dev***). For UCS integration packages, we recommend

```
# ucslint checks for common mistakes in a variety of files if enabled
# in debian/rules
#  see https://docs.software-univention.de/developer-
reference.html#misc:ucslint
```

[1] https://www.univention.com/downloads/ucs-download/
[2] https://docs.software-univention.de/quickstart-en.html

```
univention-install ucslint
```

```
# univention-config-dev takes care of installing and registering UCR
# variables if enabled in debian/rules
#   see https://docs.software-univention.de/developer-
reference.html#chap:ucr
univention-install univention-config-dev
```

```
# If you are developing a UMC module to extend the management console,
# you will need
univention-install univention-management-console-dev
```

If you already have a source directory with working code

```
dpkg-checkbuilddeps
```

should list the missing build dependencies, if any.

Chapter 3. Package the software solution

3.1. Create or use Debian packages

Feedback

You as ISV already distribute your software solution in a certain way. Univention App Center makes heavy use of the Debian package manager `dpkg` and the technology around it. Therefore, it is required that the software is provided in the Debian package format and that it can be installed non-interactively, e.g. the user will not be asked any questions for software configuration. This step has to be moved to a later step following the package installation.

Please follow this checklist:

1. If your software is provided via `.deb` files, you already have Debian packages. Please install those packages on UCS for testing purpose and evaluate if the software works as expected.

2. If your software is not provided via `.deb` files, Debian packages have to be created. Please follow the chapter packaging software[1] in the UCS developer reference about how to create Debian packages.

3.2. Structure the App

Feedback

In most cases packages of an app for Univention App Center in principle consist of:

1. packages including the vanilla software solution of the ISV

2. packages with the integration of the software solution with Univention Corporate Server

For the ease of app maintenance it is recommended to provide the vanilla software from 1. in packages on their own, independent from UCS. This allows to theoretically use the packages on other Debian-based Linux Distributions like for example Debian GNU/Linux itself or Ubuntu.

The UCS specific part from 2. should be collected in a separate package. This package depends on the "main" package from 1. and therefore automatically installs all the other packages needed via the dependency resolution of the package manager.

[1] https://docs.software-univention.de/developer-reference.html#chap:packaging

Chapter 4. Integration with UCS

Univention Corporate Server (UCS) is not just an enterprise Linux distribution. With its focus on identity and infrastructure management it has a lot of information saved about the IT infrastructure environment, the user accounts and the groups and the system configuration, to name a few.

The most obvious integration makes use of the numerous user accounts stored in the UCS directory service. Apps using this information avoid double effort in user administration. They may technically make just a simple LDAP bind for user authentication. Or if the app needs certain user attributes in its own persistence layer (e.g. the database) they may be synchronized via the Listener-/Notifier mechanism. Furthermore, existing data can be extended with app specific attributes, e.g., shall a user be allowed to use the app or what role shall the user occupy for the app. The UCS management system can be extended by attributes and the information is usually stored in the directory service. It is even possible that certain values or their change may trigger certain actions. A third integration possibility is to hook up the app in existing solution stacks of UCS, for example the mail stack or the web server. The app will among others benefit from a working configuration and a higher communication security because of already present security certificates.

Those are just a few examples to outline the possibilities for the integration. There are many more. The guiding question for the integration should be: What information about the infrastructure, the configuration and identities does UCS offer that the app will benefit from and saves efforts for the administrator?

The following sections give an impression of several integration scenarios. Further information can be found in the UCS developer reference[1].

4.1. Read information from the directory service

One primary element of the UCS management system is an LDAP directory in which the data required across the domain for the administration are stored. In addition to the user accounts and similar elements, the data basis of services such as DHCP is also saved there.

An LDAP directory has a tree-like structure, the root of which forms the so-called basis of the UCS domain. The UCS domain forms the common security and trust context for its members. An account in the LDAP directory establishes the membership in the UCS domain for users. Computers receive a computer account when they join the domain.

UCS utilizes OpenLDAP as a directory service server. The directory is provided by the master domain controller and replicated on all domain controllers (DCs) in the domain. The complete LDAP directory is also

[1] https://docs.software-univention.de/developer-reference.html

replicated on a DC backup as this can replace the DC master in an emergency. In contrast, the replication on DC slaves can be restricted to certain areas of the LDAP directory using ACLs (access control lists) in order to realize a selective replication.

The OpenLDAP server of UCS listens on port 7389 by default, not on 389. This is due to Samba 4 requiring port 389.

More information about the OpenLDAP server in UCS can be found in the manual[2].

4.1.1. Access the LDAP directory

Feedback

Sometimes software can use LDAP, but does not use the user accounts directly but is restricted to one specific user who then is used for further user authentication. This LDAP bind can be done by creating a app specific user in UDM. This should be done in a Join script via

```
ldap_base="$(ucr get ldap/base)"
APP='myapp'
PASSWORD='secret'
touch "/etc/$APP.secret"
chown root:root "/etc/$APP.secret" # or so
chmod 600 "/etc/$APP.secret"
printf '%s' "$PASSWORD" > "/etc/$APP.secret"
udm users/user create "$@" --position "cn=users,$ldap_base" \
 --set username="$APP-user" --set lastname="$APP-user" \
 --set password="$PASSWORD" --option ldap_pwd || die
```

Now you can configure your software accordingly. Here the DN will be `uid=`*`$APP-user`*`,cn=users,$ldap_base`.

If more access is needed, it is also possible to use the machine account of the UCS system. Every computer joined into the UCS domain has certain permissions. Computers in `cn=dc,cn=computers,$ldap_base` (by default DC Master, DC Backup, DC Slave) can even access the (hashed) password attributes of users and computers. The password for the machine account is stored in `/etc/machine.secret` (readable by `root`). The machine DN can be found by

```
ucr get ldap/hostdn
```

The machine password rotates. This means the password changes over time. If your software needs to adapt, you may install a script (with executable bit set!) at `/usr/lib/univention-server/server_password_change.d/`*`xx$app`* with *xx* being two digits for ordering purposes with something like the following content:

```
#!/bin/sh
case "$1" in
prechange)
 # nothing to do before the password is changed
 exit 0
 ;;
nochange)
 # nothing to do after a failed password change
 exit 0
 ;;
postchange)
 # do something with /etc/machine.secret, e.g.
 cp /etc/machine.secret /etc/$app.secret
```

[2] https://docs.software-univention.de/manual.html#domain:ldap

```
 # restart daemon after password was changed
 invoke-rc.d $app restart
 ;;
esac
```

4.1.2. Listener-/Notifier Mechanism

The data regarding identity and infrastructure management is saved in LDAP. Apps that are not LDAP-aware can use this data nonetheless by registering handlers that trigger when certain data is changed (e.g. a user is created, the IP of a computer is changed). This may be useful if

1. Your software contains some kind of user authentication/authorization, but cannot connect to LDAP
2. Your software has its own database and the data should be in sync
3. Your software needs to reconfigure as soon as certain parameters of the network topology change

More details can be found in the Developer Reference[3].

A short example how to sync first name, last name, email of a user to a (theoretical) third-party database. This script is run every time a user is added, removed or any of these attributes change. As the email is unique (forced by UCS) and all three attributes are only single-valued (also forced by UCS), this may come down to:

```
name = "app_sync_users"
description = "always be in sync with UCS users"
filter = "(&(uid=*)(!(uid=*$))"
attributes = ["givenName", "sn", "mailPrimaryAddress"]

def handler(dn, new, old):
 if new and not old:
  add_user(new)
 elif not new and old:
  remove_user(old)
 elif new and old:
  modify_user(new, old)

def add_user(new):
 new_mailPrimaryAddress = new.get('mailPrimaryAddress', [''])[0]
 new_givenName = new.get('givenName', [''])[0]
 new_sn = new.get('sn', [''])[0]
 get_db_connection().add(new_givenName, new_sn, new_mailPrimaryAddress)

def remove_user(old):
 old_mailPrimaryAddress = old.get('mailPrimaryAddress', [''])[0]
 get_db_connection().remove(old_mailPrimaryAddress)

def modify_user(new, old):
 old_mailPrimaryAddress = old.get('mailPrimaryAddress', [''])[0]
 new_mailPrimaryAddress = new.get('mailPrimaryAddress', [''])[0]
 new_givenName = new.get('givenName', [''])[0]
 new_sn = new.get('sn', [''])[0]
 get_db_connection().modify(old_mailPrimaryAddress, new_givenName,
  new_sn, new_mailPrimaryAddress)
```

[3] https://docs.software-univention.de/developer-reference.html#chap:listener

```
def get_db_connection():
 raise NotImplementedError()
```

4.2. Read configuration database

Feedback

UCS ships with a key-value store used to save parameters of the environment, the Univention Configuration Registry (UCR). It holds information about the local host (like hostname or network settings) and to some extent about the domain configuration (like the domain name or where the DC Master can be found). More details can be found in the Developer Reference[4].

The values can be accessed easily in a script by using

```
hostname=$(ucr get hostname)
```

Notable variables include:

- `hostname`
- `domainname`
- `ldap/base`
- `ldap/master` (FQDN of the DC Master)
- `ldap/master/port` (Port for LDAP bind)
- `ldap/hostdn` (May be useful to connect to LDAP with the machine account (password in `/etc/machine.secret`))

You can use the database to store your own keys and values and use those in your script:

```
ucr set myapp/loglevel=5
```

It is also possible to set the variable only if it was not set before. This is generally preferred as it allows users to overwrite those values without having to fear that it is overwritten again.

```
ucr set myapp/loglevel?4
```

You do not need to register those variables anywhere, they are just saved. It is also possible to use theses variables in your installations scripts as environment variables, for example:

```
eval "$(ucr shell)"
echo "This UCS system has the FQDN $hostname.$domainname "\
"and the LDAP base is $ldap_base."
```

A very powerful ability of UCR is its usage in templates. You may ship files that are recreated when certain variables change. For example, your app's configuration file needs to be updated every time a locale of the system is added or removed. Say your main package (***myapp.deb***) ships `/etc/myapp.conf`:

```
# configuration of myapp
title=My App
locales=en_US.UTF-8:UTF-8
```

Your integration package (***univention-myapp.deb***) can ship this file, too:

```
@%@UCRWARNING=#@%@
# configuration of myapp
```

[4] https://docs.software-univention.de/developer-reference.html#chap:ucr

```
title=My App
locales=@%@locale@%@
```

This file shall trigger each time

```
ucr set locale=...
```

is called.

You need to add the file above in ***univention-myapp***'s directory at `conffiles/etc/myapp.conf`. Furthermore you need to add the following in `debian/rules`:

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

%:
 dh $@
```

Now you need to tell the system when to recreate it. For this, create a file `debian/univention-myapp.univention-config-registry` with

```
Type: file
File: etc/myapp.conf
Variables: locale
```

And this should do the trick. Templates can even use a Python runtime to do more than just writing the exact content of certain UCR variables. See the Developer Reference for details.

4.3. Domain-Join and Unjoin

Feedback

Integration into the UCS domain works by writing into the domain wide LDAP directory. The package can only change something in the LDAP directory through a join script, otherwise the functionality is not guaranteed. Furthermore, the hostname and other basic configuration settings are first defined when the join script is executed.

A join script is just an executable script living in `/usr/lib/univention-install/`. The name is something like *xx*`$app.inst` (*xx* are two digits for ordering purposes). The file must have the executable permission bits set.

Join scripts are commonly used to (but of course not limited to):

- Create users, groups, etc, as well as modifying existing ones
- Registering an LDAP schema extension
- Extending the form for creating/modifying a user (or a computer, ...) by Extended Attributes
- Adding a service entry to the local host
- Configuring the app with parameters read from LDAP

Join scripts are normally run as `root`.

This example shows how to register a schema extension as well as adding widgets to the user form.

```
root@master:~# cat /usr/lib/univention-install/50app.inst
#!/bin/bash
```

```
# VERSION has to be set for external programs to parse
# join scripts will in general onyl be run once per VERSION
# so you need to increment this value when you are changing the script
VERSION="1"

. /usr/share/univention-lib/ldap.sh
. /usr/share/univention-join/joinscripthelper.lib

# this function of joinscripthelper.lib initializes some important
# variables as well as aborting if this script has already been run
joinscript_init

eval "$(ucr shell)"
SERVICE="My App"
APP="app"

# "$@" is IMPORTANT, because this includes parameters for LDAP bind
# Otherwise these functions will fail on systems != DC master
# An example schema file is in the section "Extend the UCS management
# system"
ucs_registerLDAPExtension "$@" --schema "/usr/share/$APP/$APP.schema"

# create a container where the extended attributes shall live
udm container/cn create "$@" \
 --ignore_exists \
 --position "cn=custom attributes,cn=univention,$ldap_base" \
 --set name="$APP" || die # if this fails, abort join script

# for more details, see the section "Extend the UCS management system"
udm settings/extended_attribute create "$@" \
 --ignore_exists \
 --position "cn=$APP,cn=custom attributes,cn=univention,$ldap_base" \
 --set module="users/user" \
 `# ...` \
 --set name="$APP-enabled" || die

# Best practice: Adds the service to the host. Then LDAP can be queried
# to lookup where the app is already installed. Also useful for unjoin
ucs_addServiceToLocalhost "${SERVICE}" "$@"

# when everything worked fine, tell the system that this VERSION does
# not need to be run again
joinscript_save_current_version
exit 0
```

An unjoin script is more or less the same, except that it lives in `/usr/lib/univention-uninstall/` (and ends with `.uinst`). Its purpose is to be called after the app is uninstalled. After uninstallation, it might be appropriate to clean up those objects that have been added in the join script. Keep in mind that the app may be installed on different servers in the domain. So one must take care to not delete important objects when another host is still running this service.

```
root@master:~# cat /usr/lib/univention-uninstall/50app-uninstall.uinst
#!/bin/bash
VERSION=1
. /usr/share/univention-lib/ldap.sh
```

```
. /usr/share/univention-join/joinscripthelper.lib
joinscript_init

eval "$(ucr shell)"
SERVICE="My App"
APP="app"

# revert ucs_addServiceToLocalhost
ucs_removeServiceFromLocalhost "${SERVICE}" "$@"

# check whether this app is still installed elsewhere
if ucs_isServiceUnused "${SERVICE}" "$@"; then
 # revert other changes made by 50app.inst
 # just remove the container, the extended attribute is removed
 # automatically
 udm container/cn remove --dn \
  "cn=$APP,cn=custom attributes,cn=univention,$ldap_base"

 # DO NOT revert ucs_registerLDAPExtension "$@" --schema
 # schema extensions should be kept forever. If attributes defined
 # there were set during the time the app was installed
 # it may break LDAP if the attribute definition gets removed!
 # See http://sdb.univention.de/1274
fi

# revert joinscript_save_current_version - so that the join script
# would run again if the app is reinstalled
joinscript_remove_script_from_status_file app

exit 0
```

Now the scripts need to be packaged. Some steps have to be done in the `postinst`, `prerm`, `postrm` files of the package. There is a helper script that does that automatically. In `debian/rules`, add

```
override_dh_auto_install:
 univention-install-joinscript
 dh_auto_install

%:
 dh $@
```

The join script needs to lie in the root directory of the source code and has to be named after the package, e.g. `50univention-myapp.inst` and `50univention-myapp-uninstall.uinst`. If you need more control, just do not `univention-install-joinscript`, details what to do can be found in the Developer Reference[5].

4.4. Extend the UCS management system

Feedback

4.4.1. Add tabs and options

Feedback

The form for creating LDAP objects can be customized by apps. Technically this is done by writing special objects into LDAP. As such, customization is generally done in a join script. The objects are created with the Univention Directory Manager (UDM).

[5] https://docs.software-univention.de/developer-reference.html#join:write

This example creates a checkbox in the users' form's tab "Advanced settings". This makes it possible to save whether the user should be allowed to use the app. The value has to be queried by the app afterwards.

```
APP="myapp"
SERVICE="My App"
# for more details, see
#   https://docs.software-univention.de/developer-reference.html#udm:ea
# "$@" is here because this should go into a join script and there
# passing the arguments of the script invocation to udm is necessary
udm settings/extended_attribute create "$@" \
 --ignore_exists \
 --position "cn=$APP,cn=custom attributes,cn=univention,$ldap_base" \
 --set module="users/user" `# extending users` \
 --set ldapMapping="${APP}Enabled" `# LDAP attribute from the schema` \
 --set objectClass="${APP}-user" \
 --set name="$APP-enabled" `# this is the name for UDM` \
 --set shortDescription="Allow $SERVICE" \
 --set longDescription="Whether this user shall be allowed ..." \
 --set translationShortDescription="\"de_DE\" \"$SERVICE erlauben\"" \
 --set translationLongDescription="\"de_DE\" \"Zeigt an, ob ...\"" \
 --set tabName="$SERVICE" `# This may create a new tab in the form` \
 --set translationTabName="\"de_DE\" \"$SERVICE\"" \
 --set tabAdvanced='0' \
 --set tabPosition='1' \
 --set syntax='TrueFalseUp' `# should be a CheckBox` \
 --set mayChange='1' \
 --set default='TRUE' || die
```

Note the

```
--set syntax='TrueFalseUp'
```

which semantically turns this attribute into a boolean field. Other syntax definitions exist, for example `string` or `ipAddress`. More examples can be found in the following file `/usr/share/pyshared/univention/admin/syntax.py`.

It is also possible to create own drop downs. The following example adds a combo box with two options "`Admin`" or "`User`"

```
udm settings/extended_attribute create "$@" \
 --ignore_exists \
 --position "cn=$APP,cn=custom attributes,cn=univention,$ldap_base" \
 --set module="users/user" \
 --set ldapMapping="${APP}Role" \
 --set objectClass="${APP}-user" \
 --set name="$APP-role" \
 --set shortDescription="Role in $SERVICE" \
 --set longDescription="Which role the user has for $SERVICE" \
 --set translationShortDescription="\"de_DE\" \"$SERVICE-Rolle\"" \
 --set translationLongDescription="\"de_DE\" \"Welche Rolle ...\"" \
 --set tabName="$SERVICE" \
 --set translationTabName="\"de_DE\" \"$SERVICE\"" \
 --set tabAdvanced='1' \
 --set tabPosition='1' \
 --set syntax="${APP}UserOrAdmin" \
 --set mayChange='1' \
```

```
 --set default='user' || die
```

The syntax is a Python class and needs to be defined in a separate file:

```
class myappUserOrAdmin(select):
 choices=[('user', 'User'), ('admin', 'Admin')]
```

This file needs to be registered in a join script:

```
ucs_registerLDAPExtension "$@" \
 --udm_syntax "/usr/share/$APP/${APP}_syntax.py"
```

4.4.2. LDAP schema extension

Feedback

The Extended Attributes are generally stored in LDAP as attributes not defined by default. A schema file needs to be created and registered for the Extended Attributes to actually work. See this section[6] for details of how to write a schema file.

The example above needs a file like this:

```
attributetype ( 1.3.6.1.4.1.10176.99998.xxx.1.1 NAME 'myapp-enabled'
 DESC 'My App allowed'
 EQUALITY caseIgnoreMatch
 SUBSTR caseIgnoreSubstringsMatch
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE
 )
attributetype ( 1.3.6.1.4.1.10176.99998.xxx.1.2 NAME 'myapp-role'
 DESC 'My App role'
 EQUALITY caseIgnoreMatch
 SUBSTR caseIgnoreSubstringsMatch
 SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE
 )
objectclass ( 1.3.6.1.4.1.10176.99998.xxx.0.1 NAME 'myapp-user'
 DESC 'My App user'
 SUP top AUXILIARY
 MUST ( cn )
 MAY ( myapp-enabled $ myapp-role )
 )
```

Note the "*xxx*" in the so called OIDs. You need a unique (worldwide!) identifier for your attributes and object classes. Either request one[7] or (probably better) talk to us, as Univention has its own namespace and any schema extension can be defined "beneath" it.

This file needs to be registered in a join script:

```
ucs_registerLDAPExtension "$@" --schema "/usr/share/$APP/$APP.schema"
```

4.4.3. Build own UDM modules

Feedback

The Univention Directory Manager (UDM) is a collection of modules written in Python to add powerful capabilities around Python's LDAP bindings. In general, vendors will extend those existing modules using Extended Attributes. But if a completely new class of object shall be saved, a new UDM module may be useful, e.g., if the app manages buildings which cannot be simply "extended groups or containers".

[6] http://www.openldap.org/doc/admin23/schema.html
[7] http://pen.iana.org/pen/PenApplication.page

When writing a UDM module, it is best to look out for existing modules that can be copied and customized for one's own needs. They live in `/usr/share/pyshared/univention/admin/handlers/`. The module `appcenter/app.py` is a rather simple module which shows how a basic module should look like:

```
svn co https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/\
management/univention-management-console-module-appcenter/\
udm/handlers/appcenter/
```

4.4.4. Build own UMC module

Feedback

The Univention Management Console (UMC) is the web based frontend which is used to administrate the UCS domain. It consists of separate modules and vendors may write such modules to further extend the Console. This may be a good idea if

- The app can be customized but currently lacks a frontend
- The app needs to be activated or manually configured to work properly

Note that many UCS users are used to UMC and the fact that everything can be configured in one place. So adding a UMC module may greatly enhance the user experience.

UMC modules are very versatile (both the JavaScript based frontend part as well as the Python backend part) and can be used for nearly anything. This guide cannot cover everything there is about UMC modules. One starting point may be

```
univention-install univention-management-console-dev
umc-create-module --help
```

Or you use UMC and look out for modules that to some extend do what you are trying to accomplish and copy the source code, e.g.

```
svn co https://forge.univention.org/svn/dev/branches/ucs-4.1/ucs-4.1-4/\
/management/univention-management-console-module-top
```

4.5. Further integration scenarios

Feedback

4.5.1. Firewall settings

Feedback

In the default setting, all incoming ports are blocked by the UCS firewall. Every package can provide rules, which free up the ports required. In this example the port 6644 is opened for TCP and UDP. It can be run in the `postinst` script or in the join script:

```
# configure firewall
univention-config-registry set \
 security/packetfilter/package/"$APP"/tcp/6644/all="ACCEPT" \
 security/packetfilter/package/"$APP"/tcp/6644/all/en="$APP" \
 security/packetfilter/package/"$APP"/udp/6644/all="ACCEPT" \
 security/packetfilter/package/"$APP"/udp/6644/all/en="$APP"
[ -x "/etc/init.d/univention-firewall" ] &&
 invoke-rc.d univention-firewall restart
```

4.5.2. Serving a web application

Feedback

UCS comes with a running Apache HTTP Server used by the UMC server as a proxy. This means that apps cannot use port 80/443 easily: It is already used. Apps can use Apache, though, by shipping a file `/etc/`

`apache2/sites-available/$APP`. Apache can then act as a proxy to the app's server (running on a different port).

```
# minimal
ProxyPass /$APP/ http://127.0.0.1:$APP_PORT/
ProxyPassReverse /$APP/ http://127.0.0.1:$APP_PORT/
```

The site needs to be enabled by a line in the `postinst` of the package:

```
a2ensite "$APP"
```

It is highly recommended to use Apache for it is the service with port 80/443. While it is possible to just let the app respond to requests on port say, 8080, many firewalls will block the app without taking further actions. One prominent example are the default security rules of the Amazon Web Services. The app may not be accessible without using Apache as a proxy!

4.5.3. Setting links to the web interface in `/ucs-overview`

The start page of any UCS system (`http://$hostname/`) lists available services on this server, notably UMC. If an app provides a web interface, this can be listed, too. The easiest way is by stating this in the `ini` file:

```
WebInterface=/$APP/
#WebInterfaceName=... # defaults to Name=
# one of the two categories in /ucs-overview.
#   "services" (default) or "admin"
#UCSOverviewCategory=services
```

If *WebInterface* is given in the `ini`, the App Center takes care of the integration on the overview site. But sometimes this is not powerful enough. This level of "automated integration" cannot handle ports other than 80/443 (as it will always use the current port which is 80 or 443) and cannot add more than one link. If a deeper level is required, this should be done in the `postinst` and `postrm` scripts of the integration package using UCR:

```
# postinst

#DEBHELPER#

# ucs/web/overview/entries/service/... or
# ucs/web/overview/entries/admin/...

export P="ucs/web/overview/entries/service"
ucr set \
 "$P/$APP"/description/de="Description of link to $APP (German)" \
 "$P/$APP"/description="Description of link to $APP (English)" \
 "$P/$APP"/icon="/url/to/icon/$APP.png" \
 "$P/$APP"/label/de="Headline of link to $APP (German)" \
 "$P/$APP"/label="Headline of link to $APP (English)" \
 "$P/$APP"/link="https://$hostname.$domainname:$APP_PORT/webinterface/"
 \
 "$P/$APP"/priority=xx-digits-for-sorting-or-just-dont-set
```

```
# postrm

#DEBHELPER#
```

```
ucr unset \
 ucs/web/overview/entries/service/"$APP"/description/de \
 ucs/web/overview/entries/service/"$APP"/description \
 ucs/web/overview/entries/service/"$APP"/icon \
 ucs/web/overview/entries/service/"$APP"/label/de \
 ucs/web/overview/entries/service/"$APP"/label \
 ucs/web/overview/entries/service/"$APP"/link \
 ucs/web/overview/entries/service/"$APP"/priority
```

4.5.4. Using PostgreSQL or MySQL

Feedback

When your application uses PostgreSQL, your package should depend on ***univention-postgres*** and you need to ship a file in `/etc/postgresql/9.1/main/pg_hba.conf.d/` or, maybe even better, in `/etc/univention/templates/files/etc/postgresql/9.1/main/pg_hba.conf.d/` (see UCR):

```
local $app_db_name $app_db_user md5
```

When your application uses MySQL, you may access the administrator password by reading `/etc/mysql.secret`. A dependency on the package ***mysql-server*** is enough as we patch the Debian package.

Chapter 5. Provide the App

Until now you should have your software solution packaged as Debian package(s) including a separate package taking care of the UCS integration. To finish the app, you'll need to add the meta data for the App Center and upload it to Univention.

> **Note**
>
> Starting with UCS 4.0, only 64 bit installation images are provided. Univention does support 32 bit for at least UCS 4.x, though. When using a 32 bit UCS 3.2, one may update to UCS 4.0. It is therefore recommended (but not required) to provide the app for amd64 and i386. If i386 shall not be supported, one may specify `SupportedArchitectures=amd64` in the ini file, see Section 5.1.

For the archive to be uploaded, the following directory structure is recommended:

- `metadata/`
- `packages/`
  - `all/`
  - `amd64/`
  - `i386/`
- `readme`

Put your packages in the appropriate subdirectories below `packages/`.

5.1. Create the App meta data

The Debian packages take care of the installation of your software solution on UCS through the Debian package manager. But, the App Center does not know what to present to the user. This gap is filled with the App meta data comprising of text information like description, website, contact, visual information like a logo and optional screenshots, optional detailed information for the users in several readme files.

Please provide the following information together with packages:

1. A text file in the `.ini` format including information like description, several website links, contact information, conflicting apps, etc. Please refer to the Developer Reference[1] for a template and the description of every attribute.

2. A product logo in SVG format, ratio: square.

The `.ini` file has the attribute *ID*. Simply name the `.ini` file and the product logo after that ID (note that you need to specify the logo file in the ini file!):

- `myapp.ini`
- `myapp.svg`

[1] https://docs.software-univention.de/developer-reference.html#app:iniFile

Put those files below the `metadata/` directory.

Note

The meta data contains the attributes *NotifyVendor* and *NotificationEmail*. If you want to receive daily email notifications upon the installation of your app, please set them appropriately. The email address here may differ from the contact address. If set to `True` the user will be informed about the delivery of such a notification before installing the app.

Note

Note to sales: You as independent software vendor are responsible for the contacts and it is up to you how to organize the follow-up. Try to contact the users very fast. The experience shows that it makes sense to organize a follow up within one week.

A detailed explanation about the notifications can be found in the Developer Reference[2].

5.2. Create optional App meta information

Feedback

You may add optional app meta data information. Please refer to the Optional application files section in the developer reference for the choice of files.

1. Another product logo in SVG format. This time, no ratio requirements. It will be shown in the details of the app rather than in the overview.

2. Screenshot of your solution: The filename of the screenshot is given in the app meta data with the attribute *Screenshot*. Recommended name: *myapp*`_screenshot.png` (you may also provide a `jpg`).

3. License agreement: If you require the user to confirm a license agreement before installation, provide the file `LICENSE_AGREEMENT_EN` and `LICENSE_AGREEMENT_DE`.

4. `README` files: You may add different readme files depending on action taking place. For example, you may present text before installation or before update only. Please refer to the Optional application files section[3] for the set of possible readme files.

Please use simple HTML in all those files and split the text into reasonable paragraphs. Copy the files below the readme directory in the recommended structure. The file names have to match the definitions.

5.3. Upload the App

Feedback

Finally, upload the whole app according to the following steps:

1. Take the above directory structure, create an archive, for example `tar.gz` or `zip`.

2. Upload the archive to https://upload.univention.de/ and remember the upload-id shown there.

3. Send the upload-id via email to `<appcenter@univention.de>`.

Congratulations! That's it, you are finished for the moment.

[2] https://docs.software-univention.de/developer-reference.html#app:notification
[3] https://docs.software-univention.de/developer-reference.html#app:optionalFiles

Chapter 6. What happens next?

After you sent the upload-id to Univention, a Univention employee will extract your files and copy them to the Test App Center. You'll receive a short note about how to activate the Test App Center for final testing of your app. After the app passed the automatic tests at Univention concerning the packages and you as vendor gave your written approval, the app will be published in the App Center. Due to feedback and further communication the approval process may last a couple of days. Further uploads may be necessary to fix issue that have been found during this iterating process.

Note

The automatic tests run after the packages have been copied into the Test App Center, only cover UCS core functionality, e.g., whether LDAP still works after adding a schema file. They do not test the App itself.

Chapter 7. Docker Apps for the Univention App Center

Starting with UCS 4.1 the Univention App Center will support Docker. Docker is an OS level virtualization software that helps deploying applications in isolated containers. In this chapter we will discuss the reasoning and aim of the Docker support, give some technical insights and also go through example apps.

7.1. Why Docker?

Prior to UCS 4.1, Apps in the Univention App Center were installed next to all other system packages. This made the development of enterprise applications for UCS fairly easy but had some shortcomings:

- Some applications required newer versions of certain packages like PHP. This had impact on the stability of the operating system and also led to (not obvious) conflicts between apps.
- With a growing number of apps in the App Center catalog it got harder for Univention to verify that the App did not break anything by enabling/disabling features of certain software components. With the Debian Maintainer scripts, App vendors had effectively root access on the system.

By supporting Docker, we aim to overcome those points while preserving the simplicity of developing an App.

If you have already worked with Docker, you may know that Docker is sometimes advertised as a way to encapsulate each and every task into its own container, sometimes referred to as *Micro services*. In the Univention App Center we currently focus on a different route: We start minimal, yet fully functional UCS systems in which one App is installed and started.

Important

For App development this means that an ISV still programs against a UCS. There is no need to build a dedicated Docker Container. Univention already provides such a container along with the tools to actually install Debian packages just like the App would be installed on metal (or just like it was prior to UCS 4.1).

An ISV needs to provide Debian packages, not Docker images (very much the same as in UCS < 4.1). The integration of the App into the UCS infrastructure may be done via an additional package or via separate scripts that live unpackaged on the App Center server (see below for an example).

7.2. Example: Docker App Radicale

Feedback

In this section we will develop a Docker based App step by step.

We use the software Radicale for it. Radicale is a CardDAV and CalDAV server already packaged in Debian and therefore available in UCS. So we do not really develop something from scratch but use an existing application and put it into the Univention App Center.

Note

In general, apps need to come in the Debian package format (`.deb`). If you do not have your application packaged in this way, you may want to check the chapter packaging[1] in the UCS developer reference as a starting point. Here we will use a pre-packaged application already present in (the unmaintained repository of) UCS.

7.2.1. Prerequisites

Feedback

You need a running UCS 4.1 system (master domain controller for the sake of convenience). As Radicale will be installed from the UCS repository, not from the App Center and Radicale is unmaintained by Univention, you need to activate the unmaintained repository:

```
ucr set repository/online/unmaintained='yes'
```

Also, make sure your system has the following packages installed: ***univention-appcenter***, ***univention-appcenter-docker***, ***univention-appcenter-dev***

After this has been done, run

```
univention-app dev-setup-local-appcenter
```

7.2.2. The ini file

Feedback

An App consists of the actual packages and meta information about the App. The App *may* ship one or more screenshots, various `README` files and so on. But every App *has* to have an `ini` file.

Important

Every version of an App (i.e. the original upload and every update of it) has its own `ini` file! It may be just a copy of the original file with the `Version=` increased, yet a new ini file is necessary.

Create a file `~/radicale.ini` (name it as you like...) and put the very first content in it:

```
[Application]
# Can be chosen arbitrarily but after it has been chosen,
# needs to stay the same in each version
ID=radicale
# Code is necessary, but given by Univention. Normally, you do not
# need to specify it
Code=RD
Name=Radicale
```

The version will be set to that of the repository:

```
Version=0.7
```

[1] https://docs.software-univention.de/developer-reference.html#chap:packaging

Radicale is developed by *Kozea*, we are "only" doing the integration work. If you develop the software and do the integration, you only need the first two lines:

```
Vendor=Kozea
WebsiteVendor=http://kozea.fr/
Maintainer=Univention
WebsiteMaintainer=http://www.univention.de/
```

Radicale can be used to manage shared calendars, so this makes it a Collaboration App:

```
Categories=Collaboration
```

Note

The following categories are allowed (multiple can be specified, separated by ","): Administration, Business, Collaboration, Education, System services, *Virtualization* (+ UCS Components, but these are meant for Univention Software)

Now the important parts: The Debian package that is to be installed is called ***radicale***. It will live in a repository created for this App on the App Center server. The package is unmaintained by Univention, so it needs to be copied from the unmaintained repository to the App's repository (this is done later).

```
DefaultPackages=radicale
```

It shall be installed as a Docker App, so the following needs to be specified:

```
DockerImage=docker.software-univention.de/ucs-appbox-amd64:4.1-0
```

This means that the package is not installed on your system but rather in a Docker Container running on your system. To access the relevant bits of the container (i.e. the calendars and contacts), you need to forward the port (which happens to be 5232 for Radicale):

```
PortsExclusive=5232
# You could also use PortsRedirection=1234:5232
```

Note

Univention provides a set of images at `docker.software-univention.de`, at least one per minor version of UCS. If unsure, it is probably a good idea to have the image's UCS version match the UCS version of the Docker Host. But this is absolutely not mandatory. In fact, it is one of the advantages of Docker apps that their OS may differ from the host's. For a list of the images, ask Univention.

Together, this makes:

```
[Application]
ID=radicale
Code=RD
Name=Radicale

Version=0.7

Vendor=Kozea
WebsiteVendor=http://kozea.fr/
Maintainer=Univention
WebsiteMaintainer=http://www.univention.de/

Categories=Collaboration
```

```
DefaultPackages=radicale

DockerImage=docker.software-univention.de/ucs-appbox-amd64:4.1-0

PortsExclusive=5232
```

Finally, put the ini file into the (local) App Center (do not forget to upload the Debian packages - here this means two unmaintained packages already built):

```
univention-app dev-populate-appcenter --new \
 --ini ~/radicale.ini \
 --unmaintained radicale python-radicale
```

7.2.3. Integration, first iteration

Feedback

Radicale should already be installable. Alas, it will not work. This is due to the way Radicale is packaged in Debian. To make it work as expected, we need to modify it. Here starts the integration work of the App.

First of all, Radicale is not started by default. To change that, we need to modify `/etc/default/radicale` and uncomment `#ENABLE_RADICALE`. The Univention App Center supports various scripts that can be added to the App and will be executed at various points in time. Maybe the most important one is the join script.

7.2.3.1. The join script

Feedback

The join script is a fundamental feature of UCS. UCS is used to run and administrate a domain. New computers may "join" the domain. The computer searches for the Domain Controller Master (DC Master) and adds itself to LDAP (hostname, IP address, etc). Join scripts are used to "join software packages" into the domain. This means that if you install, say, ***radicale***, it may need to register Radicale somewhere and make some changes in the domain.

The domain is administrated by manipulating the core database on the DC Master, the LDAP database. Normally, this is done by using tools provided by Univention, mainly the Univention Directory Manager, `udm`.

But here, we do not really need to alter LDAP. We just want to change a local file. We are just making use of the fact that the join script is executed after the package is installed. (In terms of the App Center: after the App is installed)

> **Note**
>
> We need to distinguish between the *Docker Host* - this is the "real" UCS installed. The Docker Host is running the *Docker Container*s. These may also be UCS systems, but they essentially only run the App packages.

The App Center provides an easy way to add a join script to the App by just adding it to the repository on the App Center server. The join script is then executed on the *Docker Host* after the *Docker Container* is set up. So the join script is not executed locally with respect to the App, but the Docker Host may access the file system of the Container anyway.

To add a join script, just create a file `~/radicale.inst` and add the following to it:

```
#!/bin/sh

VERSION=1

. /usr/share/univention-appcenter/joinscripthelper.sh
```

```
joinscript_init

eval "$(ucr shell ldap/base)"

joinscript_run_in_container sed -i /etc/default/radicale -e "s/
#ENABLE_RADICALE/ENABLE_RADICALE/" || die

joinscript_save_current_version

exit 0
```

Now you can add it to the App Center:

```
univention-app dev-populate-appcenter --new \
 --ini ~/radicale.ini \
 --join ~/radicale.inst \
 --unmaintained radicale python-radicale
```

Note

```
univention-app dev-populate-appcenter --new
```

will create a new version of the App and write the internal component to the screen (In this case something like `radicale_`*`yyyymmdd`*). But if you are fast enough, it will overwrite the existing App version because the date does not (yet) differ. Correct would be something like

```
univention-app dev-populate-appcenter \
 --component radicale_20150929 \
 --join ~/radicale.inst
```

to really alter the App.

7.2.3.2. Storing the data persistently

Now the App should be installable and run as expected. As specified in the ini `ucs-app-box-amd64:4.1-0` will be used as the Docker image. It is downloaded from a docker registry set up at Univention and started on the Docker Host. The image contains a minimal UCS member server which will eventually contain the App packages.

Important

There are some things you should be aware of when developing a Docker App for the Univention App Center:

- The default image is a minimal, yet fully functional UCS.
- The system will join into the domain. The Docker App will be listed as a member server when showing all hosts of the domain.
- The system will not run something like `/usr/bin/radicale` directly. Instead it runs `/sbin/init` (it is actually a slightly altered version of the original `/sbin/init` of UCS). The App will be run because it is somehow configured to be started on a certain run-level (somehow means: This is the job of the corresponding Debian package).
- The Docker image is writable, i.e. the App can create and modify all files it wants and after restarting the container, the files persist. This also holds for updates: The Docker Container may install

package updates released by Univention (so called errata updates) and even new major versions of the operating system. This means that the image can be used "forever".

Last point for this section will be storing and restoring data from the Docker Container. Although the container may be used forever, it may be that the underlying image needs to be exchanged. This is done by effectively removing the old container and setting up a completely new one. Thus, we need to store the App data just before removing the old container and restore it in the new one.

Radicale has some backends where to store the data, but preconfigured is the file backend - which makes it very easy for us to backup the data. The calender and contact data is stored at `/var/lib/radicale/collections/`. We just need to save this directory and restore it accordingly. Luckily, the Univention App Center provides a shared directory where the App can store its data easily. This directory is `/var/lib/univention-appcenter/apps/$APPID/data/`.

Create the file `~/radicale.store_data` with the following content:

```
#!/bin/sh

/usr/share/univention-docker-container-mode/restore_data_after_setup \
 "$@"

cp -r /var/lib/radicale/collections \
 /var/lib/appcenter/app/radicale/data/

exit 0
```

Next, create `~/radicale.restore_data`:

```
#!/bin/sh

/usr/share/univention-docker-container-mode/restore_data_after_setup \
 "$@"

rm -r /var/lib/radicale/collections
cp -r /var/lib/appcenter/app/radicale/data/collections \
 /var/lib/radicale/

exit 0
```

Important

`/var/lib/univention-appcenter/apps/$APPID/data/` is always mounted into the Docker Container. The very same directory exists on the Docker Host. If your App can be configured to save its data on a different location, you may want to consider using this directory. Not only will it make storing and restoring data in the image exchange process very easy, it should also be faster because it does not rely on the Docker storage driver, thereby reducing overhead. In fact, Radicale should have been configured to store its collection data there in the first place instead of copying it in `store_data`. But this would render the script useless and it shall be part of the tutorial!

Finally, make these scripts known to the App by specifying them in the ini file. You can use any name you like. The scripts will be copied to that place in the container. You may even overwrite existing files.

```
DockerScriptStoreData=/usr/share/univention-appcenter/app/radicale/
store_data
DockerScriptRestoreDataAfterSetup=/usr/share/univention-appcenter/app/
radicale/restore_data
```

Note

All Docker scripts that may be specified in the ini file have a reasonable default already installed in the default container. So it is generally a good idea to execute the default script in your script.

Warning

When using custom scripts (like `store_data`) you should also set the corresponding `DockerScript` variable in the ini file. The default of this variable is the path to the default script. If you want to run that default script in your custom script (as advised) you will instead call yourself!

Add the new scripts to your App Center:

```
univention-app dev-populate-appcenter --new \
 --ini ~/radicale.ini \
 --join ~/radicale.inst \
 --store-data ~/radicale.store_data \
 --restore-data-after-setup ~/radicale.restore_data \
 --unmaintained radicale python-radicale
```

7.2.4. Integration, second iteration

Feedback

The App should be in a functional state now.

```
univention-app install radicale
```

should give you a running CalDAV/CardDAV service at port 5232 on your Docker Host.

The configuration of the service is not optimal, though. Every user, even an anonymous user can create and change any calender and contact. We want to limit the access to domain users. And they should only be able to change their own calendars. Note that Radicale provides mechanisms for even finer grained control. This is not scope of this example, though.

7.2.4.1. Making the App configurable by the user

Feedback

Radicale's rights management can be configured in various ways. We want to let the administrator decide whether the App's configuration is `owner_write` (users need to have valid credentials; their own calenders/contacts can be modified, others' can be read) or `authenticated` (same, but others' can also be modified).

This is achieved by adding `~/radicale.ucr`. This file defines some Univention Configuration Registry variables (UCR). These are a core feature of UCS and thus of a Docker App based on UCS.

```
[radicale/rights/type]
Description[de]=Rechtevergabe für Kalender, Kontakte
Description[en]=Access control for calendars, contacts
Type=list
Labels[de]="Alle lesen, eigene schreiben" "Alle lesen, alle schreiben"
Labels[en]="Read all, write own" "Read all, write all"
Values=owner_write authenticated
Default=owner_write
Categories=apps
```

The definition itself is not very useful on its own. It just sets the variable in the database. The interesting part of UCR are the triggers and the templates. One may define triggers that are executed whenever a variable is changed. And one may define templates that overwrite existing files depending on UCR variables.

Normally, one would now create a package with all the triggers. But for this example, we do not want to create packages at all. We may abuse the join script to add this trigger and a template in the container.

> **Note**
>
> The integration part starts getting complicated. One should really create a package ***univention-radicale*** and add the UCR definition there. One could also do the modification of `/etc/default/radicale` there, e.g. by using `dpkg-divert`. The package would depend on exactly one package (***radicale***) and be defined as the only `DefaultPackages` in the ini file.

We will show what to do in the next section as the join script is modified one more time.

7.2.4.2. Making the App LDAP aware

Radicale comes with LDAP support. We just need to configure it. Radicale binds to LDAP and can then check any credentials. But for that we need a `binddn` for Radicale. A new user for that App needs to be created in LDAP.

After that this very user has to be specified in the configuration file of Radicale, `/etc/radicale/conf`.

This should be done in the join script. We change `~/radicale.inst` so that it adds a new user for us and creates a file in the container. Note that the latter operation is quick and dirty and is only done because we would have to create a package for basically one file otherwise.

```
#!/bin/sh

VERSION=1

. /usr/share/univention-appcenter/joinscripthelper.sh
. /usr/share/univention-lib/all.sh

joinscript_init

eval "$(ucr shell ldap/base)"

ucs_addServiceToLocalhost "${SERVICE}" "$@"

joinscript_add_simple_app_system_user "$@"

cat > "$(joinscript_container_file_touch /etc/univention/templates/
files/var/lib/radicale/.config/radicale/config)" <<- EOF
@%@UCRWARNING=#@%@

[rights]
type = @%@radicale/rights/type@%@

[auth]
# Access method
# Value: None | htpasswd | LDAP | PAM | courier
type = LDAP

# Usernames used for public collections, separated by a comma
#public_users = public
# Usernames used for private collections, separated by a comma
#private_users = private
```

```
# Htpasswd filename
#htpasswd_filename = /etc/radicale/users
# Htpasswd encryption method
# Value: plain | sha1 | crypt
#htpasswd_encryption = crypt

# LDAP server URL, with protocol and port
ldap_url = ldap://@%@ldap/server/name@%@:@%@ldap/server/port@%@/
# LDAP base path
ldap_base = @%@ldap/base@%@
# LDAP login attribute
#ldap_attribute = uid
# LDAP filter string
# placed as X in a query of the form (&(...)X)
# example: (objectCategory=Person)(objectClass=User)
(memberOf=cn=calenderusers,ou=users,dc=example,dc=org)
# leave empty if no additional filter is needed
#ldap_filter =
# LDAP dn for initial login, used if LDAP server does not allow
 anonymous searches
# Leave empty if searches are anonymous
ldap_binddn = uid=radicale-systemuser,cn=users,@%@ldap/base@%@
# LDAP password for initial login, used with ldap_binddn
@!@
#ldap_password =
print 'ldap_password = %s' % open('/etc/radicale.secret').read()
@!@
# LDAP scope of the search
ldap_scope = SubTree

# PAM group user should be member of
#pam_group_membership =

# Path to the Courier Authdaemon socket
#courier_socket =
EOF

cat > "$(joinscript_container_file /etc/univention/templates/info/
univention-radicale.info)" <<- EOF
Type: file
File: var/lib/radicale/.config/radicale/config
Variables: radicale/.*
Variables: ldap/base
Variables: ldap/server/port
Variables: ldap/server/name
EOF

joinscript_run_in_container ucr update
joinscript_run_in_container ucr commit /var/lib/radicale/.config/
radicale/config

joinscript_run_in_container sed -i /etc/default/radicale -e "s/
#ENABLE_RADICALE/ENABLE_RADICALE/" || die
joinscript_run_in_container invoke-rc.d radicale restart
```

```
joinscript_save_current_version

exit 0
```

Note that `ucs_addServiceToLocalhost` was added. This is a best practice to make the domain administrator aware where the application Radicale is installed.

Important

The command `ucs_addServiceToLocalhost` should be reverted when the App is uninstalled. But even more importantly, the join script needs to be run again as soon as the App is uninstalled and then installed again (changing `/etc/default/radicale` and so on...). As the join script saved the information bit that it was successfully executed at the end of itself, we need to revert that, too. To do so, we need an *unjoin script*. Technically it is similar to a join script but run after uninstalling the App, not after installing it. The script would simple do this (save to `~/radicale.uinst`):

```
#!/bin/sh

VERSION=1

. /usr/share/univention-appcenter/joinscripthelper.sh
. /usr/share/univention-lib/all.sh

joinscript_init

ucs_removeServiceFromLocalhost "${SERVICE}" "$@"

joinscript_remove_script_from_status_file radicale

exit 0
```

One last time we need to update our App Center server:

```
univention-app dev-populate-appcenter --new \
 --ini ~/radicale.ini \
 --join ~/radicale.inst \
 --unjoin ~/radicale.uinst \
 --store-data ~/radicale.store_data \
 --restore-data-after-setup ~/radicale.restore_data \
 --ucr ~/radicale.ucr \
 --unmaintained radicale python-radicale
```

Now we can test and use our application:

```
univention-app install radicale
```

The configuration option regarding the rights management can now be set in the Univention Management Console or via

```
univention-app configure radicale \
 --set radicale/rights/type=authenticated
univention-app restart radicale
```

7.3. Epilogue

Feedback

In this section we will go briefly over things we may have missed in the example.

7.3.1. Docker scripts

For every App, the App Center server holds a repository containing the Debian packages. But it also may hold several scripts that do not need to be packaged. In theory one may integrate one's App without needing to build an additional integration package. That being said, with increasing complexity, it is advised to build such a package nonetheless. Furthermore, not everything can be achieved easily with these scripts. There are interfaces rarely used that the App Center does not support. Should your App require these interfaces, it may be necessary to create a package.

We need to distinguish between *outer scripts* and *inner scripts*. An *outer script* is called on the *Docker Host*, *inner scripts* are called in the running *Docker Container*.

All scripts are called with root privileges. For *inner scripts* this means the local root of the *Docker Container*. Some scripts may get LDAP credentials. Normally these credentials are that of the *Administrator account*.

Note

None of the following scripts is mandatory, all have reasonable defaults / fallbacks. But every scripts may be overridden! Just upload them along with the ini file and Univention will place them on the App Center server. See also Section 7.3.2 on how to specify these scripts in the ini file.

The following scripts are for installation and uninstallation. Upgrades use these scripts, too, if (and only if) the upgrade includes an image exchange (i.e. the new version with the new ini file specifies a different `DockerImage`). In that case the old App is uninstalled and the new App is installed.

`preinst`
: An *outer script* called before the Docker Container is initialized, even before the image is downloaded. Its purpose is to check whether installation may be successful. For example, the preinst may fail if certain hardware requirements are not met. Any exit code other than 0 will result in cancellation of the installation process. If the installation of an App is the result of an image exchange (and thus more of an upgrade) the preinst is also called.

`restore_data_before_setup`
: An *inner script* called before *setup* is called, right after the container is started. Its purpose is to restore those bits that may be needed to successfully run the setup script. May be useful in an upgrade process where one needs to restore the state the old container was in instead of setting up the container as if it were fresh.

`setup`
: An *inner script* and heart of the whole installation process. By default it joins the system, (a script called `univention-join` provided by Univention), adds the repository that was created on the App Center server and installs `DefaultPackages` specified in the ini file. After that it once again joins, running all scripts that may have been installed during the App installation. If the script fails (exit code != 0) the installation is aborted.

`restore_data_after_setup`
: An *inner script* called after *setup* is called. Its purpose is to actually restore the data that may have been saved in *store_data* – now that the App is up and running but the database is still empty.

`inst`
: Also called *join script*. An *outer script* called after the *Docker Container* is configured. Think of it as a postinst of the App. See the Developer Reference[2] for how to write a join script. If the join script runs successfully, the join script may save this information in a status file. If this does not happen, the user is constantly reminded to re-run the join script. So the join script does not need to run successfully. The

[2] https://docs.software-univention.de/developer-reference.html#join:write

installation will not be aborted at this point. But of course at some point it should run through successfully. The main purpose of the join script is to set up the domain for the new App, e.g. by adding a domain user that can be used as the `LDAP binddn` that your App may need to authenticate against the central user management of UCS.

`prerm`
: An *outer script* called before the Docker Container is removed. Its purpose is to check whether an uninstallation may be successful. But it may be used to somehow prepare the system for the uninstallation. For example, the prerm may fail if other software still depends on it. Any exit code other than 0 will result in cancellation of the uninstallation process.

`store_data`
: An *inner script* called before removing the *Docker Container*. In fact the App is not really uninstalled. The container is just thrown away. This also happens during image upgrades: The current image is thrown away and a new image is set up. Therefore it is important to store the data of the current App to be able to start into exactly that state when the container was removed. You may call any App specific commands like `myapp-backup --full` or just copy the important files. The App Center always mounts the following directory into the container, which can be used to store the data (and later restore the data from there as the very same directory will be mounted into the new container): `/var/lib/univention-appcenter/apps/$APPID/data/`.

`uinst`
: Also called *unjoin script*. An *outer script* called after the *Docker Container* is removed. Think of it as a postrm of the App. See the Developer Reference[3] for how to write an unjoin script. It should somehow revert most (if not all) changes done in the *inst* script (or join script). With the notable exception of schema registration. An LDAP schema extension should never be removed once it was registered.

The following scripts are for upgrading the software within the *Docker Container*. Three possible scenarios have to be covered:

- Upgrade of system packages (security fixes, UCS calls them Errata Updates)
- Upgrade of the operating system (a new patchlevel release or even a minor/major update of UCS (4.1-1 or 4.2-0)
- Upgrade of App packages, i.e. a new version (with the same `DockerImage`) was released

`update_available`
: An *inner script* called when the App Center wants to find out whether the container may install package updates or release updates. By default, this is done automatically once a day. It can be triggered manually, too, though. The script needs to `echo` the result:

  - *packages* if mere package updates are available. These include security fixes for operating system packages.
  - *release: RELEASE* if a new version of the operating itself is available. *RELEASE* can be anything, it is just presented to the user.

  Note

  The script does not search for App updates, nor does it search for a change in the `DockerImage`! This is done by the "outer" App Center.

`update_packages`
: An *inner script* called when (all) updates of existing packages shall be installed.

[3] https://docs.software-univention.de/developer-reference.html#join:write

`update_release`
An *inner script* called when a new version of the operating system shall be installed.

`update_app_version`
An *inner script* called when a new version of the App packages shall be installed. This script is different from `update_packages`. This has technical reasons, because for normal UCS based Docker images, this requires a new repository to be registered. But it also has usability reasons: First, users may want to distinguish between necessary, stability improving package updates and (potentially) ground-shaking App updates; second, at least the UCS containers are configured to install security / stability updates automatically.

7.3.2. Docker related variables in the ini file

We used some of the ini's variables to adjust the Docker App to our needs. Here is a brief overview regarding all Docker related variables:

`DockerImage`
The image the container will be based on. The Univention App Center will provide a list of UCS based images that can be used to run the application. Other images may work but are strongly discouraged as they will lack domain functionality that would be required to integrate the App into the UCS domain.

> **Note**
>
> If you want to develop a Docker App, this variable is *required*. The following variables on the other hand have meaningful defaults (or are empty, which is fine, too).

`DockerAllowedImages`
When the App does not rely on a well defined basis, it may be convenient to update `DockerImage` every now and then (e.g. when a new minor version of UCS is released). This is not required, but not doing so would configure an outdated base image that needs to be upgraded after the App installation. This may take a long time depending on the age of the image.

If `DockerImage` is changed, the App Center recognizes this and will exchange the image by removing the container and setting up a new one. This is of course not necessary for an old, but constantly updated image. So one can put the old image name in the `DockerAllowedImages` list and the App Center will accept it. New apps will be installed with `DockerImage`, though.

`DockerVolumes`
Apart from `/var/lib/univention-appcenter/apps/$APPID/{data,conf}/`, one may choose further directories that shall be mounted into the Docker Container. This is a good idea if it makes storing / restoring data easy and also provides the benefit of increased I/O performance. The syntax is `DockerVolumes=/path/on/host/:/path/in/container/,...`.

`DockerServerRole`
The Univention App Center adds a computer object for the container in the LDAP database. This computer object can either be a `memberserver` (default) or a `domaincontroller_slave`. The latter is useful if the container needs more rights (DC systems may have more rights in the ACL definitions of LDAP). Other than that, both roles are free to choose their software selection (depending on the App package definitions in its `debian/control`).

`DockerScript*`
The various `DockerScript*` variables are used to specify the path to the script that shall be executed. So an App package may ship its own `DockerScriptStoreData` and the ini file can point to it. More importantly though, the App can ship these scripts unpackaged along with the ini file and the App Center will install these scripts at that destination and then call it. See Section 7.3.1 for a brief overview over the scripts. The name in the ini file for `store_data` is `DockerScriptStoreData` and so on.

`PortsExclusive`
: Not really limited to Docker Apps, but this variable will be essential to make your Docker App work: It specifies a comma separated list of ports that the Docker Container shall expose publicly. If your App opens port, say, 3000, you need to specify that in the ini file, otherwise the port will not be accessible from any host but the Docker Container itself. The Docker Host will configure the container according to this variable on startup. By default, no port is exposed. One exception is a port to a `WebInterface` (also to be specified in the ini file) which is handled automatically by the Univention App Center.

> **Note**
>
> ***univention-firewall*** on the Docker Host will be reconfigured automatically to allow connections to these ports.

`PortsRedirection`
: Just like `PortsExclusive` but with an the option to map the local port to one other port on the Docker Host. Useful when exposing a service that is used commonly and therefore will generate port conflicts. E.g., if your App shall be accessible via SSH, you cannot exclusively claim port 22. But you can specify `PortsRedirection=10022:22`.

`WebInterface`
: Not exclusive to Docker Apps, but a bit different as to what happens in the background: If your App provides a web interface (e.g., `/radicale/`), you can specify it here. In this case, the Univention App Center will automatically make the Docker Container's web interface publicly accessible by using ***mod_proxy*** of ***apache*** (using the port range 40000 up to 41000, controlled by the Univention App Center itself).

> **Note**
>
> The proxy entry in the Apache configuration will be something like:
>
> ```
> ProxyPass /radicale http://127.0.0.1:40000/radicale
> ```
>
> where the port 40000 is mapped to Docker Container's port `WebInterfacePortHTTP` (which defaults to 80). So your App needs to make `/radicale/` accessible, too. Note that there will be a separate entry for HTTPS, too.

> **Note**
>
> HTTP or HTTPS access can be disabled by specifying `WebInterfacePortHTTP=0` (`WebInterfacePortHTTPS` respectively).

Please refer to the Developer Reference[4] for a template and the description of every attribute in the ini file.

7.4. Debugging

Feedback

So you developed the App, put it into the App Center and now that you want to test it, something fails? If something goes wrong really badly (like the installation of packages fails), the App Center reverts the installation (i.e. it removes the container). This means you cannot look into it to see what exactly happened. For testing purposes you may use the undocumented options `--do-not-revert` (still using Radicale):

```
univention-app install radicale --do-not-revert
```

This will leave you with a running, yet somewhat "unconfigured" container. You may now login into this container by doing

[4] https://docs.software-univention.de/developer-reference.html#app:iniFile

```
docker ps -a
```

which will list your container. Find the container and call:

```
docker exec -it "$CONTAINER "/bin/bash
```

Normally you should be able to do this without finding the container by hand:

```
CONTAINER="$(ucr get appcenter/apps/radicale/container)"
docker exec -it "$CONTAINER" /bin/bash
```

From here you can debug the system as normal.

Maybe it is sufficient to look into the log files to know what went wrong. Important log files are:

- `/var/log/univention/appcenter.log`
- `/var/log/univention/management-console-module-appcenter.log` (if you installed the App via the UMC module - which is actually a good idea as users will install it that way, too).
- `/var/log/docker.log`