KIWI NG Documentation

Release 9.23.20

Marcus Schäfer

Mar 12, 2021

CONTENTS

Note: This documentation covers KIWI Next Generation (KIWI NG) 9.23.20- the command line utility to build Linux system appliances. KIWI NG is stable and all new features, bugfixes, and improvements will be developed here. Versions older or equal to v7.x.x are out of maintenance and do not get any updates or bugfixes. If you still need this version, refer to the documentation for KIWI Legacy

CHAPTER ONE

OVERVIEW

Note: Abstract

This document provides a conceptual overview about the steps of creating an image with KIWI NG. It also explains the terminology regarding the concept and process when building system images with KIWI NG 9.23.20.

1.1 Basic Workflow

Note: Abstract

Installation of a Linux system generally occurs by booting the target system from an installation source such as an installation CD/DVD, a live CD/DVD, or a network boot environment (PXE). The installation process is often driven by an installer that interacts with the user to collect information about the installation. This information generally includes the *software to be installed*, the *timezone*, system *user* data, and other information. Once all the information is collected, the installer installs the software onto the target system using packages from the software sources (repositories) available. After the installation is complete the system usually reboots and enters a configuration procedure upon start-up. The configuration may be fully automatic or it may include user interaction.

This description applies for version 9.23.20.

A system image (usually called “image”), is a *complete installation* of a Linux system within a file. The image represents an operational system and, optionally, contains the “final” configuration.

The behavior of the image upon deployment varies depending on the image type and the image configuration since KIWI NG allows you to completely customize the initial start-up behavior of the image. Among others, this includes images that:

- can be deployed inside an existing virtual environment without requiring configuration at start-up.
- automatically configure themselves in a known target environment.

- prompt the user for an interactive system configuration.

The image creation process with KIWI NG is automated and does not require any user interaction. The information required for the image creation process is provided by the primary configuration file named `config.xml`. This file is validated against the schema documented in: *Image Description*.

In addition, the image can optionally be customized using the `config.sh` and `images.sh` scripts and by using an *overlay tree (directory)* called `root`. See *Components of an Image Description* section for further details.

Note: Previous Knowledge

This documentation assumes that you are familiar with the general concepts of Linux, including the boot process, and distribution concepts such as package management.

1.1.1 Components of an Image Description

A KIWI NG image description can composed by several parts. The main part is the KIWI NG description file itself (named `config.xml` or an arbitrary name plus the `*.kiwi` extension). The configuration XML is the only required component, others are optional.

These are the optional components of an image description:

1. `config.sh` shell script

   Is the configuration shell script that runs and the end of the *prepare step* if present. It can be used to fine tune the unpacked image.

   Note that the script is directly invoked by the operating system if its executable bit is set. Otherwise it is called by `bash` instead.

2. `images.sh` shell script

   Is the configuration shell script that runs at the beginning of the create step. So it is expected to be used to handle image type specific tasks. It is called in a similar fashion as `config.sh`

3. Overlay tree directory

   The *overlay tree* is a folder (called `root`) or a tarball file (called `root.tar.gz`) that contains files and directories that will be copied to the target image build tree during the *prepare step*. It is executed after all the packages included in the `config.xml` file have been installed. Any already present file is overwritten.

4. CD root user data

   For live ISO images and install ISO images an optional cdroot archive is supported. This is a tar archive matching the name `config-cdroot.tar[.compression_postfix]`. If present it will be unpacked as user data on the ISO image. This is mostly useful to add e.g license files or user documentation on the CD/DVD which can be read directly without booting from the media.

5. Archives included in the `config.xml` file.

   The archives that are included in the `<packages>` using the `<archive>` subsection:

   ```
   <packages type="image">
       <archive name="custom-archive.tgz"/>
   </packages>
   ```

1.2 Conceptual Overview

A system image (usually called "image"), is a *complete installation* of a Linux system within a file. The image represents an operation system and, optionally, contains the "final" configuration.

KIWI NG creates images in a two step process:

1. The first step, the *prepare operation*, generates a so-called *unpacked image tree* (directory) using the information provided in the image description.
2. The second step, the *create operation*, creates the *packed image* or *image* in the specified format based on the unpacked image and the information provided in the configuration file.

The image creation process with KIWI NG is automated and does not require any user interaction. The information required for the image creation process is provided by the image description.

1.3 Terminology

Appliance An appliance is a ready to use image of an operating system including a pre-configured application for a specific use case. The appliance is provided as an image file and needs to be deployed to, or activated in the target system or service.

Image The result of a KIWI NG build process.

Image Description Specification to define an appliance. The image description is a collection of human readable files in a directory. At least one XML file `config.xml` or `.kiwi` is required. In addition there may be as well other files like scripts or configuration data. These can be used to customize certain parts either of the KIWI NG build process or of the initial start-up behavior of the image.

Overlay Files A directory structure with files and subdirectories stored as part of the Image Description. This directory structure is packaged as a file `root.tar.gz` or stored inside a directory named `root`. Additional overlay directories for selected profiles are supported too and are taken into account if the directory name matches the name of the profile. The content of each of the directory structures is copied on top of the the existing file system (overlayed) of the appliance root. This also includes permissions and attributes as a supplement.

KIWI NG An OS appliance builder.

Virtualization Technology Software simulated computer hardware. A virtual machine acts like a real computer, but is separated from the physical hardware. Within this documentation the QEMU virtualization system is used. Another popular alternative is Virtualbox.

1.4 System Requirements

To use and run KIWI NG, you need:

- A recent Linux distribution, see *Build Host Constraints* for details.
- Enough free disk space to build and store the image. We recommend a minimum of 15GB.
- Python version 3.5 or higher
- Git (package `git`) to clone a repository.
- Any virtualization technology to start the image. We recommend QEMU.

CHAPTER

TWO

INSTALLATION

Note: This document describes how to install KIWI NG. Apart from the preferred method to install KIWI NG via rpm, it is also available on pypi and can be installed via pip.

2.1 Installation from OBS

The most up to date packages of KIWI NG can be found on the Open Build Service in the Virtualization:Appliances:Builder project.

To install KIWI NG, follow these steps:

1. Open the URL https://download.opensuse.org/repositories/Virtualization:/Appliances:/Builder in your browser.
2. Right-click on the link of your preferred operating system and copy the URL. In Firefox it is the menu *Copy link address*.
3. Insert the copied URL from the last step into your shell. The `DIST` placeholder contains the respective distribution. Use **zypper addrepo** to add it to the list of your repositories:

   ```
   $ sudo zypper addrepo http://download.opensuse.org/repositories/
   ↪Virtualization:/Appliances:/Builder/<DIST> appliance-builder
   ```

   If your distribution is not using **zypper**, please use your package manager's appropriate command instead. For **dnf** that is:

   ```
   $ sudo dnf config-manager --add-repo https://download.opensuse.
   ↪org/repositories/Virtualization:/Appliances:/Builder/<DIST>/
   ↪Virtualization:Appliances:Builder.repo
   ```

4. Add the repositories' signing-key to your package manager's database. For rpm run:

   ```
   $ sudo rpm --import https://build.opensuse.org/projects/
   ↪Virtualization:Appliances:Builder/public_key
   ```

   And verify that you got the correct key:

```
$ rpm -qi gpg-pubkey-74cbe823-* | gpg2
gpg: WARNING: no command supplied.  Trying to guess what you␣
↪mean ...
pub   dsa1024 2009-05-04 [SC] [expires: 2020-10-09]
      F7E82012C74FD0B85F5334DC994B195474CBE823
uid           Virtualization:Appliances OBS Project
↪<Virtualization:Appliances@build.opensuse.org>
```

Note: **rpm** requires network utilities in order to download and import remote keys. Make sure **curl** is installed before trying to import remote keys using **rpm**.

Alternatively, the package manager, if not executed in non-interactive mode, will ask you to trust or not the public key of the new repository when refreshing repositories or installing new packages. If trusted the package manager will automatically import it.

5. Install KIWI NG:

```
$ sudo zypper in python3-kiwi
```

2.2 Installation from your distribution's repositories

Note: There are many packages that contain the name KIWI NG in their name, some of these are even python packages. Please double check the packages' description whether it is actually the KIWI NG Appliance builder before installing it.

Some Linux distributions ship KIWI NG in their official repositories. These include openSUSE Tumbleweed, openSUSE Leap, and Fedora since version 28. Note, these packages tend to not be as up to date as the packages from OBS, so some features described here might not exist yet.

To install KIWI NG on openSUSE, run the following command:

```
$ sudo zypper install python3-kiwi
```

On Fedora, use the following command instead:

```
$ sudo dnf install kiwi-cli
```

2.3 Installation from PyPI

KIWI NG can be obtained from the Python Package Index (PyPi) via Python's package manager pip:

```
$ pip install kiwi
```

2.4 Example Appliance Descriptions

There is a GitHub project hosting example appliance descriptions to be used with the next generation KIWI NG. Users who need an example to start with should clone the project as follows:

```
$ git clone https://github.com/OSInside/kiwi-descriptions
```

CHAPTER

THREE

QUICK START

Note: Abstract

This document describes how to start working with KIWI NG, an OS appliance builder. This description applies for version 9.23.20.

3.1 Before you start

1. Install KIWI NG first, either via your distributions' package manager (see *Installation*) or via:

   ```
   $ pip install kiwi
   ```

2. Clone the repository containing example appliances (see *Example Appliance Descriptions*):

   ```
   $ git clone https://github.com/OSInside/kiwi-descriptions
   ```

3.2 Choose a First Image

Take a look which images are available in the example appliances repository and select one that matches your desired image as close as possible. Or just use the one given in the examples below.

3.3 Build your First Image

Your first image will be a simple system disk image which can run in any full virtualization system like QEMU. Invoke the following KIWI NG command in order to build it:

```
$ sudo kiwi-ng --profile=Disk --type oem system build \
    --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
    --target-dir /tmp/myimage
```

The resulting image will be placed into the folder `/tmp/myimage` with the suffix `.raw`.

If you don't wish to create a openSUSE Leap 15.1 image, substitute the folder following the `--description` option with another folder that contains the image description which you selected.

3.4 Run your Image

Running an image actually means booting the operating system. In order to do that attach the disk image to a virtual system. In this example we use QEMU and boot it as follows:

```
$ qemu -boot c \
    -drive file=Leap-15.1_appliance.x86_64-1.15.1.raw,format=raw,
↪if=virtio \
    -m 4096
```

3.5 Tweak and Customize your Image

Now that you have successfully built and started your first image, you can start tweaking it to match your needs.

CHAPTER

FOUR

WORKING FROM THE COMMAND LINE

Note: This document provides a list of the existing KIWI Next Generation (KIWI NG) commands for version 9.23.20.

4.1 kiwi-ng

4.1.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng -h | --help
kiwi-ng [--profile=<name>...]
        [--type=<build_type>]
        [--logfile=<filename>]
        [--debug]
        [--color-output]
        [--config=<configfile>]
    image <command> [<args>...]
kiwi-ng [--debug]
        [--color-output]
        [--config=<configfile>]
    result <command> [<args>...]
kiwi-ng [--profile=<name>...]
        [--shared-cache-dir=<directory>]
        [--type=<build_type>]
        [--logfile=<filename>]
        [--debug]
        [--color-output]
        [--config=<configfile>]
    system <command> [<args>...]
kiwi-ng compat <legacy_args>...
kiwi-ng -v | --version
kiwi-ng help
```

4.1.2 DESCRIPTION

KIWI NG is an imaging solution that is based on an image XML description. Such a description is represented by a directory which includes at least one `config.xml` or `.kiwi` file and may as well include other files like scripts or configuration data.

A collection of example image descriptions can be found on the github repository here: https://github.com/OSInside/kiwi-descriptions. Most of the descriptions provide a so called appliance image. Appliance means that it's a small, text only based image including a predefined remote source setup to allow installation of missing software components at a later point in time.

KIWI NG operates in two steps. The system build command combines both steps into one to make it easier to start with KIWI NG. The first step is the preparation step and if that step was successful, a creation step follows which is able to create different image output types.

In the preparation step, you prepare a directory including the contents of your new filesystem based on one or more software package source(s) The creation step is based on the result of the preparation step and uses the contents of the new image root tree to create the output image.

KIWI NG supports the creation of the following image types:

- ISO Live Systems
- Virtual Disk for e.g cloud frameworks
- OEM Expandable Disk for system deployment from ISO or the network
- File system images for deployment in a pxe boot environment

Depending on the image type a variety of different disk formats and architectures are supported.

4.1.3 GLOBAL OPTIONS

--color-output Use Escape Sequences to print different types of information in colored output. The underlaying terminal has to understand those escape characters. Error messages appear red, warning messages yellow and debugging information will be printed light grey.

--config=<configfile> Use specified runtime configuration file. If not specified the runtime configuration is looked up at `~/.config/kiwi/config.yml` or `/etc/kiwi.yml`

--debug Print debug information on the commandline.

--logfile=<filename> Specify log file. the logfile contains detailed information about the process.

--profile=<name> Select profile to use. The specified profile must be part of the XML description. The option can be specified multiple times to allow using a combination of profiles

--shared-cache-dir=<directory> Specify an alternative shared cache directory. The directory is shared via bind mount between the build host and image root system and contains information about package repositories and their cache and meta data. The default location is set to /var/cache/kiwi

--type=<build_type> Select image build type. The specified build type must be configured as part of the XML description.

--version Show program version

4.1.4 EXAMPLE

```
$ git clone https://github.com/OSInside/kiwi-descriptions

$ kiwi --type oem system build \
    --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
    --target-dir /tmp/myimage
```

4.1.5 COMPATIBILITY

This version of KIWI NG uses a different caller syntax compared to former versions. However there is a compatibility mode which allows to use a legacy KIWI NG commandline as follows:

```
$ kiwi compat \
    --build kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
    --type oem -d /tmp/myimage
```

4.2 kiwi-ng result list

4.2.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng result list -h | --help
kiwi-ng result list --target-dir=<directory>
kiwi-ng result list help
```

4.2.2 DESCRIPTION

List build results from a previous build or create command. Please note if you build an image several times with the same target directory the build result information will be overwritten each time you build the image. Therefore the build result list is valid for the last build

4.2.3 OPTIONS

--target-dir=<directory> directory containing the kiwi build results

4.3 kiwi-ng result bundle

4.3.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng result bundle -h | --help
kiwi-ng result bundle --target-dir=<directory> --id=<bundle_id> --
↪bundle-dir=<directory>
    [--zsync_source=<download_location>]
kiwi-ng result bundle help
```

4.3.2 DESCRIPTION

Create result bundle from the image build results in the specified target directory. Each result image will contain the specified bundle identifier as part of its filename. Uncompressed image files will also become xz compressed and a sha sum will be created from every result image.

4.3.3 OPTIONS

--bundle-dir=<directory> directory containing the bundle results, compressed versions of image results and their sha sums

--id=<bundle_id> bundle id, could be a free form text and is appended to the image version information if present as part of the result image filename

--target-dir=<directory> directory containing the kiwi build results

--zsync_source=<download_location> Specify the download location from which the bundle file(s) can be fetched from. The information is effective if `zsync` is used to sync the bundle.

- The zsync control file is only created for those bundle files which are marked for compression because in a KIWI NG build only those are meaningful for a partial binary file download.
- It is expected that all files from a bundle are placed to the same download location

4.4 kiwi-ng system prepare

4.4.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng system prepare -h | --help
kiwi-ng system prepare --description=<directory> --root=<directory>
    [--allow-existing-root]
    [--clear-cache]
    [--ignore-repos]
    [--ignore-repos-used-for-build]
    [--set-repo=<source,type,alias,priority,imageinclude,package_
↪gpgcheck>]
    [--add-repo=<source,type,alias,priority,imageinclude,package_
↪gpgcheck>...]
    [--add-package=<name>...]
    [--add-bootstrap-package=<name>...]
    [--delete-package=<name>...]
    [--set-container-derived-from=<uri>]
    [--set-container-tag=<name>]
    [--add-container-label=<label>...]
    [--signing-key=<key-file>...]
kiwi-ng system prepare help
```

4.4.2 DESCRIPTION

Create a new image root directory. The prepare step builds a new image root directory from the specified XML description. The specified root directory is the root directory of the new image root system. As the root user you can enter this system via chroot as follows:

```
$ chroot <directory> bash
```

4.4.3 OPTIONS

--add-bootstrap-package=<name> specify package to install as part of the early kiwi bootstrap phase. The option can be specified multiple times

--add-container-label=<name=value> add a container label in the container configuration metadata. It overwrites the label with the provided key-value pair in case it was already defined in the XML description

--add-package=<name> specify package to add(install). The option can be specified multiple times

--add-repo=<source,type,alias,priority,imageinclude,package_gpgcheck> Add a new repository to the existing repository setup in the XML description. This option can be specified multiple times. For details about the provided option values see the **–set-repo** information below

--allow-existing-root allow to re-use an existing image root directory

--clear-cache delete repository cache for each of the used repositories before installing any package. This is useful if an image build should take and validate the signature of the package from the original repository source for any build. Some package managers unconditionally trust the contents of the cache, which is ok for cache data dedicated to one build but in case of kiwi the cache is shared between multiple image builds on that host for performance reasons.

--delete-package=<name> specify package to delete. The option can be specified multiple times

--description=<directory> Path to the kiwi XML description. Inside of that directory there must be at least a config.xml of *.kiwi XML description.

--ignore-repos Ignore all repository configurations from the XML description. Using that option is usally done with a sequence of –add-repo options otherwise there are no repositories available for the image build which would lead to an error.

--ignore-repos-used-for-build Works the same way as –ignore-repos except that repository configurations which has the imageonly attribute set to true will not be ignored.

--root=<directory> Path to create the new root system.

--set-repo=<source,type,alias,priority,imageinclude,package_gpgcheck> Overwrite the first repository entry in the XML description with the provided information:

- **source**

source url, pointing to a package repository which must be in a format supported by the selected package manager. See the URI_TYPES section for details about the supported source locators.

- **type**

  repository type, could be one of `rpm-md`, `rpm-dir` or `yast2`.

- **alias**

  An alias name for the repository. If not specified kiwi calculates an alias name as result from a sha sum. The sha sum is used to uniquely identify the repository, but not very expressive. We recommend to set an expressive and uniq alias name.

- **priority**

  A number indicating the repository priority. How the value is evaluated depends on the selected package manager. Please refer to the package manager documentation for details about the supported priority ranges and their meaning.

- **imageinclude**

  Set to either **true** or **false** to specify if this repository should be part of the system image repository setup or not.

- **package_gpgcheck**

  Set to either **true** or **false** to specify if this repository should validate the package signatures.

--set-container-derived-from=<uri> overwrite the source location of the base container for the selected image type. The setting is only effective if the configured image type is setup with an initial derived_from reference

--set-container-tag=<name> overwrite the container tag in the container configuration. The setting is only effective if the container configuraiton provides an initial tag value

--signing-key=<key-file> set the key file to be trusted and imported into the package manager database before performing any opertaion. This is useful if an image build should take and validate repository and package signatures during build time. This option can be specified multiple times.

4.5 kiwi-ng system update

4.5.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng system update -h | --help
kiwi-ng system update --root=<directory>
    [--add-package=<name>...]
    [--delete-package=<name>...]
kiwi-ng system update help
```

4.5.2 DESCRIPTION

Update a previously prepare image root tree. The update command refreshes the contents of the root directory with potentially new versions of the packages according to the repository setup of the image XML description. In addition the update command also allows to add or remove packages from the image root tree

4.5.3 OPTIONS

--add-package=<name> specify package to add(install). The option can be specified multiple times

--delete-package=<name> specify package to delete. The option can be specified multiple times

--root=<directory> Path to the root directory of the image.

4.6 kiwi-ng system build

4.6.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng system build -h | --help
kiwi-ng system build --description=<directory> --target-dir=
↪<directory>
    [--allow-existing-root]
    [--clear-cache]
    [--ignore-repos]
    [--ignore-repos-used-for-build]
    [--set-repo=<source,type,alias,priority,imageinclude,package_
↪gpgcheck>]
```

(continues on next page)

(continued from previous page)

```
    [--add-repo=<source,type,alias,priority,imageinclude,package_
↪gpgcheck>...]
    [--add-package=<name>...]
    [--add-bootstrap-package=<name>...]
    [--delete-package=<name>...]
    [--set-container-derived-from=<uri>]
    [--set-container-tag=<name>]
    [--add-container-label=<label>...]
    [--signing-key=<key-file>...]
kiwi-ng system build help
```

4.6.2 DESCRIPTION

build an image in one step. The build command combines kiwi's prepare and create steps in order to build an image with just one command call. The build command creates the root directory of the image below `<target-dir>/build/image-root` and if not specified differently writes a log file `<target-dir>/build/image-root.log`. The result image files are created in the specified target-dir.

4.6.3 OPTIONS

--add-bootstrap-package=<name> specify package to install as part of the early kiwi bootstrap phase. The option can be specified multiple times

--add-container-label=<name=value> add a container label in the container configuration metadata. It overwrites the label with the provided key-value pair in case it was already defined in the XML description

--add-package=<name> specify package to add(install). The option can be specified multiple times

--add-repo=<source,type,alias,priority,imageinclude,package_gpgcheck> Add a new repository to the existing repository setup in the XML description. This option can be specified multiple times. For details about the provided option values see the **–set-repo** information below

--allow-existing-root Allow to use an existing root directory from an earlier build attempt. Use with caution this could cause an inconsistent root tree if the existing contents does not fit to the former image type setup

--clear-cache delete repository cache for each of the used repositories before installing any package. This is useful if an image build should take and validate the signature of the package from the original repository source for any build. Some package managers unconditionally trust the contents of the cache, which is ok for cache data dedicated to one build but in case of kiwi the cache is shared between multiple image builds on that host for performance reasons.

--delete-package=<name> specify package to delete. The option can be specified multiple times

--description=<directory> Path to the XML description. This is a directory containing at least one _config.xml_ or _*.kiwi_ XML file.

--ignore-repos Ignore all repository configurations from the XML description. Using that option is usally done with a sequence of –add-repo options otherwise there are no repositories available for the image build which would lead to an error.

--ignore-repos-used-for-build Works the same way as –ignore-repos except that repository configurations which has the imageonly attribute set to true will not be ignored.

--set-repo=<source,type,alias,priority,imageinclude,package_gpgcheck>
Overwrite the first repository entry in the XML description with the provided information:

- **source**

  source url, pointing to a package repository which must be in a format supported by the selected package manager. See the URI_TYPES section for details about the supported source locators.

- **type**

  repository type, could be one of `rpm-md`, `rpm-dir` or `yast2`.

- **alias**

  An alias name for the repository. If not specified kiwi calculates an alias name as result from a sha sum. The sha sum is used to uniquely identify the repository, but not very expressive. We recommend to set an expressive and uniq alias name.

- **priority**

  A number indicating the repository priority. How the value is evaluated depends on the selected package manager. Please refer to the package manager documentation for details about the supported priority ranges and their meaning.

- **imageinclude**

  Set to either **true** or **false** to specify if this repository should be part of the system image repository setup or not.

- **package_gpgcheck**

Set to either **true** or **false** to specify if this repository should validate the package signatures.

--set-container-derived-from=<uri> overwrite the source location of the base container for the selected image type. The setting is only effective if the configured image type is setup with an initial derived_from reference

--set-container-tag=<name> overwrite the container tag in the container configuration. The setting is only effective if the container configuraiton provides an initial tag value

--signing-key=<key-file> set the key file to be trusted and imported into the package manager database before performing any opertaion. This is useful if an image build should take and validate repository and package signatures during build time. This option can be specified multiple times

--target-dir=<directory> Path to store the build results.

4.6.4 URI_TYPES

- **http:// | https:// | ftp://**

  remote repository delivered via http or ftp protocol.

- **obs://**

  Open Buildservice repository. The source data is translated into an http url pointing to http://download.opensuse.org.

- **ibs://**

  Internal Open Buildservice repository. The source data is translated into an http url pointing to download.suse.de.

- **iso://**

  Local iso file. kiwi loop mounts the file and uses the mount point as temporary directory source type

- **dir://**

  Local directory

4.7 kiwi-ng system create

4.7.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng system create -h | --help
kiwi-ng system create --root=<directory> --target-dir=<directory>
    [--signing-key=<key-file>...]
kiwi-ng system create help
```

4.7.2 DESCRIPTION

Create an image from a previously prepared image root directory. The kiwi create call is usually issued after a kiwi prepare command and builds the requested image type in the specified target directory

4.7.3 OPTIONS

--root=<directory> Path to the image root directory. This directory is usually created by the kiwi prepare command. If a directory is used which was not created by kiwi's prepare command, it's important to know that kiwi stores image build metadata below the image/ directory which needs to be present in order to let the create command operate correctly.

--target-dir=<directory> Path to store the build results.

--signing-key=<key-file> set the key file to be trusted and imported into the package manager database before performing any opertaion. This is useful if an image build should take and validate repository and package signatures during build time. In create step this option only affects the boot image. This option can be specified multiple times

4.8 kiwi-ng image resize

4.8.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng image resize -h | --help
kiwi-ng image resize --target-dir=<directory> --size=<size>
```

(continues on next page)

(continued from previous page)

```
    [--root=<directory>]
kiwi-ng image resize help
```

4.8.2 DESCRIPTION

For disk based images, allow to resize the image to a new disk geometry. The additional space is free and not in use by the image. In order to make use of the additional free space a repartition process is required like it is provided by kiwi's oem boot code. Therefore the resize operation is useful for oem image builds most of the time.

4.8.3 OPTIONS

--root=<directory> The path to the root directory, if not specified kiwi searches the root directory in build/image-root below the specified target directory

--size=<size> New size of the image. The value is either a size in bytes or can be specified with m=MB or g=GB. Example: 20g

--target-dir=<directory> Directory containing the kiwi build results

4.9 kiwi-ng image info

4.9.1 SYNOPSIS

```
kiwi-ng [global options] service <command> [<args>]

kiwi-ng image info -h | --help
kiwi-ng image info --description=<directory>
    [--resolve-package-list]
    [--ignore-repos]
    [--add-repo=<source,type,alias,priority>...]
    [--print-xml|--print-yaml]
kiwi-ng image info help
```

4.9.2 DESCRIPTION

Provides information about the specified image description. If no specific info option is provided the command just lists basic information about the image which could also be directly obtained by reading the image XML description file. Specifying an extension option like `resolve-package-list` will cause a dependency resolver to run over the list of packages and thus provides more detailed information about the image description.

4.9.3 OPTIONS

--add-repo=<source,type,alias,priority> Add repository with given source, type, alias and priority.

--description=<directory> The description must be a directory containing a kiwi XML description and optional metadata files.

--ignore-repos Ignore all repository configurations from the XML description. Using that option is usally done with a sequence of –add-repo options otherwise there are no repositories available for the processing the requested image information which could lead to an error.

--resolve-package-list Solve package dependencies and return a list of all packages including their attributes e.g size, shasum, and more.

--print-xml Print image description in XML format. The given image description is read in, transformed internally to XML and send to the XSLT stylesheet processor. From there the result gets validated using the RelaxNG schema and the schematron rules. This result data will then be displayed. The typical use case for this command is to turn an old image description to the latest schema.

--print-yaml Behaves the same like `--print-xml` except that after validation the result data will be transformed into the YAML format and displayed. Due to this processing the command can be used for different operations:

- Conversion of a given image description from or into different formats. It's required to install the `anymarkup` python module for this to work. The module is not a hard requirement and loaded on demand. If not available and a request to convert into a format different from XML is made an exception will be thrown.
- Update of an old image description to the latest schema

CHAPTER FIVE

TROUBLESHOOTING

Note: Abstract

This document describes situations which leads to issues during build or boot time of the image build with KIWI NG. The suggested solutions are considered best practice but are just one out of other possible solution candidates.

5.1 Incompatible Filesystem Settings on Host vs. Image

Note: Abstract

This page provides further information how to solve image boot problems if the filesystem tool chain on the image build host is incompatible with the image target distribution

When KIWI NG builds an image which requests the creation of a filesystem, the required filesystem creation tool, for example `mkfs.xfs`, is called from the host on which KIWI NG gets called. It is expected that the generated filesystem is compatible with the image target distribution. This expectation is not always correct and depends on the compatibility of the filesystem default settings between build host and image target. We know about the following settings that causes an incompatible filesystem which will not be able to be used on boot:

Ext[2,3,4] Check `/etc/mke2fs.conf` on the build host and make sure the configured `inode_size` is the same as the setting used for the target image. To solve an issue of this type use the following filesystem creation option in your KIWI NG image configuration:

```
<type fscreateoptions="-I inode-size"/>
```

XFS Check the XFS metadata setup on the build host and make sure the settings are compatible with the target image. XFS has the default settings compiled in, thus it might be needed to build the image first and use the `xfs_info` tool in a `disk.sh` script to fetch the settings at build time of the image. We know from community reports that the setting `sparse=1` will cause issues on older versions of grub's xfs module, which does not know how to handle this setting properly. To solve an issue of this type use the following filesystem creation option in your KIWI NG image configuration:

```
<type fscreateoptions="-i sparse=0"/>
```

Note: There can be more inconsistencies in the area of filesystems which we haven't listed here. In general it's advisable to build the image in a compatible environment. At best the build host distribution is of the same major Linux version than the image target. For this purpose KIWI NG provides the so called `boxed-plugin`. Further details can be found in *Building in a Self-Contained Environment*

5.2 Build Host Constraints

For building images a host system is required that runs the build process. Tools to create the image are used from that host and this creates an indirect dependency to the target image. For example; Building an Ubuntu image requires the apt and dpkg tools and metadata to be available and functional on the host to build an Ubuntu image. There are many more of those host vs. image dependencies and not all of them can be resolved in a clear and clean way.

The most compatible environment is provided if the build host is of the same distribution than the target image. In other cases our recommendation is that the build host is of the same distribution than the target and near to the major version (+-1) compared to the target. Such an environment can be found in:

- The Open Build Service OBS.
- The KIWI NG boxed plugin: *Building in a Self-Contained Environment*

In general, our goal is to support any major distribution with KIWI NG. However for building images we rely on core tools which are not under our control. Also several design aspects of distributions like **secure boot** and working with **upstream projects** are different and not influenced by us. There are many side effects that can be annoying especially if the build host is not of the same distribution vendor than the image target.

5.3 Architectures

As described in the section above one requirement between the build host and the image when it comes to architecture support is, that the image architecture should match the build host architecture. Cross arch building would require any core tool that is used to build an image to be cross arch capable. To patch e.g an x86_64 system such that it can build an aarch64 image would require some work on binutils and hacks as well as performance tweaks which is all not worth the effort and still can lead to broken results. Thus we recommend to provide native systems for the target architecture and build there. One possible alternative is to use the kiwi boxed plugin as mentioned above together with a box created for the desired architecture. However keep in mind the performance problematic when running a VM of a different architecture.

The majority of the image builds are based on the x86 architecture. As mentioned KIWI NG also supports other architectures, shown in the table below:

Architecture	Supported
x86_64	yes
ix86	yes **note:distro**
s390/s390x	yes **note:distro**
arm/aarch64	yes **note:distro**
ppc64	yes **note:distro**

distro

The support status for an architecture depends on the distribution. If the distribution does not build its packages for the desired architecture, KIWI NG will not be able to build an image for it

CHAPTER

SIX

BUILDING IN A SELF-CONTAINED ENVIRONMENT

Note: Abstract

Users building images with KIWI NG face problems if they want to build an image matching one of the following criteria:

- build should happen as non root user.
- build should happen on a host system distribution for which no KIWI NG packages exists.
- build happens on an incompatible host system distribution compared to the target image distribution. For example building an apt/dpkg based system on an rpm based system.
- run more than one build process at the same time on the same host.
- run a build process for a different target architecture compared to the host architecture (Cross Arch Image Build)

This document describes how to perform the build process in a self contained environment using fast booting virtual machines to address the issues listed above.

The changes on the machine to become a build host will be reduced to the requirements of the KIWI NG `boxed plugin`

6.1 Requirements

Add the KIWI NG repo from the Open Build Service. For details see *Installation from OBS*. The following KIWI NG plugin needs to be installed on the build system:

```
$ sudo zypper in python3-kiwi_boxed_plugin
```

6.2 Building with the boxbuild command

The installation of the KIWI NG boxed plugin has registered a new kiwi command named `boxbuild`. The command implementation uses KVM as virtualization technology and runs the KIWI NG `build` command inside of a KVM controlled virtual machine. For running the build process in a virtual machine it's required to provide VM images that are suitable to perform this job. We call the VM images `boxes` and they contain kiwi itself as well as all other components needed to build appliances. Those boxes are hosted in the Open Build Service and are publicly available at the `Subprojects` tab in the: Virtualization:Appliances:SelfContained project.

As a user you don't need to work with the boxes because this is all done by the plugin and provided as a service by the KIWI NG team. The `boxbuild` command knows where to fetch the box and also cares for an update of the box when it has changed.

Building an image with the `boxbuild` command is similar to building with the `build` command. The plugin validates the given command call with the capabilities of the `build` command. Thus one part of the `boxbuild` command is exactly the same as with the `build` command. The separation between `boxbuild` and `build` options is done using the `--` separator like shown in the following example:

```
$ kiwi-ng --type iso system boxbuild --box leap -- \
      --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
      --target-dir /tmp/myimage
```

Note: The provided `--description` and `--target-dir` options are setup as shared folders between the host and the guest. No other data will be shared with the host.

CHAPTER

SEVEN

CONCEPT AND WORKFLOW

Note: Abstract

The following sections describe the concept and general workflow of building appliances with KIWI NG 9.23.20.

7.1 Host Requirements To Build Images

When building OS images, several tools and sub-systems are used and required on the host KIWI NG is called at. For example, to build a virtual disk image, several tools needs to be available on the host that builds the image. This includes tools for partition table setup or tools to create filesystems.

The number of required components depends on the selected image type and the *features* used with the image. We cannot expect the users of KIWI NG to know about each and every component that is needed to build the image. Therefore a concept to help with the host requirements exists and is named `kiwi-systemdeps`

The `kiwi-systemdeps` concept consists out of a collection of sub-packages provided with the `python-kiwi` main package. Each individual package requires a number of tools and subsystem packages which belongs to the package category. There are the following systemdeps packages:

`kiwi-systemdeps-core:`

- Supports building the simple root archive `tbz` image type.
- Installs the package managers which are supported by the target distribution as well as the `tar` archiving tool.

`kiwi-systemdeps-containers:`

- Supports building `docker` and `appx` image types.
- Installs the distribution specific tool chain to build OCI compliant and WSL container images.

`kiwi-systemdeps-iso-media:`

- Supports building `iso` image types and `oem` install media.
- Installs all tools required to build ISO filesystems.
- Depends on the `-core`, `-filesystems` and `-bootloaders` kiwi-systemdeps packages.

`kiwi-systemdeps-bootloaders`:

- Supports building bootable `oem` and `iso` image types.
- Installs all bootloader tools depending on the host architecture to allow setup and install of the bootloader. The pulled in components are required for any image that is able to boot through some BIOS or firmware.
- Depends on the `-core` kiwi-systemdeps packages.

Note: The `iso` type is an exception which might not require the `-bootloaders` systemdeps. In case of the `firmware` attribute to be set to `bios`, KIWI NG builds bootable ISO images still based on isolinux which is provided with the `-iso-media` systemdeps. However, by default, any KIWI NG created ISO image is BIOS and EFI capable and based on the grub bootloader which causes a requirement to the `-bootloaders` systemdeps.

`kiwi-systemdeps-filesystems`:

- Supports building `fs-type`, `oem`, `pxe`, `kis` and live `iso` image types.
- Installs all tools to create filesystems supported with KIWI NG. The pulled in components are needed for any image type that needs to create a filesystem. This excludes the archive based image types like `docker`, `appx` or `tbz`. The package also installs tools one level below the actual filesystem creation toolkit. These are components to manage loop devices as well as partition table setup and subsystem support like LVM and LUKS.
- Depends on the `-core` kiwi-systemdeps packages.

`kiwi-systemdeps-disk-images`:

- Supports building the `oem` image type.
- Installs all tools to create virtual disks. In KIWI NG, virtual disks are created using the QEMU toolchain.
- Depends on the `-filesystems` and `-bootloaders` kiwi-systemdeps packages.

`kiwi-systemdeps-image-validation`:

- Installs the `jing` tool to validate the image description. This is useful for detailed error reports from KIWI NG in case of an image description validation error. In addition, the `anymarkup` Python module is installed if the the option to install recommended packages is set. With `anymarkup` available, KIWI NG can also handle image descriptions in another format than the XML markup, like YAML.

Depending on the image type the kiwi-systemdeps packages can help to setup the host system quickly for the task to build an image. In case the host should support *everything* there is also the main `kiwi-systemdeps` package which has a dependency on all other existing systemdeps packages.

Note: Pulling in all `kiwi-systemdeps` packages can result in quite some packages to become installed on the host. This is because the required packages itself comes with a number of dependencies like java for jing as one example.

7.2 Setting up Repositories

A crucial part of each appliance is the repository selection. KIWI NG allows the end user to completely customize the selection of repositories and packages via the `repository` element.

7.2.1 Adding repositories

KIWI NG installs packages into your appliance from the repositories defined in the image description. Therefore at least one repository **must** be defined, as KIWI NG will otherwise not be able to fetch any packages.

A repository is added to the description via the `repository` element, which is a child of the top-level `image` element:

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <!-- snip -->
    <repository type="rpm-md" alias="kiwi" priority="1">
        <source path="obs://Virtualization:Appliances:Builder/
↪openSUSE_Leap_15.1"/>
    </repository>
    <repository type="rpm-md" alias="OS" imageinclude="true">
        <source path="obs://openSUSE:Leap:15.1/standard"/>
    </repository>
</image>
```

In the above snippet we defined two repositories:

1. The repository belonging to the KIWI NG project: *obs://Virtualization:Appliances:Builder/openSUSE_Leap_15.1* at the Open Build Service (OBS)
2. The RPM repository belonging to the OS project: *obs://openSUSE:Leap:15.1/standard*, at the Open Build Service (OBS). The translated http URL will also be included in the final appliance.

The `repository` element accepts one `source` child element, which contains the URL to the repository in an appropriate format and the following optional attributes:

- `imageinclude`: Specify whether this repository should be added to the resulting image, defaults to false.
- `imageonly`: A repository with `imageonly="true"` will not be available during image build, but only in the resulting appliance. Defaults to false.
- `priority`: An integer priority for all packages in this repository. If the same package is available in more than one repository, then the one with the highest priority is used.
- `alias`: Name to be used for this repository, it will appear as the repository's name in the image, which is visible via `zypper repos` or `dnf repolist`. KIWI NG will construct an alias from the path in the `source` child element (replacing each `/` with a _), if no value is given.
- `repository_gpgcheck`: Specify whether or not this specific repository is configured to to run repository signature validation. If not set, the package manager's default is used.
- `package_gpgcheck`: Boolean value that specifies whether each package's GPG signature will be verified. If omitted, the package manager's default will be used
- `components`: Distribution components used for `deb` repositories, defaults to `main`.
- `distribution`: Distribution name information, used for deb repositories.
- `profiles`: List of profiles to which this repository applies.

Supported repository paths

The actual location of a repository is specified in the `source` child element of `repository` via its only attribute `path`. KIWI NG supports the following paths types:

- `http://URL` or `https://URL` or `ftp://URL`: a URL to the repository available via HTTP(s) or FTP.
- `obs://$PROJECT/$REPOSITORY`: evaluates to the repository `$REPOSITORY` of the project `$PROJECT` available on the Open Build Service (OBS). By default KIWI NG will look for projects on build.opensuse.org, but this can be overridden using the runtime configuration file (see *The Runtime Configuration File*). Note that it is not possible to add repositories using the `obs://` path from **different** OBS instances (use direct URLs to the `.repo` file instead in this case).
- `obsrepositories:/`: special path only available for builds using the Open Build Service. The repositories configured for the OBS project in which the KIWI NG image resides will be available inside the appliance. This allows you to configure the repositories of your image from OBS itself and not having to modify the image description.
- `dir:///path/to/directory` or `file:///path/to/file`: an absolute path to a local directory or file available on the host building the appliance.
- `iso:///path/to/image.iso`: the specified ISO image will be mounted during the build of the KIWI NG image and a repository will be created pointing to the mounted ISO.

7.3 Adding and Removing Packages

On top of the *Setting up Repositories* setup the package setup is required. KIWI NG allows the end user to completely customize the selection of packages via the `packages` element.

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <packages type="bootstrap">
        <package name="udev"/>
        <package name="filesystem"/>
        <package name="openSUSE-release"/>
        <!-- additional packages installed before the chroot is␣
↪created -->
    </packages>
    <packages type="image">
        <package name="patterns-openSUSE-base"/>
        <!-- additional packages to be installed into the chroot -->
    </packages>
</image>
```

The `packages` element provides a collection of different child elements that instruct KIWI NG when and how to perform package installation or removal. Each `packages` element acts as a group, whose behavior can be configured via the following attributes:

- `type`: either `bootstrap`, `image`, `delete`, `uninstall` or one of the following build types: `docker`, `iso`, `oem`, `kis`, `oci`.

  Packages for `type="bootstrap"` are pre-installed to populate the images' root file system before chrooting into it.

  Packages in `type="image"` are installed immediately after the initial chroot into the new root file system.

  Packages in `type="delete"` and `type="uninstall"` are removed from the image, for details see *Uninstall System Packages*.

  And packages which belong to a build type are only installed when that specific build type is currently processed by KIWI NG.

- `profiles`: a list of profiles to which this package selection applies (see *Image Profiles*).
- `patternType`: selection type for patterns, supported values are: `onlyRequired`, `plusRecommended`, see: *The product and namedCollection element*.

The following sections describes the different child elements of a `packages` group.

7.3.1 The `package` element

The `package` element represents a single package to be installed (or removed), whose name is specified via the mandatory `name` attribute:

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <!-- snip -->
    <packages type="bootstrap">
        <package name="udev"/>
    </packages>
</image>
```

which adds the package `udev` to the list of packages to be added to the initial filesystem. Note, that the value that you pass via the `name` attribute is passed directly to the used package manager. Thus, if the package manager supports other means how packages can be specified, you may pass them in this context too. For example, RPM based package managers (like **dnf** or **zypper**) can install packages via their `Provides:`. This can be used to add a package that provides a certain capability (e.g. `Provides:  /usr/bin/my-binary`) via:

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <!-- snip -->
    <packages type="bootstrap">
        <package name="/usr/bin/my-binary"/>
    </packages>
</image>
```

Whether this works depends on the package manager and on the environment that is being used. In the Open Build Service, certain `Provides` either are not visible or cannot be properly extracted from the KIWI NG description. Therefore, relying on `Provides` is not recommended.

Packages can also be included only on specific host architectures via the `arch` attribute. KIWI NG compares the `arch` attributes value with the host architecture that builds the image according to the output of `uname -m`.

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <!-- snip -->
    <packages type="image">
        <package name="grub2"/>
        <package name="grub2-x86_64-efi" arch="x86_64"/>
        <package name="shim" arch="x86_64"/>
    </packages>
</image>
```

which results in `grub2-x86_64-efi` and `shim` being only installed if the build host is a 64bit x86 machine, but `grub2` will be installed independent of the architecture.

7.3.2 The `archive` element

It is sometimes necessary to include additional packages into the image which are not available in the package manager's native format. KIWI NG supports the inclusion of ordinary tar archives via the `archive` element, whose `name` attribute specifies the filename of the archive (KIWI NG looks for the archive in the image description folder).

```
<packages type="image">
    <archive name="custom-program1.tgz"/>
    <archive name="custom-program2.tar"/>
</packages>
```

KIWI NG will extract the archive into the root directory of the image using GNU tar, thus only archives supported by it can be included. When multiple `archive` elements are specified then they will be applied in a top to bottom order. If a file is already present in the image, then the file from the archive will overwrite it (same as with the image overlay).

7.3.3 Uninstall System Packages

KIWI NG supports two different methods how packages can be removed from the appliance:

1. Packages present as a child element of `<packages type="uninstall">` will be gracefully uninstalled by the package manager alongside with dependent packages and orphaned dependencies.
2. Packages present as a child element of `<packages type="delete">` will be removed by RPM/DPKG without any dependency check, thus potentially breaking dependencies and compromising the underlying package database.

Both types of removals take place after `config.sh` is run in the *prepare step* (see also *User Defined Scripts*).

Warning: An `uninstall` packages request deletes:

- the listed packages,
- the packages dependent on the listed ones, and
- any orphaned dependency of the listed packages.

Use this feature with caution as it can easily cause the removal of sensitive tools leading to failures in later build stages.

Removing packages via `type="uninstall"` can be used to completely remove a build time tool (e.g. a compiler) without having to specify a all dependencies of that tool (as one would have when using `type="delete"`). Consider the following example where we wish to compile a custom program in `config.sh`. We ship its source code via an `archive` element and add the build tools (`ninja`, `meson` and `clang`) to `<packages type="image">` and `<packages type="uninstall">`:

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <!-- snip -->
    <packages type="image">
        <package name="ca-certificates"/>
        <package name="coreutils"/>
        <package name="ninja"/>
        <package name="clang"/>
        <package name="meson"/>
        <archive name="foo_app_sources.tar.gz"/>
    </packages>
    <!-- These packages will be uninstalled after running config.sh␣
↪-->
    <packages type="uninstall">
        <package name="ninja"/>
        <package name="meson"/>
        <package name="clang"/>
    </packages>
</image>
```

The tools `meson`, `clang` and `ninja` are then available during the *prepare step* and can thus be used in `config.sh` (for further details, see *User Defined Scripts*), for example to build `foo_app`:

```
pushd /opt/src/foo_app
mkdir build
export CC=clang
meson build
cd build && ninja && ninja install
popd
```

The `<packages type="uninstall">` element will make sure that the final appliance will no longer contain our tools required to build `foo_app`, thus making our image smaller.

There are also other use cases for `type="uninstall"`, especially for specialized appliances. For containers one can often remove the package `shadow` (it is required to setup new user accounts) or any left over partitioning tools (`parted` or `fdisk`). All networking tools can be safely uninstalled in images for embedded devices without a network connection.

7.3.4 The `product` and `namedCollection` element

KIWI NG supports the inclusion of openSUSE products or of namedCollections (*patterns* in SUSE based distributions or *groups* for RedHat based distributions). These can be added via the `product` and `namedCollection` child elements, which both take the mandatory `name` attribute and the optional `arch` attribute.

`product` and `namedCollection` can be utilized to shorten the list of packages that need to be added to the image description tremendously. A named pattern, specified with the namedCollection element is a representation of a predefined list of packages. Specifying a pattern will install all packages listed in the named pattern. Support for patterns is distribution specific and available in SLES, openSUSE, CentOS, RHEL and Fedora. The optional `patternType` attribute on the packages element allows you to control the installation of dependent packages. You may assign one of the following values to the `patternType` attribute:

- `onlyRequired`: Incorporates only patterns and packages that the specified patterns and packages require. This is a "hard dependency" only resolution.
- `plusRecommended`: Incorporates patterns and packages that are required and recommended by the specified patterns and packages.

7.3.5 The `ignore` element

Packages can be explicitly marked to be ignored for installation inside a `packages` collection. This useful to exclude certain packages from being installed when using patterns with `patternType="plusRecommended"` as shown in the following example:

```xml
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <packages type="image" patternType="plusRecommended">
        <namedCollection name="network-server"/>
        <package name="grub2"/>
        <package name="kernel"/>
        <ignore name="ejabberd"/>
        <ignore name="puppet-server"/>
    </packages>
</image>
```

Packages can be marked as ignored during the installation by adding a `ignore` child element with the mandatory `name` attribute set to the package's name. Optionally one can also specify the architecture via the `arch` similarly to *The package element*.

Warning: Adding `ignore` elements as children of a `<packages type="delete">` or a `<packages type="uninstall">` element has no effect! The packages will still get deleted.

7.4 Image Profiles

A *profile* is a namespace for additional settings that can be applied by KIWI NG on top of the default settings (or other profiles), thereby allowing to build multiple appliances with the same build type but with different configurations.

The use of profiles is advisable to distinguish image builds of the same type but with different settings. In the following example, two virtual machine images of the `oem` type are configured: one for QEMU (using the `qcow2` format) and one for VMWare (using the `vmdk` format).

```xml
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <profiles>
```

(continues on next page)

(continued from previous page)

```
        <profile name="QEMU" description="virtual machine for QEMU"/
↪>
        <profile name="VMWare" description="virtual machine for␣
↪VMWare"/>
    </profiles>
    <preferences>
        <version>15.0</version>
        <packagemanager>zypper</packagemanager>
    </preferences>
    <preferences profiles="QEMU">
        <type image="oem" format="qcow2" filesystem="ext4">
    </preferences>
    <preferences profiles="VMWare">
        <type image="oem" format="vmdk" filesystem="ext4">
    </preferences>
</image>
```

Each profile is declared via the element `profile`, which itself must be a child of `profiles` and must contain the `name` and `description` attributes. The `description` is only present for documentation purposes, `name` on the other hand is used to instruct KIWI NG which profile to build via the command line. Additionally, one can provide the boolean attribute `import`, which defines whether this profile should be used by default when KIWI NG is invoked via the command line.

A profile inherits the default settings which do not belong to any profile. It applies only to elements that contain the profile in their `profiles` attribute. The attribute `profiles` expects a comma separated list of profiles for which the settings of this element apply.

Profiles can furthermore inherit settings from another profile via the `requires` sub-element:

```
<profiles>
    <profile name="VM" description="virtual machine"/>
    <profile name="QEMU" description="virtual machine for QEMU">
        <requires profile="VM"/>
    </profile>
</profiles>
```

The profile `QEMU` would inherit the settings from `VM` in the above example.

For further details on the usage of *profiles* see *Building Images with Profiles*

7.5 Adding Users

User accounts can be added or modified via the `users` element, which supports a list of multiple `user` child elements:

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <users>
        <user
            password="this_is_soo_insecure"
            home="/home/me" name="me"
            groups="users" pwdformat="plain"
        />
        <user
            password="$1$wYJUgpM5$RXMMeASDc035eX.NbYWFl0"
            home="/root" name="root" groups="root"
        />
    </users>
</image>
```

Each `user` element represents a specific user that is added or modified. The following attributes are mandatory:

- `name`: the UNIX username
- `password`: The password for this user account. It can be provided either in cleartext form (`pwdformat="plain"`) or in `crypt`'ed form (`pwdformat="encrypted"`). Plain passwords are discouraged, as everyone with access to the image description would know the password. It is recommended to generate a hash of your password using `openssl` as follows:

  ```
  $ openssl passwd -1 -salt 'xyz' YOUR_PASSWORD
  ```

Additionally, the following optional attributes can be specified:

- `home`: the path to the user's home directory
- `groups`: A comma separated list of UNIX groups. The first element of the list is used as the user's primary group. The remaining elements are appended to the user's supplementary groups. When no groups are assigned then the system's default primary group will be used.
- `id`: The numeric user id of this account.
- `pwdformat`: The format in which `password` is provided, either `plain` or `encrypted` (the latter is the default).

7.6 User Defined Scripts

Note: Abstract

This chapter describes the purpose of the user defined scripts `config.sh`, `image.sh` and `disk.sh`, which can be used to further customize an image in ways that are not possible via the image description alone.

KIWI NG supports the following optional scripts that it runs in a root environment (chroot) containing your new appliance:

config.sh runs at the end of the *prepare step* and after users have been set and the *overlay tree directory* has been applied. It is usually used to apply a permanent and final change of data in the root tree, such as modifying a package provided config file.

images.sh is executed at the beginning of the *image creation process*. It runs in the same image root tree that has been created by the prepare step but is invoked any time an image should be created from that root tree. It is usually used to apply image type specific changes to the root tree such as a modification to a config file that should be done when building a live iso but not when building a virtual disk image.

disk.sh is executed for the disk image type `oem` only and runs after the synchronisation of the root tree into the disk image loop file. At call time of the script the device name of the currently mapped root device is passed as a parameter. The chroot environment for this script call is the virtual disk itself and not the root tree as with `config.sh` and `images.sh`. The script `disk.sh` is usually used to apply changes at parts of the system that are not an element of the file based root tree such as the partition table, the bootloader or filesystem attributes.

KIWI NG executes scripts via the operating system if their executable bit is set (in that case a shebang is mandatory) otherwise they will be invoked via the BASH. If a script exits with a non-zero exit code then KIWI NG will report the failure and abort the image creation.

7.6.1 Script Template for config.sh / images.sh

KIWI NG provides a collection of methods and variables that supports users with custom operations. For details see *Functions and Variables Provided by KIWI NG*. The following template shows how to import this information in your script:

```
#======================================
# Include functions & variables
#--------------------------------------
test -f /.kconfig && . /.kconfig
test -f /.profile && . /.profile

...
```

Warning: Modifications of the unpacked root tree

Keep in mind that there is only one unpacked root tree the script operates in. This means that all changes are permanent and will not be automatically restored!

7.6.2 Functions and Variables Provided by KIWI NG

KIWI NG creates the `.kconfig` and `.profile` files to be sourced by the shell scripts `config.sh` and `images.sh`. `.kconfig` contains various helper functions which can be used to simplify the image configuration and `.profile` contains environment variables which get populated from the settings provided in the image description.

Functions

The `.kconfig` file provides a common set of functions. Functions specific to SUSE Linux Enterprise and openSUSE begin with the name `suse`, functions applicable to all Linux distributions start with the name `base`.

The following list describes all functions provided by `.kconfig`:

baseCleanMount Unmount the filesystems `/proc`, `/dev/pts`, `/sys` and `/proc/sys/fs/binfmt_misc`.

baseGetPackagesForDeletion Return the name(s) of the packages marked for deletion in the image description.

baseGetProfilesUsed Return the name(s) of profiles used to build this image.

baseSetRunlevel {value} Set the default run level.

baseSetupUserPermissions Set the ownership of all home directories and their content to the correct users and groups listed in `/etc/passwd`.

baseStripAndKeep {list of info-files to keep} Helper function for the `baseStrip*` functions, reads the list of files to check from stdin for removing params: files which should be kept

baseStripDocs {list of docu names to keep} Remove all documentation files, except for the ones given as the parameter.

baseStripInfos {list of info-files to keep} Remove all info files, except for the one given as the parameter.

baseStripLocales {list of locales} Remove all locales, except for the ones given as the parameter.

baseStripTranslations {list of translations} Remove all translations, except for the ones given as the parameter.

baseStripMans {list of manpages to keep} Remove all manual pages, except for the ones given as the parameter.

Example:

```
baseStripMans more less
```

suseImportBuildKey Add the SUSE build keys to the RPM database.

baseStripUnusedLibs Remove libraries which are not directly linked against applications in the bin directories.

baseUpdateSysConfig {filename} {variable} {value} Update the contents of a sysconfig variable

suseConfig This function is deprecated and is a NOP.

baseSystemdServiceInstalled {service} Prints the path of the first found systemd unit or mount with name passed as the first parameter.

baseSysVServiceInstalled {service} Prints the name `${service}` if a SysV init service with that name is found, otherwise it prints nothing.

baseSystemdCall {service_name} {args} Calls `systemctl ${args} ${service_name}` if a systemd unit, a systemd mount or a SysV init service with the `${service_name}` exist.

baseInsertService {servicename} Activate the given service via **`systemctl`**.

baseRemoveService {servicename} Deactivate the given service via **`systemctl`**.

baseService {servicename} {on|off} Activate or deactivate a service via **`systemctl`**. The function requires the service name and the value `on` or `off` as parameters.

Example to enable the sshd service on boot:

```
baseService sshd on
```

suseInsertService {servicename} Calls baseInsertService and exists only for compatibility reasons.

suseRemoveService {servicename} Calls baseRemoveService and exists only for compatibility reasons.

suseService {servicename} {on|off} Calls baseService and exists only for compatibility reasons.

suseActivateDefaultServices Activates the `network` and `cron` services to run at boot.

suseSetupProduct Creates the `/etc/products.d/baseproduct` link pointing to the product referenced by either `/etc/SuSE-brand` or `/etc/os-release` or the latest `prod` file available in `/etc/products.d`

suseSetupProductInformation Uses **`zypper`** to search for the installed product and installs all product specific packages. This function fails when **`zypper`** is not the appliances package manager.

baseVagrantSetup Configures the image to work as a vagrant box by performing the following changes:

- add the `vagrant` user to `/etc/sudoers` or `/etc/sudoers.d/vagrant`
- insert the insecure vagrant ssh key, apply recommended ssh settings and start the ssh daemon
- create the default shared folder `/vagrant`

Debug {message} Helper function to print the supplied message if the variable DEBUG is set to 1 (it is off by default).

Echo {echo commandline} Helper function to print a message to the controlling terminal.

Rm {list of files} Helper function to delete files and log the deletion.

Rpm {rpm commandline} Helper function for calling `rpm`: forwards all commandline arguments to `rpm` and logs the call.

Profile Environment Variables

The `.profile` environment file is created by KIWI NG and contains a specific set of variables which are listed below.

$kiwi_compressed The value of the `compressed` attribute set in the `type` element in `config.xml`.

$kiwi_delete A list of all packages which are children of the `packages` element with `type="delete"` in `config.xml`.

$kiwi_drivers A comma separated list of the driver entries as listed in the `drivers` section of the `config.xml`.

$kiwi_iname The name of the image as listed in `config.xml`.

$kiwi_iversion The image version as a string.

$kiwi_keytable The contents of the keytable setup as done in `config.xml`.

$kiwi_language The contents of the locale setup as done in `config.xml`.

$kiwi_profiles A comma separated list of profiles used to build this image.

$kiwi_timezone The contents of the timezone setup as done in `config.xml`.

$kiwi_type The image type as extracted from the `type` element in `config.xml`.

7.6.3 Configuration Tips

1. **Locale configuration:**

   KIWI in order to set the locale relies on **`systemd-firstboot`**, which in turn writes the locale configuration file `/etc/locale.conf`. The values for the locale settings are taken from the description XML file in the `<locale>` element under `<preferences>`.

KIWI assumes systemd adoption to handle these locale settings, in case the build distribution does not honor `/etc/locale.conf` this is likely to not produce any effect on the locale settings. As an example, in SLE12 distribution the locale configuration is already possible by using the systemd toolchain, however this approach overlaps with SUSE specific managers such as YaST. In that case using **`systemd-firstboot`** is only effective if locales in `/etc/sysconfig/language` are not set or if the file does not exist at all. In SLE12 `/etc/sysconfig/language` has precendence over `/etc/locale.conf` for compatibility reasons and management tools could still relay on `sysconfig` files for locale settings.

In any case the configuration is still possible in KIWI by using any distribution specific way to configure the locale setting inside the `config.sh` script or by adding any additional configuration file as part of the overlay root-tree.

1. **Stateless systemd UUIDs:**

   Machine ID files are created and set (`/etc/machine-id`, `/var/lib/dbus/machine-id`) during the image package installation when *systemd* and/or *dbus* are installed. Those UUIDs are intended to be unique and set only once in each deployment. KIWI NG follows the systemd recommendations and wipes any `/etc/machine-id` content, leaving it as an empty file. Note, this only applies to images based on a dracut initrd, it does not apply for container images.

   In case this setting is also required for a non dracut based image, the same result can achieved by removing `/etc/machine-id` in `config.sh`.

   Note: Avoid interactive boot

   It is important to remark that the file `/etc/machine-id` is set to an empty file instead of deleting it. **`systemd`** may trigger **`systemd-firstboot`** service if this file is not present, which leads to an interactive firstboot where the user is asked to provide some data.

   Note: Avoid inconsistent `/var/lib/dbus/machine-id`

   Note that `/etc/machine-id` and `/var/lib/dbus/machine-id` **must** contain the same unique ID. On modern systems `/var/lib/dbus/machine-id` is already a symlink to `/etc/machine-id`. However on older systems those might be two different files. This is the case for SLE-12 based images. If you are targeting these older operating systems, it is recommended to add the symlink creation into `config.sh`:

```
#======================================
# Make machine-id consistent with dbus
#--------------------------------------
if [ -e /var/lib/dbus/machine-id ]; then
    rm /var/lib/dbus/machine-id
```

(continues on next page)

(continued from previous page)

```
fi
ln -s /etc/machine-id /var/lib/dbus/machine-id
```

7.7 The Runtime Configuration File

KIWI NG supports an additional configuration file for runtime specific settings that do not belong into the image description but which are persistent and would be unsuitable for command line parameters.

The runtime configuration file must adhere to the YAML syntax and can be provided via the global `--config` option at call time of KIWI NG. If no config file is provided at the commandline, KIWI NG searches for the runtime configuration file in the following locations:

1. `~/.config/kiwi/config.yml`
2. `/etc/kiwi.yml`

A default runtime config file in `/etc/kiwi.yml` is provided with the python3-kiwi package. The file contains all settings as comments including a short description of each setting.

7.8 Customizing the Boot Process

Most Linux systems use a special boot image to control the system boot process after the system firmware, BIOS or UEFI, hands control of the hardware to the operating system. This boot image is called the `initrd`. The Linux kernel loads the `initrd`, a compressed cpio initial RAM disk, into the RAM and executes **`init`** or, if present, **`linuxrc`**.

Depending on the image type, KIWI NG creates the boot image automatically during the `create` step. It uses a tool called `dracut` to create this initrd. Dracut generated initrd archives can be extended by custom modules to add functionality which is not natively provided by dracut itself. In the scope of KIWI NG the following dracut modules are used:

`kiwi-dump` Serves as an image installer. It provides the required implementation to install a KIWI NG image on a selectable target. This module is required if one of the attributes `installiso`, `installstick` or `installpxe` is set to `true` in the image type definition

`kiwi-dump-reboot` Serves to boot the system into the installed image after installation is completed.

`kiwi-live` Boots up a KIWI NG live image. This module is required if the `iso` image type is selected

`kiwi-overlay` Allows to boot disk images configured with the attribute `overlayroot` set to `true`. Such a disk has its root partition compressed and readonly and boots up using overlayfs for the root filesystem using an extra partition on the same disk for persistent data.

kiwi-repart Resizes an OEM disk image after installation onto the target disk to meet the size constraints configured in the `oemconfig` section of the image description. The module takes over the tasks to repartition the disk, resizing of RAID, LVM, LUKS and other layers and resizing of the system filesystems.

kiwi-lib Provides functions of general use and serves as a library usable by other dracut modules. As the name implies, its main purpose is to function as library for the above mentioned kiwi dracut modules.

Note: Using Custom Boot Image Support

Apart from the standard dracut based creation of the boot image, KIWI NG supports the use of custom boot images for the image types `oem` and `pxe`. The use of a custom boot image is activated by setting the following attribute in the image description:

```
<type ... initrd_system="kiwi"/>
```

Along with this setting it is now mandatory to provide a reference to a boot image description in the `boot` attribute like in the following example:

```
<type ... boot="netboot/suse-leap15.1"/>
```

Such boot descriptions for the OEM and PXE types are currently still provided by the KIWI NG packages but will be moved into its own repository and package soon.

The custom boot image descriptions allows a user to completely customize what and how the initrd behaves by its own implementation. This concept is mostly used in PXE environments which are usually highly customized and requires a specific boot and deployment workflow.

7.8.1 Boot Image Hook-Scripts

The dracut initrd system uses `systemd` to implement a predefined workflow of services which are documented in the bootup man page at:

http://man7.org/linux/man-pages/man7/dracut.bootup.7.html

To hook in a custom boot script into this workflow it's required to provide a dracut module which is picked up by dracut at the time KIWI NG calls it. The module files can be either provided as a package or as part of the overlay directory in your image description

The following example demonstrates how to include a custom hook script right before the system rootfs gets mounted.

1. Create a subdirectory for the dracut module:

   ```
   $ mkdir -p root/usr/lib/dracut/modules.d/90my-module
   ```

2. Register the dracut module in a configuration file:

```
$ vi root/etc/dracut.conf.d/90-my-module.conf

add_dracutmodules+=" my-module "
```

3. Create the hook script:

```
$ touch root/usr/lib/dracut/modules.d/90my-module/my-script.sh
```

4. Create a module setup file in `root/usr/lib/dracut/modules.d/90my-module/module-setup.sh` with the following content:

```
#!/bin/bash

# called by dracut
check() {
    # check module integrity
}

# called by dracut
depends() {
    # return list of modules depending on this one
}

# called by dracut
installkernel() {
    # load required kernel modules when needed
    instmods _kernel_module_list_
}

# called by dracut
install() {
    declare moddir=${moddir}
    inst_multiple _tools_my_module_script_needs_

    inst_hook pre-mount 30 "${moddir}/my-script.sh"
}
```

That's it! At the time KIWI NG calls dracut the `90my-module` will be taken into account and is installed into the generated initrd. At boot time systemd calls the scripts as part of the `dracut-pre-mount.service`.

The dracut system offers a lot more possibilities to customize the initrd than shown in the example above. For more information, visit the dracut project page.

7.8.2 Boot Image Parameters

A dracut generated initrd in a KIWI NG image build process includes one or more of the KIWI NG provided dracut modules. The following list documents the available kernel boot parameters for this modules:

`rd.kiwi.debug` Activates the debug log file for the KIWI NG part of the boot process at `/run/initramfs/log/boot.kiwi`.

`rd.kiwi.install.pxe` Tells an OEM installation image to lookup the system image on a remote location specified in `rd.kiwi.install.image`.

`rd.kiwi.install.image=URI` Specifies the remote location of the system image in a PXE based OEM installation

`rd.kiwi.install.pass.bootparam` Tells an OEM installation image to pass an additional boot parameters to the kernel used to boot the installed image. This can be used e.g. to pass on first boot configuration for a PXE image. Note, that options starting with `rd.kiwi` are not passed on to avoid side effects.

`rd.kiwi.oem.maxdisk=size[KMGT]` Configures the maximum disk size an unattended OEM installation should consider for image deployment. Unattended OEM deployments default to deploying on `/dev/sda` (more exactly, the first device not filtered out by `oem-device-filter`). With RAID controllers, it can happen that your buch of big JBOD disks is for example `/dev/sda` to `/dev/sdi` and the 480G RAID1 configured for OS deployment is `/dev/sdj`. With `rd.kiwi.oem.maxdisk=500G` the deployment will land on that RAID disk.

`rd.live.overlay.size` Tells a live ISO image the size for the `tmpfs` filesystem that is used for the `overlayfs` mount process. If the write area of the overlayfs mount uses this tmpfs, any new data written during the runtime of the system will fillup this space. The default value used is set to `50%` which means one half of the available RAM space can be used for writing new data.

`rd.live.overlay.persistent` Tells a live ISO image to prepare a persistent write partition.

`rd.live.overlay.cowfs` Tells a live ISO image which filesystem should be used to store data on the persistent write partition.

`rd.live.cowfile.mbsize` Tells a live ISO image the size of the COW file in MB. When using tools like `live-grub-stick` the live ISO will be copied as a file on the target device and a GRUB loopback setup is created there to boot the live system from file. In such a case the persistent write setup, which usually creates an extra write partition on the target, will fail in almost all cases because the target has no free and unpartitioned space available. Because of that a cow file(live_system.cow) instead of a partition is created. The cow file will be created in the same directory the live iso image file was read from by grub and takes the configured size or the default size of 500MB.

`rd.live.dir` Tells a live ISO image the directory which contains the live OS root directory. Defaults to `LiveOS`.

`rd.live.squashimg` Tells a live ISO image the name of the squashfs image file which holds the OS root. Defaults to `squashfs.img`.

Boot Debugging

If the boot process encounters a fatal error, the default behavior is to stop the boot process without any possibility to interact with the system. Prevent this behavior by activating dracut's builtin debug mode in combination with the kiwi debug mode as follows:

```
rd.debug rd.kiwi.debug
```

This should be set at the Kernel command line. With those parameters activated, the system will enter a limited shell environment in case of a fatal error during boot. The shell contains a basic set of commands and allows for a closer look to:

```
less /run/initramfs/log/boot.kiwi
```

7.9 Overview

KIWI NG builds so-called *system images* (a fully installed and optionally configured system in a single file) of a Linux distribution in two steps (for further details, see *Image Building Process*):

1. *Prepare operation*: generate an *unpacked image tree* of your image. The unpacked tree is a directory containing the future file system of your image, generated from your image description.
2. *Create operation*: the unpacked tree generated in step 1 is packaged into the format required for the final usage (e.g. a `qcow2` disk image to launch the image with QEMU).

KIWI NG executes these steps using the following components, which it expects to find in the *description directory*:

***Image Description*:** The `config.xml` file contains the image description, which is a collection of general settings of the final image, like the image layout installed packages, present users, etc.

> **Note:** The filename `config.xml` is not mandatory, the image description file can also have an arbitrary name plus the `*.kiwi` extension. KIWI NG first looks for a `config.xml` file. If it cannot be found, it picks the first `*.kiwi` file.

***User Defined Scripts*:** If present, custom configuration shell scripts run at different stages of the build process. They can be used to fine tune the image in ways that are not possible via the settings provided in `config.xml`.

Overlay tree directory: The *overlay tree* is a folder (called `root`) or a tarball (called `root.tar.gz`) that contains files and directories that will be copied into the *unpacked image* *tree* during the *Prepare operation*. The copying is executed after all the packages included in `config.xml` have been installed. Any already present files are overwritten.

CD root user data: For live ISO images and install ISO images an optional archive is supported. This is a tar archive matching the name `config-cdroot.tar[.compression_postfix]`.

If present, the archive will be unpacked as user data on the ISO image. For example, this is used to add license files or user documentation. The documentation can then be read directly from the CD/DVD without booting from the media.

7.10 Image Building Process

KIWI NG creates images in a two step process: The first step, the *prepare* operation, generates a so-called *unpacked image tree* (directory) using the information provided in the `config.xml` configuration file (see *Image Description*)

The second step, the *create* operation, creates the *packed image* or *image* in the specified format based on the unpacked image tree and the information provided in the `config.xml` configuration file.

7.10.1 The Prepare Step

As the first step, KIWI NG creates an *unpackaged image tree*, also called “root tree”. This directory will be the installation target for software packages to be installed during the image creation process.

For the package installation, KIWI NG relies on the package manager specified in the `packagemanager` element in `config.xml`. KIWI NG supports the following package managers: `dnf`, `zypper` (default) and `apt-get`.

The prepare step consists of the following substeps:

1. **Create Target Root Directory**

   KIWI NG aborts with an error if the target root tree already exists to avoid accidental deletion of an existing unpacked image.

2. **Install Packages**

   First, KIWI NG configures the package manager to use the repositories specified in the configuration file, via the command line, or both. After the repository setup, the packages specified in the `bootstrap` section of the image description are installed in a temporary directory external to the target root tree. This establishes the initial environment to support the completion of the process in a chroot setting. The essential packages are `filesystem` and `glibc-locale` to specify as part of the bootstrap. The dependency chain of these two packages is usually sufficient to populate the bootstrap environment with all required software to support the installation of packages into the new root tree. The aforementioned two packages might not be enough for every distribution. Consult the kiwi-descriptions repository containing examples for various Linux distributions.

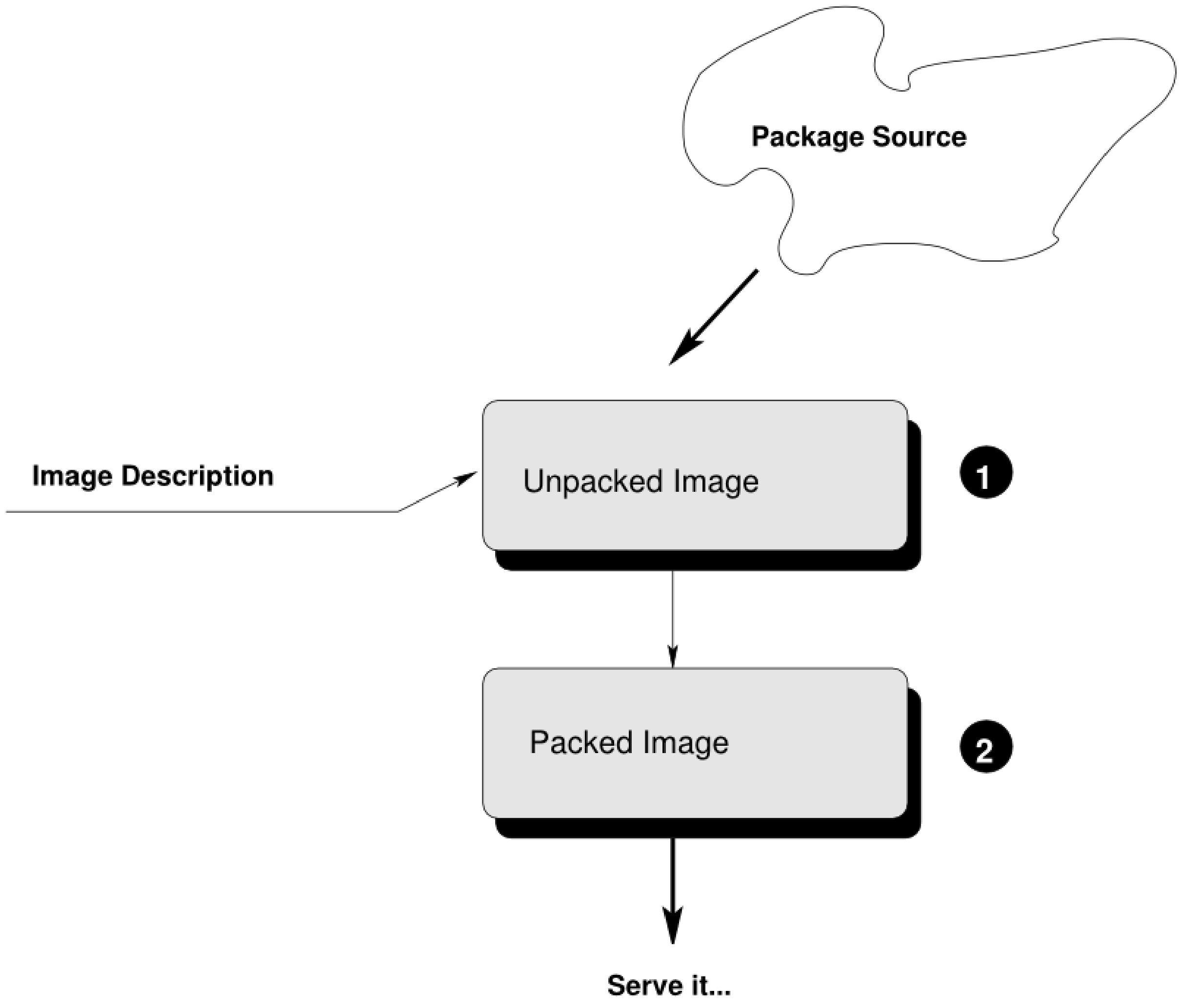

Fig. 1: Image Creation Architecture

The installation of software packages through the selected package manager may install unwanted packages. Removing these packages can be accomplished by marking them for deletion in the image description, see *Uninstall System Packages*.

3. **Apply the Overlay Tree**

   Next, KIWI NG applies all files and directories present in the overlay directory named `root` or in the compressed overlay `root.tar.gz` to the target root tree. Files already present in the target root directory are overwritten. This allows you to overwrite any file that was installed by one of the packages during the installation phase.

4. **Apply Archives**

   All archives specified in the `archive` element of the `config.xml` file are applied in the specified order (top to bottom) after the overlay tree copy operation is complete (see *The archive element*). Files and directories are extracted relative to the top level of the new root tree. As with the overlay tree, it is possible to overwrite files already existing in the target root tree.

5. **Execute the user-defined script** `config.sh`

   At the end of the preparation stage the script `config.sh` is executed (if present). It is run in the top level directory of the target root tree. The script's primary function is to complete the system configuration. For more details about custom scripts see *User Defined Scripts*

6. **Modify the Root Tree**

   The unpacked image tree is now finished to be converted into the final image in the *create step*. It is possible to make manual modifications to the unpacked tree before it is converted into the final image.

   Since the unpacked image tree is just a directory, it can be modified using the standard tools. Optionally, it is also possible to "change root (**`chroot`**)" into it, for instance to invoke the package manager. Beside the standard file system layout, the unpacked image tree contains an additional directory named `/image` that is not present in a regular system. It contains information KIWI NG requires during the create step, including a copy of the `config.xml` file.

   By default, KIWI NG will not stop after the *prepare step* and will directly proceed with the *create step*. Therfore to perform manual modifications, proceed as follows:

```
$ kiwi-ng system prepare $ARGS
$ # make your changes
$ kiwi-ng system create $ARGS
```

   Warning: Modifications of the unpacked root tree

   Do not make any changes to the system, since they are lost when re-running the `prepare` step again. Additionally, you may introduce errors that occur during the `create` step which are difficult to track. The recommended way to apply changes to the unpacked image directory is to change the configuration and re-run the `prepare` step.

7.10.2 The Create Step

KIWI NG creates the final image during the *create step*: it converts the unpacked root tree into one or multiple output files appropriate for the respective build type.

It is possible to create multiple images from the same unpacked root tree, for example, a self installing OEM image and a virtual machine image from the same image description. The only prerequisite is that both image types are specified in `config.xml`.

During the *create step* the following operations are performed by KIWI NG:

1. **Execute the User-defined Script** `images.sh`

   At the beginning of the image creation process the script named `images.sh` is executed (if present). For more details about custom scripts see *User Defined Scripts*

2. **Create the Requested Image Type**

   KIWI NG converts the unpacked root into an output format appropriate for the requested build type.

CHAPTER

EIGHT

IMAGE DESCRIPTION

Note: This document explains the toplevel structure of the KIWI NG image description document for version 9.23.20

8.1 Image Description Elements

Note: This document provides a reference for the elements and attributes of the KIWI NG XML document in version 9.23.20

8.1.1 <image>

The toplevel of any KIWI NG image description

```
<image schemaversion="7.2" name="Leap-15.1_appliance">
    <!-- descendants -->
</image>
```

The image definition starts with an image tag and requires the schema format at version 7.2. The attribute name specifies the name of the image which is also used for the filenames created by KIWI. Because we don't want spaces in filenames the name attribute must not have any spaces in its name.

The following optional attributes can be inserted in the image tag:

displayname Allows setup of the boot menu title for the selected boot loader. So you can have *suse-SLED-foo* as the image name but a different name as the boot display name. Spaces are not allowed in the display name because it causes problems for some boot loaders and kiwi did not take the effort to separate the ones which can display them correctly from the ones which can't

id sets an identification number which appears as file `/etc/ImageID` within the image.

8.1.2 <description>

Provide an image identity.

```
<description type="system">
  <author>name</author>
  <contact>contact</contact>
  <specification>text</specification>
</description>
```

The mandatory description section contains information about the creator of this image description. The attribute type could be either of the value `system` which indicates this is a system image description or at value `boot` for custom kiwi boot image descriptions.

The following optional sub sections can be inserted below the description tag:

license Specifies the license name which applies to this image description.

8.1.3 <preferences>

Setup image type and layout.

```
<preferences arch="arch">
  <version>1.2.3</version>
  <packagemanager name="zypper"/>
  <type image="tbz"/>
</preferences>
```

The mandatory preferences section contains information about the supported image type(s), the used package manager, the version of this image, and further optional elements. The preferences section can be configured to apply only for a certain architecture. In this case specify the `arch` attribute with a value as it is reported by **`uname -m`**

<preferences><version>

The mandatory image version must be a three-part version number of the format: **Major**.**Minor**.**Release**. In case of changes to the image description the following rules should apply:

- For smaller image modifications that do not add or remove any new packages, only the release number is incremented. The XML description file(`config.xml`) remains unchanged.
- For image changes that involve the addition or removal of packages the minor number is incremented and the release number is reset.
- For image changes that changes the behavior or geometry of the image file the major number is incremented.

<preferences><packagemanager>

The mandatory packagemanager element specifies which package manager should be used to handle software packages. The packagemanager setup is connected to the distribution used to build the image. The following table shows which package manager is connected to which distributor:

Distributor	Package Manager
SUSE	zypper
RedHat	dnf
Debian Based	apt-get
Arch Linux	pacman

In general the specification of one preferences section is sufficient. However, it's possible to specify multiple preferences sections and distinguish between the sections via the profiles attribute.

In combination with the above the preferences element supports the following optional elements:

<preferences><rpm-locale-filtering>

locale-filtering can be set to "true" or "false". If set to "true" it sets the install_lang macro for RPM based installations to the RPM configured locale list. This results in language specific files to become filtered out by `rpm` if they don't match the configured list.

```
<preferences>
  <rpm-locale-filtering>true</rpm-locale-filtering>
</preferences>
```

Note: It depends on the individual package design if the install_lang macro contents apply to the package or not.

<preferences><rpm-check-signatures>

Specifies whether package signatures should be checked or not

```
<preferences>
  <rpm-check-signatures>true</rpm-check-signatures>
</preferences>
```

<preferences><rpm-excludedocs>

Specifies whether files marked as documentation should be skipped during installation

```
<preferences>
  <rpm-excludedocs>true</rpm-excludedocs>
</preferences>
```

<preferences><keytable>

Specifies the name of the console keymap to use. The value corresponds to a map file in `/usr/share/kbd/keymaps/xkb`.

```
<preferences>
  <keytable>us</keytable>
</preferences>
```

<preferences><timezone>

Specifies the time zone. Available time zones are located in the `/usr/share/zoneinfo` directory. Specify the attribute value relative to `/usr/share/zoneinfo`. For example, specify Europe/Berlin for `/usr/share/zoneinfo/Europe/Berlin`.

```
<preferences>
  <keytable>Europe/Berlin</keytable>
</preferences>
```

<preferences><locale>

Specifies the name of the UTF-8 locale to use, which defines the contents of the RC_LANG system environment variable used in the image and to run the custom scripts specified as part of the KIWI NG image description. Please note only UTF-8 locales are supported here which also means that the encoding must *not* be part of the locale information. This means you need to specify the locale using the 4-digit name like the following example: en_US or en_US,de_DE

```
<preferences>
  <locale>en_US</locale>
</preferences>
```

<preferences><bootsplash-theme>

Specifies the name of the plymouth bootsplash theme to use

```
<preferences>
  <bootsplash-theme>bgrt</bootsplash-theme>
</preferences>
```

<preferences><bootloader-theme>

Specifies the name of the bootloader theme to use if that used bootloader has theme support.

```
<preferences>
  <bootloader-theme>openSUSE</bootloader-theme>
</preferences>
```

Along with the version and the packagemanager at least one image type element must be specified to indicate which image type should be build.

<preferences><type>

At least one type element must be configured. It is possible to specify multiple type elements in a preferences block. To set a given type description as the default image use the boolean attribute primary and set its value to true:

```
<preferences>
  <type image="typename" primary="true"/>
</preferences>
```

The image type to be created is determined by the value of the image attribute. The following list describes the supported types and possible values of the image attribute:

image=”tbz” A simple tar archive image. The tbz type packs the contents of the image root tree into a xz compressed tarball.

image=”btrfs|ext2|ext3|ext4|squashfs|xfs” A filesystem image. The image root tree data is packed into a filesystem image of the given type. An image of that type can be loop mounted and accessed according to the capabiities of the selected filesystem.

image=”iso” An iso image which can be dumped on a CD/DVD or USB stick and boots off from this media without interfering with other system storage components. A useful pocket system for testing and demo and debugging purposes.

image=”oem” An image representing an expandable system disk. This means after deployment the system can resize itself to the new disk geometry. The resize operation is configurable as part of the image description and an installation image for CD/DVD, USB stick and Network deployment can be created in addition. For use in cloud frameworks like Amazon EC2, Google Compute Engine or Microsoft Azure this disk type also supports the common virtual disk formats.

image="docker" An archive image suitable for the docker container engine. The image can be loaded via the `docker load` command and works within the scope of the container engine

image="oci" An archive image that builds a container matching the OCI (Open Container Interface) standard. The container should be able to run with any oci compliant container engine.

image="appx" An archive image suitable for the Windows Subsystem For Linux container engine. The image can be loaded From a Windows System that has support for WSL activated.

image="kis" An optional root filesystem image associated with a kernel and initrd. The use case for this component image type is highly customizable. Many different deployment strategies are possible.

For completion of a type description, there could be several other optional attributes and child elements. The `type` element supports a plethora of optional attributes, some of these are only relevant for certain build types and will be covered in extra chapters that describes the individual image types more detailed. Certain attributes are however useful for nearly all build types and will be covered next:

bootpartition="true|false": Boolean parameter notifying KIWI NG whether an extra boot partition should be used or not (the default depends on the current layout). This will override KIWI NG's default layout.

bootpartsize="nonNegativeInteger": For images with a separate boot partition this attribute specifies the size in MB. If not set the boot partition size is set to 200 MB

efipartsize="nonNegativeInteger": For images with an EFI fat partition this attribute specifies the size in MB. If not set the EFI partition size is set to 20 MB

efiparttable="msdos|gpt": For images with an EFI firmware specifies the partition table type to use. If not set defaults to the GPT partition table type

btrfs_quota_groups="true|false": Boolean parameter to activate filesystem quotas if the filesystem is `btrfs`. By default quotas are inactive.

btrfs_root_is_snapshot="true|false": Boolean parameter that tells KIWI NG to install the system into a btrfs snapshot. The snapshot layout is compatible with snapper. By default snapshots are turned off.

btrfs_root_is_readonly_snapshot="true|false": Boolean parameter notifying KIWI NG that the btrfs root filesystem snapshot has to made read-only. if this option is set to true, the root filesystem snapshot it will be turned into read-only mode, once all data has been placed to it. The option is only effective if `btrfs_root_is_snapshot` is also set to true. By default the root filesystem snapshot is writable.

compressed="true|false": Specifies whether the image output file should be compressed or not. This option is only used for filesystem only images or for the `pxe` or `cpio` types.

editbootconfig="file_path": Specifies the path to a script which is called right before the bootloader is installed. The script runs relative to the directory which contains the image structure.

editbootinstall="file_path": Specifies the path to a script which is called right after the bootloader is installed. The script runs relative to the directory which contains the image structure.

filesystem="btrfs|ext2|ext3|ext4|squashfs|xfs": The root filesystem

firmware="efi|uefi": Specifies the boot firmware of the appliance, supported options are: `bios`, `ec2`, `efi`, `uefi`, `ofw` and `opal`. This attribute is used to differentiate the image according to the firmware which boots up the system. It mostly impacts the disk layout and the partition table type. By default `bios` is used on x86, `ofw` on PowerPC and `efi` on ARM.

force_mbr="true|false": Boolean parameter to force the usage of a MBR partition table even if the system would default to GPT. This is occasionally required on ARM systems that use a EFI partition layout but which must not be stored in a GPT. Note that forcing a MBR partition table incurs limitations with respect to the number of available partitions and their sizes.

fsmountoptions="option_string": Specifies the filesystem mount options which are passed via the `-o` flag to **mount** and are included in `/etc/fstab`.

fscreateoptions="option_string": Specifies the filesystem options used to create the filesystem. In KIWI NG the filesystem utility to create a filesystem is called without any custom options. The default options are filesystem specific and are provided along with the package that provides the filesystem utility. For the Linux `ext[234]` filesystem, the default options can be found in the `/etc/mke2fs.conf` file. Other filesystems provides this differently and documents information about options and their defaults in the respective manual page, e.g **man mke2fs**. With the `fscreateoptions` attribute it's possible to directly influence how the filesystem will be created. The options provided as a string are passed to the command that creates the filesystem without any further validation by KIWI NG. For example, to turn off the journal on creation of an ext4 filesystem the following option would be required:

```
<preferences>
  <type fscreateoptions="-O ^has_journal"/>
</preferences>
```

kernelcmdline="string": Additional kernel parameters passed to the kernel by the bootloader.

luks="passphrase": Supplying a value will trigger the encryption of the partitions using the LUKS extension and using the provided string as the password. Note that the password must be entered when booting the appliance!

target_blocksize="number": Specifies the image blocksize in bytes which has to match the logical blocksize of the target storage device. By default 512 Bytes is used, which works on many disks. You can obtain the blocksize from the `SSZ` column in the output of the following command:

```
blockdev --report $DEVICE
```

target_removable="true|false": Indicate if the target disk for oem images is deployed to a removable device e.g a USB stick or not. This only affects the EFI setup if requested and in the end avoids the creation of a custom boot menu entry in the firmware of the target machine. By default the target disk is expected to be non-removable

spare_part="number": Request a spare partition right before the root partition of the requested size. The attribute takes a size value and allows a unit in MB or GB, e.g 200M. If no unit is given the value is considered to be mbytes. A spare partition can only be configured for the disk image type oem

spare_part_mountpoint="dir_path": Specify mount point for spare partition in the system. Can only be configured for the disk image type oem

spare_part_fs="btrfs|ext2|ext3|ext4|xfs": Specify filesystem for spare partition in the system. Can only be configured for the disk image type oem

spare_part_fs_attributes="attribute_list": Specify filesystem attributes for the spare partition. Attributes can be specified as comma separated list. Currently the attributes `no-copy-on-write` and `synchronous-updates` are available. Can only be configured for the disk image type oem

spare_part_is_last="true|false": Specify if the spare partition should be the last one in the partition table. Can only be configured for the `oem` type with oem-resize switched off. By default the root partition is the last one and the spare partition lives before it. With this attribute that setup can be toggled. However, if the root partition is no longer the last one the oem repart/resize code can no longer work because the spare part would block it. Because of that moving the spare part at the end of the disk is only applied if oem-resize is switched off. There is a runtime check in the KIWI NG code to check this condition

devicepersistency="by-uuid|by-label|by-path": Specifies which method to use for persistent device names. This will affect all files written by kiwi that includes device references for example `etc/fstab` or the `root=` parameter in the kernel commandline. By default by-uuid is used

squashfscompression="uncompressed|gzip|lzo|lz4|xz|zstd": Specifies the compression type for mksquashfs

overlayroot="true|false" For the `oem` type only, specifies to use an `overlayfs` based root filesystem consisting out of a squashfs compressed read-only root filesystem combined with a write-partition or tmpfs. The optional kernel boot parameter `rd.root.overlay.readonly` can be used to point the write area into a `tmpfs` instead of the existing persistent write-partition. In this mode all written data is temporary until reboot of the system. The kernel boot parameter `rd.root.overlay.size` can be used to configure the size for the `tmpfs` that is used for the `overlayfs` mount process if `rd.root.overlay.readonly` is requested. That size basically configures the amount of space available for writing new data during the runtime of the system. The default value is set to `50%` which means one half of the available RAM space can be used for writing new data. By default the persistent write-partition is used. The size of that partition can be influenced via the optional `<size>` element in the `<type>` section or via the optional `<oem-resize>` element in the `<oemconfig>` section of the XML description. Setting a fixed `<size>` value will set the size of the image disk to that value and results in an image file of that size. The available space for the write partition is that size reduced by the size the squashfs read-only system needs. If the `<oem-resize>` element is set to `true` an eventually given `<size>` element will not have any effect because the write partition will be resized on first boot to the available disk space.

bootfilesystem="ext2|ext3|ext4|fat32|fat16": If an extra boot partition is required this attribute specify which filesystem should be used for it. The type of the selected bootloader might overwrite this setting if there is no alternative possible though.

flags="overlay|dmsquash" For the iso image type specifies the live iso technology and dracut module to use. If set to overlay the kiwi-live dracut module will be used to support a live iso system based on squashfs+overlayfs. If set to dmsquash the dracut standard dmsquash-live module will be used to support a live iso system based on the capabilities of the upstream dracut module.

format="gce|ova|qcow2|vagrant|vmdk|vdi|vhd|vhdx|vhd-fixed": For disk image type oem, specifies the format of the virtual disk such that it can run on the desired target virtualization platform.

formatoptions="string": Specifies additional format options passed on to qemu-img formatoptions is a comma separated list of format specific options in a name=value format like qemu-img expects it. kiwi will take the information and pass it as parameter to the -o option in the qemu-img call

fsmountoptions="string": Specifies the filesystem mount options which also ends up in fstab The string given here is passed as value to the -o option of mount

fscreateoptions="string": Specifies options to use at creation time of the filesystem

force_mbr="true|false": Force use of MBR (msdos table) partition table even if the use of the GPT would be the natural choice. On e.g some arm systems an EFI partition layout is required but must not be stored in a GPT. For those rare cases this attribute allows to force the use of the msdos table including all its restrictions in max partition size and amount of partitions

gpt_hybrid_mbr="true|false": For GPT disk types only: Create a hybrid GPT/MBR partition table

hybridpersistent="true|false": For the live ISO type, triggers the creation of a partition for a COW file to keep data persistent over a reboot

hybridpersistent_filesystem="ext4|xfs": For the live ISO type, set the filesystem to use for persistent writing if a hybrid image is used as disk on e.g a USB Stick. By default the ext4 filesystem is used.

initrd_system="kiwi|dracut": Specify which initrd builder to use, default is set to `dracut`

metadata_path="dir_path": Specifies a path to additional metadata required for the selected image type or its tools used to create that image type.

Note: Currently this is only effective for the appx container image type.

installboot="failsafe-install|harddisk|install": Specifies the bootloader default boot entry for the initial boot of a KIWI NG install image.

Note: This value is only evaluated for grub

install_continue_on_timeout=”true|false”: Specifies the boot timeout handling for the KIWI NG install image. If set to “true” the configured timeout or its default value applies. If set to “false” no timeout applies in the boot menu of the install image.

installprovidefailsafe=”true|false”: Specifies if the bootloader menu should provide an fail-safe entry with special kernel parameters or not

installiso=”true|false” Specifies if an install iso image should be created. This attribute is only available for the `oem` type. The generated ISO image is an hybrid ISO which can be used as disk on e.g a USB stick or as ISO.

installpxe=”true|false”: Specifies if a tarball that contains all data for a pxe network installation should be created. This attribute is only available for the `oem` type.

mediacheck=”true|false”: For ISO images, specifies if the bootloader menu should provide an mediacheck entry to verify ISO integrity or not. Disabled by default and only available for the x86 arch family.

mdraid=”mirroring|striping”: Setup software raid in degraded mode with one disk Thus only mirroring and striping is possible

primary=”true|false”: Specifies this type to be the primary type. If no type option is given on the commandline, KIWI NG will build this type

ramonly=”true|false”: For all images that are configured to use the overlay filesystem this setting forces any COW(Copy-On-Write) action to happen in RAM.

rootfs_label=”string”: Label name to set for the root filesystem. By default `ROOT` is used

volid=”string”: For the ISO type only, specifies the volume ID (volume name or label) to be written into the master block. There is space for 32 characters.

vhdfixedtag=”GUID_string”: For the VHD disk format, specifies the GUID

derived_from=”string”: For container images, specifies the image URI of the container image. The image created by KIWI NG will use the specified container as the base root to work on.

publisher=”string”: For ISO images, specifies the publisher name of the ISO.

The following sections shows the supported child elements of the `type` element including references to their usage in a detailed type setup:

<preferences><type><bootloader>

Used to describe the bootloader setup in the oem disk image type. For details see: *Setting up the Bootloader of the Image*

<preferences><type><containerconfig>

Used to describe the container configuration metadata in docker or wsl image types. For details see: *Build a Docker Container Image* and: *Build a WSL Container Image*

<preferences><type><vagrantconfig>

Used to describe vagrant configuration metadata in a disk image that is being used as a vagrant box. For details see: *Image Description for Vagrant*

<preferences><type><systemdisk>

Used to describe the geometry, partitions and volumes, in a disk image. For details see: *Custom Disk Volumes*

<preferences><type><oemconfig>

Used to customize the deployment process in an oem disk image. For details see: *OEM Customization*

<preferences><type><size>

Used to customize the size of the resulting disk image in an oem image. For details see: *Modifying the Size of the Image*

<preferences><type><machine>

Used to customize the virtual machine configuration which describes the components of an emulated hardware. For details see: *Customizing the Virtual Machine*

<preferences><type><installmedia>

Used to customize the installation media images created for oem images deployment. For details see: *Installation Media Customization*

8.1.4 <repository>

Setup software sources for the image.

```
<repository>
  <source path="uri"/>
</repository>
```

The mandatory repository element specifies the location and type of a repository to be used by the package manager as a package installation source. KIWI NG supports apt, dnf, pacman and zypper as package managers, specified with the packagemanager element. The repository element has the following optional attributes:

alias="name" Specifies an alternative name for the configured repository. If the attribute is not specified KIWI NG will generate a random alias name for the repository.

components="name" Used for Debian (apt) based repositories only. Specifies the component name that should be used from the repository. By default the `main` component is used

distribution="name" Used for Debian (apt) based repositories only. Specifies the distribution name to be used on call of `debootstrap`

imageonly="true|false" Specifies whether or not this repository should be configured in the resulting image without using it at build time. By default the value is set to false

repository_gpgcheck="true|false" Specifies whether or not this specific repository is configured to to run repository signature validation. If not set, no value is appended into the repository configuration file. If set the relevant key information needs to be provided on the KIWI NG commandline using the `--signing-key` option

imageinclude="true|false" Specifies whether the given repository should be configured as a repository in the image or not. The default behavior is that repositories used to build an image are not configured as a repository inside the image. This feature allows you to change the behavior by setting the value to true.

Note: Scope of repository uri's

The repository is configured in the image according to the source path as specified with the path attribute of the source element. Therefore, if the path is not a fully qualified URL, you may need to adjust the repository file in the image to accommodate the expected location. It is recommended that you use the alias attribute in combination with the imageinclude attribute to avoid having unpredictable random names assigned to the repository you wish to include in the image.

password="string" Specifies a password for the given repository. The password attribute must be used in combination with the username attribute. Dependent on the repository location this information may not be used.

username="name" Specifies a user name for the given repository. The username attribute must be used in combination with the password attribute. Dependent on the repository location this information may not be used.

prefer-license="true|false" The repository providing this attribute will be used primarily to install the license tarball if found on that repository. If no repository with a preferred license attribute exists, the search happens over all repositories. It's not guaranteed in that case that the search order follows the repository order like they are written into the XML description.

priority="number" Specifies the repository priority for this given repository. Priority values are treated differently by different package managers. Repository priorities allow the package management system to disambiguate packages that may be contained in more than one of the configured repositories. The zypper package manager for example prefers packages from a repository with a *lower* priority over packages from a repository with higher priority values. The value 99 means "no priority is set". For other package managers please refer to the individual documentation about repository priorities.

sourcetype="baseurl|metalink|mirrorlist" Specifies the source type of the repository path. Depending on if the source path is a simple url or a pointer to a metadata file or mirror list, the configured package manager needs to be setup appropriately. By default the source is expected to be a simple repository baseurl.

use_for_bootstrap="true|false" Used for Debian (apt) based repositories only. It specifies whether this repository should be the one used for bootstrapping or not. It is set to 'false' by default. Only a single repository is allowed to be used for bootstrapping, if no repository is set for the bootstrap the last one in the description XML is used.

<repository><source>

The location of a repository is specified by the path attribute of the mandatory source child element:

```
<repository alias="kiwi">
  <source path="obs://Virtualization:Appliances:Builder/openSUSE_
↪Leap_15.1"/>
</repository>
```

The location specification may include the `%arch` macro which will expand to the architecture of the image building host. The value for the path attribute may begin with any of the following location indicators:

- `dir:///local/path` An absolute path to a directory accessible through the local file system.
- `ftp://<ftp://>` A ftp protocol based network location.
- `http://<http://>` A http protocol based network location.

- `https://<https://>` A https protocol based network location.

Note: https repositories

When specifying a https location for a repository it is generally necessary to include the openssl certificates and a cracklib word dictionary as package entries in the bootstrap section of the image configuration. The names of the packages to include are individual to the used distribution. On SUSE systems as one example this would be `openssl-certs` and `cracklib-dict-full`

- `iso://<iso://>` An absolute path to an .iso file accessible via the local file system. KIWI NG will loop mount the the .iso file to a temporary directory with a generated name. The generated path is provided to the specified package manager as a directory based repository location.
- `obs://Open:Build:Service:Project:Name` A reference to a project in the Open Build Service (OBS). KIWI NG translates the given project path into a remote url at which the given project hosts the packages.
- `obsrepositories:/` A placeholder for the Open Build Service (OBS) to indicate that all repositories are taken from the project configuration in OBS.

8.1.5 <packages>

Setup software components to be installed in the image.

```
<packages type="type"/>
```

The mandatory packages element specifies the setup of a packages group for the given type. The value of the type attribute specifies at which state in the build process the packages group gets handled, supported values are as follows:

type="bootstrap" Bootstrap packages, list of packages to be installed first into a new (empty) root tree. The packages list the required components to support a chroot environment from which further software components can be installed

type="image" Image packages, list of packages to be installed as part of a chroot operation inside of the new root tree.

type="uninstall|delete" Packages to be uninstalled or deleted. For further details see *Uninstall System Packages*

type="*image_type_name*" Packages to be installed for the given image type name. For example if set to type="iso", the packages in this group will only be installed if the iso image type is build.

The packages element must contain at least one child element of the following list to provide specific configuration information for the specified packages group:

<packages><package>

```
<packages type="image"/>
  <package name="name" arch="arch"/>
</packages>
```

The package element installs the given package name. The optional `arch` attribute can be used to limit the installation of the package to the host architecture from which KIWI NG is called. The `arch` attribute is also available in all of the following elements.

<packages><namedCollection>

```
<packages type="image" patternType="onlyRequired">
  <namedCollection name="base"/>
</packages>
```

The namedCollection element is used to install a number of packages grouped together under a name. This is a feature of the individual distribution and used in the implementation of the KIWI NG package manager backend. At the moment collections are only supported for SUSE and RedHat based distributions. The optional `patternType` attribute is used to control the behavior of the dependency resolution of the package collection. `onlyRequired` installs only the collection and its required packages. `plusRecommended` installs the collection, any of its required packages and any recommended packages.

Note: Collections on SUSE

On SUSE based distributions collections are names patterns and are just simple packages. To get the names of the patterns such that they can be used in a namedCollection type the following command: `$ zypper patterns`. If for some reason the collection name cannot be used it is also possible to add the name of the package that provides the collection as part of a `package` element. To get the names of the pattern packages type the following command: `$ zypper search patterns`. By convention all packages that starts with the name “patterns-” are representing a pattern package.

Note: Collections on RedHat

On RedHat based distributions collections are named groups and are extra metadata. To get the names of these groups type the following command: `$ dnf group list`. Please note that group names are allowed to contain whitespace characters.

<packages><archive>

```
<packages type="image"/>
  <archive name="name"/>
</packages>
```

The archive element takes the `name` attribute and looks up the given name as file on the system. If specified relative KIWI NG looks up the name in the image description directory. The archive is installed using the `tar` program. Thus the file name is expected to be a tar archive. The compression of the archive is detected automatically by the tar program.

<packages><ignore>

```
<packages type="image"/>
  <ignore name="name"/>
</packages>
```

The ignore element instructs the used package manager to ignore the given package name at installation time. Please note whether or not the package can be ignored is up to the package manager. Packages that are hard required by other packages in the install procedure cannot be ignored and the package manager will simply ignore the request.

<packages><product>

```
<packages type="image">
  <product name="name"/>
</packages>
```

The product element instructs the used package manager to install the given product. What installation of a product means is up to the package manager and also distribution specific. This feature currently only works on SUSE based distributions

8.1.6 <users>

Setup image users.

```
<users>
  <user
    name="user"
    groups="group_list"
    home="dir"
    id="number"
    password="text"
    pwdformat="encrypted|plain"
    realname="name"
    shell="path"
```

(continues on next page)

(continued from previous page)

```
  />
</users>
```

The optional users element contains the user setup KIWI NG should create in the system. At least one user child element must be specified as part of the users element. Multiple user elements may be specified.

Each `user` element represents a specific user that is added or modified. The following attributes are mandatory:

name="name": the UNIX username

password="string" The password for this user account. It can be provided either in cleartext form or encrypted. An encrypted password can be created using `openssl` as follows:

```
$ openssl passwd -1 -salt xyz PASSWORD
```

It is also possible to specify the password as a non encrypted string by using the pwdformat attribute and setting it's value to `plain`. KIWI NG will then encrypt the password prior to the user being added to the system.

> **Warning:** plain text passwords
>
> We do not recommend plain passwords as they will be readable in the image configuration in plain text

All specified users and groups will be created if they do not already exist. The defined users will be part of the group(s) specified with the groups attribute or belong to the default group as configured in the system. If specified the first entry in the groups list is used as the login group.

Additionally, the following optional attributes can be specified:

home="path": The path to the user's home directory

groups="group_a,group_b": A comma separated list of UNIX groups. The first element of the list is used as the user's primary group. The remaining elements are appended to the user's supplementary groups. When no groups are assigned then the system's default primary group will be used.

id="number": The numeric user id of this account.

pwdformat="plain|encrypted": The format in which `password` is provided. The default if not specified is `encrypted`.

8.1.7 <profiles>

Manage image namespace(s).

```
<profiles>
  <profile name="name" description="text"/>
</profiles>
```

The optional profiles section lets you maintain one image description while allowing for variation of other sections that are included. A separate profile element must be specified for each variation. The profile child element, which has name and description attributes, specifies an alias name used to mark sections as belonging to a profile, and a short description explaining what this profile does.

For example to mark a set of packages as belonging to a profile, simply annotate them with the profiles attribute as shown below:

```
<packages type="image" profiles="profile_name">
  <package name="name"/>
</packages>
```

It is also possible to mark sections as belonging to multiple profiles by separating the names in the profiles attribute with a comma:

```
<packages type="image" profiles="profile_A,profile_B">
  <package name="name"/>
</packages>
```

If a section tag does not have a profiles attribute, it is globally present in the configuration. If global sections and profiled sections contains the same sub-sections, the profiled sections will overwrite the global sections in the order of the provided profiles. For a better overview of the result configuration when profiles are used we recommend to put data that applies in any case to non profiled (global) sections and only extend those global sections with profiled data. For example:

```
<preferences>
  <version>1.2.3</version>
  <packagemanager name="zypper"/>
</preferences>

<preferences profiles="oem_qcow_format">
  <type image="oem" filesystem="ext4" format="qcow2"/>
</preferences>

<preferences profiles="oem_vmdk_format">
  <type image="oem" filesystem="ext4" format="vmdk"/>
</preferences>
```

The above example configures two version of the same oem type. One builds a disk in qcow2 format the other builds a disk in vmdk format. The global preferences section without a profile assigned will be used in any case and defines those preferences settings that are common to any build process. A user can select both profiles at a time but that will result in building the disk format that is specified last because one is overwriting the other.

Use of one or more profile(s) during image generation is triggered by the use of the `--profile` command line argument. multiple profiles can be selected by passing this option multiple times.

8.2 Main Root

```
<image/>
```

The mandatory *<image>* element represents the root (top-level element) of an image description. All other elements must be descendants of this element. There can be only one `image` element.

8.3 Image Identity

```
<description/>
```

The mandatory *<description>* element contains information about the author, contact, license and the specification about the use case of this image. All data together forms the identity card of the image. There can be only one `description` element

8.4 Image Preferences

```
<preferences/>
```

The mandatory *<preferences>* element contains information to classify the image and to describe the layout. All data about the image type, its version, the partition layout and much more is specified here. There can be multiple `preferences` elements

8.5 Image Software Sources

```
<repository/>
```

The mandatory *<repository>* element contains information where to find the software packages that are used to build the image. There can be multiple `repository` elements

8.6 Image Content Setup

```
<packages/>
```

The mandatory *<packages>* element contains information to list which software should be installed from the configured repositories into the image. Software can be defined as names for packages, collections, archives or products. There can be multiple `packages` elements

8.7 Image Users

```
<users/>
```

The optional *<users>* element contains information about system users to be created inside of the image. There can be multiple `users` elements

8.8 Image Namespace

```
<profiles/>
```

The optional *<profiles>* element contains information to create one or more namespaces to an image description. The namespace can be used with any of the above elements and therefore tie them into a namespace which can be selected at call time of KIWI NG

CHAPTER

NINE

IMAGE TYPES

Note: Before building an image with KIWI NG it's important to understand the different image types and their meaning. This document provides an overview about the supported KIWI NG image types, their results and some words about the environment to run the build.

ISO Hybrid Live Image An iso image which can be dumped on a CD/DVD or USB stick and boots off from this media without interfering with other system storage components. A useful pocket system for testing and demo and debugging purposes. For further details refer to *Build an ISO Hybrid Live Image*

Virtual Disk Image An image representing the system disk, useful for cloud frameworks like Amazon EC2, Google Compute Engine or Microsoft Azure. For further details refer to *Build a Virtual Disk Image*

OEM Expandable Disk Image An image representing an expandable system disk. This means after deployment the system can resize itself to the new disk geometry. The resize operation is configurable as part of the image description and an installation image for CD/DVD, USB stick and Network deployment can be created in addition. For further details refer to: *Build an Expandable Disk Image*

Docker Container Image An archive image suitable for the docker container engine. The image can be loaded via the `docker load` command and works within the scope of the container engine. For further details refer to: *Build a Docker Container Image*

WSL Container Image An archive image suitable for the Windows Subsystem For Linux container engine. The image can be loaded From a Windows System that has support for WSL activated. For further details refer to: *Build a WSL Container Image*

KIS Root File System Image An optional root filesystem image associated with a kernel and initrd. The use case for this component image type is highly customizable. Many different deployment strategies are possible. For further details refer to: *Build KIS Image (Kernel, Initrd, System)*

9.1 Image Results

KIWI NG execution results in an appliance image after a successful run of *kiwi-ng system build* or *kiwi-ng system create* command. The result is the image binary in addition a couple of metadata files that can be handy to describe or identify the resulting image. The output files follow this naming convention:

`<image-name>.<arch>-<version>.<extension>`

where `<image-name>` is the name stated in the *Image Description* as an attribute of the *<image>* element. The `<arch>` is the CPU architecture used for the build, `<version>` is the image version defined in *<version>* element of the image description and the `<extension>` is dependent on the image type and its definition.

Any KIWI NG appliance build results in, at least, the following output files:

1. The image binary, `<image-name>.<arch>-<version>.<image-extension>`:

   This is the file containig the actual image binary, depending on the image type and its definition it can be a virtual disk image file, and ISO image, a tarball, etc.

2. The `<image-name>.<arch>-<version>.packages` file:

   This file includes a sorted list of the packages that are included into the image. In fact this is normalized dump of the package manager database. It follows the following cvs format where each line is represented by:

   `<name>|<epoch>|<version>|<release>|<arch>|<disturl>|<license>`

   The values represented here are mainly based on RPM packages metadata. Other package managers may not provide all of these values, in such cases the format is the same and the fields that cannot be provided are set as `None` value. This list can be used to track changes across multiple builds of the same image description over time by diffing the packages installed.

3. The `<image-name>.<arch>-<version>.verified` file:

   This file is the output of a verification done by the package manager against the package data base. More specific it is the output of the **rpm** verification process or **dpkg** verification depending on the packaging technology selected for the image. In both cases the output follows the RPM verification syntax. This provides an overview of all packages status right before any boot of the image.

More specific the result files for a given image name and version such as `Leap-15.1_appliance` and `1.15.1` will be:

- **image packages**: `Leap-15.1_appliance.x86_64-1.15.1.packages`
- **image verified**: `Leap-15.1_appliance.x86_64-1.15.1.verified`

In addition to the image binaries itself that depend on the image type:

image="tbz" For this image type the result is mainly a root tree packed in a tarball:

- **root archive**: `Leap-15.1_appliance.x86_64-1.15.1.tar.xz`

image="btrfs|ext2|ext3|ext4|squashfs|xfs" The image root tree data is packed into a filesystem image of the given type, hence the resutl for an `ext4` image would be:

- **filesystem image**: `Leap-15.1_appliance.x86_64-1.15.1.ext4`

image="iso" The image result is an ISO file:

- **live image**: `Leap-15.1_appliance.x86_64-1.15.1.iso`

image="oem" An image representing an expandable disk image. KIWI NG can also produce an installation ISO for this disk image by setting `installiso="true"` in the *<preferences><type>*) section or a tarball including the artifacts for a network deployment by setting `installiso="true"`. For further details see *Build an Expandable Disk Image*. The results for `oem` can be:

- **disk image**: `Leap-15.1_appliance.x86_64-1.15.1.raw`
- **installation image (optional)**: `Leap-15.1_appliance.x86_64-1.15.1.install.iso`
- **installation pxe archive (optional)**: `Leap-15.1_appliance.x86_64-1.15.1.install.tar`

The disk image can also be provided in one of the various virtual disk formats which can be specified in `format` attribute of the *<preferences><type>* section. For further details see *Build a Virtual Disk Image*. The result for e.g `format="qcow2"` would be:

- **disk image**: `Leap-15.1_appliance.x86_64-1.15.1.qcow2`

instead of the `raw` default disk format.

image="docker" An archive image suitable for the docker container engine. The result is a loadable (**`docker load -i <image>`**) tarball:

- **container**: `Leap-15.1_appliance.x86_64-1.15.1.docker.tar.xz`

image="oci" An archive image that builds a container matching the OCI (Open Container Interface) standard. The result is a tarball matching OCI standards:

- **container**: `Leap-15.1_appliance.x86_64-1.15.1.oci.tar.xz`

image="appx" An archive image suitable for the Windows Subsystem For Linux container engine. The result is an `appx` binary file:

- **container**: `Leap-15.1_appliance.x86_64-1.15.1.appx`

image="kis" An optional root filesystem image associated with a kernel and initrd. All three binaries are packed in a tarball, see *Build KIS Image (Kernel, Initrd, System)* for further details about the kis archive:

- **kis archive**: `Leap-15.1_appliance.x86_64-1.15.1.tar.xz`

CHAPTER

TEN

BUILDING IMAGES FOR SUPPORTED TYPES

Note: This document provides an overview how to build and use the KIWI NG supported image types.

10.1 Build an ISO Hybrid Live Image

Abstract

This page explains how to build a live image. It contains:

- how to build an ISO image
- how to run it with QEMU

A Live ISO image is a system on a removable media, e.g CD/DVD or USB stick. Once built and deployed it boots off from this media without interfering with other system storage components making it a useful pocket system for testing and demo- and debugging-purposes.

To add a live ISO build to your appliance, create a `type` element with `image` set to `iso` in your `config.xml` as shown below:

```
<image schemaversion="7.2" name="Tumbleweed_appliance">
  <!-- snip -->
  <preferences>
    <type image="iso" primary="true" flags="overlay"␣
↪hybridpersistent_filesystem="ext4" hybridpersistent="true"/>
    <!-- additional preferences -->
  </preferences>
  <!-- snip -->
</image>
```

The following attributes of the `type` element are relevant when building live ISO images:

- `flags`: Specifies the live ISO technology and dracut module to use, can be set to `overlay` or to `dmsquash`.

If set to `overlay`, the kiwi-live dracut module will be used to support a live ISO system based on squashfs and overlayfs. If set to `dmsquash`, the dracut standard dmsquash-live module will be used to support a live ISO system based on squashfs and the device mapper. Note, both modules support a different set of live features. For details see *Decision for a live ISO technology*

- `hybridpersistent`: Accepts `true` or `false`, if set to `true` then the resulting image will be created with a COW file to keep data persistent over a reboot
- `hybridpersistent_filesystem`: The filesystem used for the COW file. Possible values are `ext4` or `xfs`, with `ext4` being the default.

With the appropriate settings present in `config.xml` KIWI NG can now build the image:

```
$ sudo kiwi-ng --type iso system build \
      --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
      --target-dir /tmp/myimage
```

The resulting image is saved in the folder `/tmp/myimage` and can be tested with QEMU:

```
$ qemu -cdrom Leap-15.1_appliance.x86_64-1.15.1.iso -m 4096
```

The image is now complete and ready to use. See *Deploy ISO Image on an USB Stick* and *Deploy ISO Image as File on a FAT32 Formated USB Stick* for further information concerning deployment.

10.1.1 Decision for a live ISO technology

The decision for the `overlay` vs. `dmsquash` dracut module depends on the features one wants to use. From a design perspective the `overlay` module is conceived for live ISO deployments on disk devices which allows the creation of a write partition or cow file. The `dmsquash` module is conceived as a generic mapping technology using device-mapper snapshots. The following list describes important live ISO features and their support status compared to the `overlay` and `dmsquash` modules.

ISO scan Usable in the same way with both dracut modules. This feature allows to boot the live ISO as a file from a grub loopback configured bootloader. The `live-grub-stick` tool is just one example that uses this feature. For details how to setup ISO scan with the `overlay` module see *Deploy ISO Image as File on a FAT32 Formated USB Stick*

ISO in RAM completely Usable with the `dmsquash` module through `rd.live.ram`. The `overlay` module does not support this mode but KIWI NG supports RAM only systems as OEM deployment into RAM from an install ISO media. For details how to setup RAM only deployments in KIWI NG see: *Deploy and Run System in a RamDisk*

Overlay based on overlayfs Usable with the `overlay` module. A squashfs compressed readonly root gets overlayed with a readwrite filesystem using the kernel overlayfs filesystem.

Overlay based on device mapper snapshots Usable with the `dmsquash` module. A squashfs compressed readonly root gets overlayed with a readwrite filesystem using a device mapper snapshot. This method was the preferred one before overlayfs existed in the Linux kernel.

Media Checksum Verification Boot the live iso only for ISO checksum verification. This is possible with both modules but the `overlay` module uses the `checkmedia` tool whereas the upstream `dmsquash` module uses `checkisomd5`. The activation of the verification process is done by passing the kernel option `mediacheck` for the `overlay` module and `rd.live.check` for the `dmsquash` module.

Live ISO through PXE boot Boot the live image via the network. This is possible with both modules but uses different technologies. The `overlay` module supports network boot only in combination with the AoE (Ata Over Ethernet) protocol. For details see *Booting a Live ISO Image from Network*. The `dmsquash` module supports network boot by fetching the ISO image into memory from `root=live:` using the `livenet` module.

Persistent Data Keep new data persistent on a writable storage device. This can be done with both modules but in different ways. The `overlay` module activates persistency with the kernel boot parameter `rd.live.overlay.persistent`. If the persistent setup cannot be created the fallback to the non persistent mode applies automatically. The `overlay` module auto detects if it is used on a disk or ISO scan loop booted from a file. If booted as disk, persistency is setup on a new partition of that disk. If loop booted from file, persistency is setup on a new cow file. The cow file/partition setup can be influenced with the kernel boot parameters: `rd.live.overlay.cowfs` and `rd.live.cowfile.mbsize`. The `dmsquash` module configures persistency through the `rd.live.overlay` option exclusively and does not support the automatic creation of a write partition in disk mode.

dmsquash documentation

Documentation for the upstream `dmsquash` module can be found here. Options to setup `dmsquash` are marked with `rd.live`

10.2 Build a Virtual Disk Image

Abstract

This page explains how to build a simple disk image. It contains how to:

- define a simple disk image in the image description
- build a simple disk image
- run it with QEMU

A simple Virtual Disk Image is a compressed system disk with additional metadata useful for cloud frameworks like Amazon EC2, Google Compute Engine, or Microsoft Azure. It is used as the native disk of a system and does not require an extra installation workflow or a complex first boot setup procedure which is why we call it a *simple* disk image.

To instruct KIWI NG to build a simple disk image add a `type` element with `image="oem"` in `config.xml` that has the `oem-resize` feature disabled. An example configuration for a 42 GB large VMDK image with 512 MB RAM, an IDE controller and a bridged network interface is shown below:

```xml
<image schemaversion="7.2" name="Tumbleweed_appliance">
  <!-- snip -->
  <preferences>
    <type image="oem" filesystem="ext4" format="vmdk">
      <bootloader name="grub2" timeout="0"/>
      <size unit="G">42</size>
      <oemconfig>
          <oem-resize>false</oem-resize>
      </oemconfig>
      <machine memory="512" guestOS="suse" HWversion="4">
        <vmdisk id="0" controller="ide"/>
        <vmnic driver="e1000" interface="0" mode="bridged"/>
      </machine>
    </type>
    <!-- additional preferences -->
  </preferences>
  <!-- snip -->
</image>
```

The following attributes of the `type` element are of special interest when building simple disk images:

- `format`: Specifies the format of the virtual disk, possible values are: `gce`, `ova`, `qcow2`, `vagrant`, `vmdk`, `vdi`, `vhd`, `vhdx` and `vhd-fixed`.
- `formatoptions`: Specifies additional format options passed to **qemu-img**. `formatoptions` is a comma separated list of format specific options in a `name=value` format like **qemu-img** expects it. KIWI NG will forward the settings from this attribute as a parameter to the `-o` option in the **qemu-img** call.

The `bootloader`, `size` and `machine` child-elements of `type` can be used to customize the virtual machine image further. We describe them in the following sections: *Setting up the Bootloader of the Image*, *Modifying the Size of the Image* and *Customizing the Virtual Machine*

Once your image description is finished (or you are content with a image from the *example descriptions* and use one of them) build the image with KIWI NG:

```
$ sudo kiwi-ng --type oem system build \
    --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
    --target-dir /tmp/myimage
```

The created image will be in the target directory `/tmp/myimage` with the file extension `.raw`.

The live image can then be tested with QEMU:

```
$ qemu \
    -drive file=Leap-15.1_appliance.x86_64-1.15.1.raw,format=raw,
↪if=virtio \
    -m 4096
```

For further information how to setup the image to work within a cloud framework see:

- *Image Description for Amazon EC2*
- *Image Description for Microsoft Azure*
- *Image Description for Google Compute Engine*

For information how to setup a Vagrant box, see: *Image Description for Vagrant*.

10.2.1 Setting up the Bootloader of the Image

```
<preferences>
  <bootloader name="grub2"/>
</preferences>
```

The `bootloader` element is used to select the bootloader. At the moment `grub2`, `isolinux`, `zipl` and `grub2_s390x_emu` (a combination of zipl and a userspace GRUB2) are supported. The special `custom` entry allows to skip the bootloader configuration and installation and leaves this up to the user which can be done by using the `editbootinstall` and `editbootconfig` custom scripts.

In addition to the mandatory name attribute the following optional attributes are supported:

console="console|gfxterm|serial": Specifies the bootloader console. The attribute is available for the grub and isolinux bootloader types. By default a graphics console setup is used

serial_line="string": Specifies the bootloader serial line setup. The setup is effective if the bootloader console is set to use the serial line. The attribute is available for the grub bootloader only

timeout="number": Specifies the boot timeout in seconds prior to launching the default boot option. By default the timeout is set to 10 seconds. It makes sense to set this value to `0` for images intended to be started non-interactively (e.g. virtual machines).

timeout_style="countdown|hidden": Specifies the boot timeout style to control the way in which the timeout interacts with displaying the menu. If set the display of the bootloader menu is delayed after the timeout expired. In countdown mode an indication of the remaining time is displayed. The attribute is available for the grub loader only

targettype="CDL|LDL|FBA|SCSI": Specifies the device type of the disk zipl should boot. On zFCP devices use `SCSI`, on DASD devices use `CDL` or `LDL` on emulated DASD devices use `FBA`. The attribute is available for the zipl loader only

10.2.2 Modifying the Size of the Image

The `size` child element of `type` specifies the size of the resulting disk image. The following example shows a image description where 20 GB are added to the virtual machine image of which 5 GB are left unpartitioned:

```xml
<preferences>
  <type image="oem" format="vmdk">
    <size unit="G" additive="true" unpartitioned="5">20</size>
    <oemconfig>
        <oem-resize>false</oem-resize>
    </oemconfig>
  </type>
</preferences>
```

The following optional attributes can be used to customize the image size further:

- `unit`: Defines the unit used for the provided numerical value, possible settings are `M` for megabytes and `G` for gigabytes. The default unit are megabytes.
- `additive`: boolean value that determines whether the provided value will be added to the current image's size (`additive="true"`) or whether it is the total size (`additive="false"`). The default is `false`.
- `unpartitioned`: Specifies the image space in the image that will not be partitioned. This value uses the same unit as defined in the attribute `unit` or the default.

10.2.3 Customizing the Virtual Machine

The `machine` child element of `type` can be used to customize the virtual machine configuration which is used when the image is run, like the number of CPUs or the connected network interfaces.

The following attributes are supported by the `machine` element:

- `ovftype`: The OVF configuration type. The Open Virtualization Format is a standard for describing virtual appliances and distribute them in an archive called Open Virtual Appliance (OVA). The standard describes the major components associated with a disk image. The exact specification depends on the product using the format.

  Supported values are `zvm`, `powervm`, `xen` and `vmware`.
- `HWversion`: The virtual machine's hardware version (`vmdk` and `ova` formats only), see https://kb.vmware.com/s/article/1003746 for further details which value to choose.
- `arch`: the VM architecture (`vmdk` format only), possible values are: `ix86` (= `i585` and `i686`) and `x86_64`.
- `xen_loader`: the Xen target loader which is expected to load this guest, supported values are: `hvmloader`, `pygrub` and `pvgrub`.
- `guestOS`: The virtual guest OS' identification string for the VM (only applicable for `vmdk` and `ova` formats, note that the name designation is different for the two formats).

- `min_memory`: The virtual machine's minimum memory in MB (`ova` format only).
- `max_memory`: The virtual machine's maximum memory in MB (`ova` format only).
- `min_cpu`: The virtual machine's minimum CPU count (`ova` format only).
- `max_cpu`: The virtual machine's maximum CPU count (`ova` format only).
- `memory`: The virtual machine's memory in MB (all formats).
- `ncpus`: The umber of virtual CPUs available to the virtual machine (all formats).

Additionally, `machine` supports additional child elements that are covered in the following subsections.

Modifying the VM Configuration Directly

The `vmconfig-entry` element is used to add entries directly into the virtual machine's configuration file. This is currently only supported for the `vmdk` format where the provided strings are directly pasted into the `.vmx` file.

The `vmconfig-entry` element has no attributes and can appear multiple times, the entries are added to the configuration file in the provided order. Note, that KIWI NG does not check the entries for correctness. KIWI NG only forwards them.

The following example adds the two entries `numvcpus = "4"` and `cpuid.coresPerSocket = "2"` into the VM configuration file:

```
<preferences>
  <type image="oem" filesystem="ext4" format="vmdk">
    <machine memory="512" guestOS="suse" HWversion="4">
      <vmconfig-entry>numvcpus = "4"</vmconfig-entry>
      <vmconfig-entry>cpuid.coresPerSocket = "2"</vmconfig-entry>
    </machine>
  </type>
</preferences>
```

Adding Network Interfaces to the VM

Network interfaces can be explicitly specified for the VM when required via the `vmnic` element. This can be used to add another bridged interface or to specify the driver which is being used.

Note, that this element is only used for the `vmdk` image format.

In the following example we add a bridged network interface using the `e1000` driver:

```
<preferences>
  <type image="oem" filesystem="ext4" format="vmdk">
    <machine memory="4096" guestOS="suse" HWversion="4">
      <vmnic driver="e1000" interface="0" mode="bridged"/>
    </machine>
```

(continues on next page)

(continued from previous page)

```
  </type>
</preferences>
```

The `vmnic` element supports the following attributes:

- `interface`: **mandatory** interface ID for the VM's network interface.
- `driver`: optionally the driver which will be used can be specified
- `mac`: this interfaces' MAC address
- `mode`: this interfaces' mode.

Note that KIWI NG will **not** verify the values that are passed to these attributes, it will only paste them into the appropriate configuration files.

Specifying Disks and Disk Controllers

The `vmdisk` element can be used to customize the disks and disk controllers for the virtual machine. This element can be specified multiple times, each time for each disk or disk controller present.

Note that this element is only used for `vmdk` and `ova` image formats.

The following example adds a disk with the ID 0 using an IDE controller:

```
<preferences>
  <type image="oem" filesystem="ext4" format="vmdk">
    <machine memory="512" guestOS="suse" HWversion="4">
      <vmdisk id="0" controller="ide"/>
    </machine>
  </type>
</preferences>
```

Each `vmdisk` element can be further customized via the following optional attributes:

- `controller`: The disk controller used for the VM guest (`vmdk` format only). Supported values are: `ide`, `buslogic`, `lsilogic`, `lsisas1068`, `legacyESX` and `pvscsi`.
- `device`: The disk device to appear in the guest (`xen` format only).
- `diskmode`: The disk mode (`vmdk` format only), possible values are: `monolithicSparse`, `monolithicFlat`, `twoGbMaxExtentSparse`, `twoGbMaxExtentFlat` and `streamOptimized` (see also https://www.vmware.com/support/developer/converter-sdk/conv60_apireference/vim.OvfManager.CreateImportSpecParams.DiskProvisioningType.html).
- `disktype`: The type of the disk as it is internally handled by the VM (`ova` format only). This attribute is currently unused.
- `id`: The disk ID of the VM disk (`vmdk` format only).

Adding CD/DVD Drives

KIWI NG supports the addition of IDE and SCSCI CD/DVD drives to the virtual machine using the `vmdvd` element for the `vmdk` image format. In the following example we add two drives: one with a SCSCI and another with a IDE controller:

```
<preferences>
  <type image="oem" filesystem="ext4">
    <machine memory="512" xen_loader="hvmloader">
      <vmdvd id="0" controller="scsi"/>
      <vmdvd id="1" controller="ide"/>
    </machine>
  </type>
</preferences>
```

The `vmdvd` element features just these two **mandatory** attributes:

- `id`: The CD/DVD ID of the drive
- `controller`: The CD/DVD controller used for the VM guest, supported values are `ide` and `scsi`.

10.3 Build an Expandable Disk Image

Abstract

This page explains how to build an expandable disk image. It contains how to:

- build an expandable disk image
- deploy an expandable disk image
- run the deployed system

An expandable disk represents the system disk with the capability to auto expand the disk and its filesystem to a custom disk geometry. This allows deploying the same disk image on target systems with different hardware setup.

The following example shows how to build and deploy such a disk image based on openSUSE Leap using a QEMU virtual machine as the target system:

1. Make sure you have checked out the example image descriptions, see *Example Appliance Descriptions*.
2. Build the image with KIWI NG:

```
$ sudo kiwi-ng --type oem system build \
    --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
    --target-dir /tmp/myimage
```

Find the following result images below /tmp/myimage.

- The disk image with the suffix .raw is an expandable virtual disk. It can expand itself to a custom disk geometry.
- The installation image with the suffix install.iso is a hybrid installation system which contains the disk image and is capable to install this image on any target disk.

10.3.1 Deployment Methods

The basic idea behind an expandable disk image is to provide the virtual disk data for OEM vendors to support easy deployment of the system to physical storage media.

There are the following basic deployment strategies:

1. *Manual Deployment*

   Manually deploy the disk image onto the target disk.

2. *CD/DVD Deployment*

   Boot the installation image and let KIWI NG's installer deploy the disk image from CD/DVD or USB stick onto the target disk.

3. *Network Deployment*

   PXE boot the target system and let KIWI NG's installer deploy the disk image from the network onto the target disk.

10.3.2 Manual Deployment

The manual deployment method can be tested using virtualization software such as QEMU, and an additional virtual target disk of a larger size. The following steps shows how to do it:

1. Create a target disk

   ```
   $ qemu-img create target_disk 20g
   ```

   Note: Retaining the Disk Geometry

   If the target disk geometry is less or equal to the geometry of the disk image itself, the disk expansion performed for a physical disk install during the boot workflow will be skipped and the original disk geometry stays untouched.

2. Dump disk image on target disk.

   ```
   $ dd if=Leap-15.1_appliance.x86_64-1.15.1.raw of=target_disk␣
   ↪conv=notrunc
   ```

3. Boot the target disk

```
$ qemu -hda target_disk -m 4096
```

At first boot of the target_disk the system is expanded to the configured storage layout. By default the system root partition and filesystem is resized to the maximum free space available.

10.3.3 CD/DVD Deployment

The deployment from CD/DVD via the installation image can also be tested using virtualization software such as QEMU. The following steps shows how to do it:

1. Create a target disk

   Follow the steps above to create a virtual target disk

2. Boot the installation image as CD/DVD with the target disk attached.

```
$ qemu -cdromLeap-15.1_appliance.x86_64-1.15.1.install.iso -hda␣
↪target_disk -boot d -m 4096
```

Note: USB Stick Deployment

Like any other ISO image built with KIWI NG, also the installation image is a hybrid image. Thus it can also be used on USB stick and serve as installation stick image like it is explained in *Build an ISO Hybrid Live Image*

10.3.4 Network Deployment

The deployment from the network downloads the disk image from a PXE boot server. This requires a PXE network boot server to be setup as explained in *Setting Up a Network Boot Server*

If the PXE server is running the following steps shows how to test the deployment process over the network using a QEMU virtual machine as target system:

1. Make sure to create an installation PXE TAR archive along with your disk image by replacing the following setup in kiwi-descriptions/suse/x86_64/suse-leap-15.1/config.xml

   Instead of

```
<type image="oem" installiso="true"/>
```

   setup

```
<type image="oem" installpxe="true"/>
```

2. Rebuild the image, unpack the resulting `Leap-15.1_appliance.x86_64-1.15.1.install.tar.xz` file to a temporary directory and copy the initrd and kernel images to the PXE server:

   a) Unpack installation tarball

   ```
   mkdir /tmp/pxe && cd /tmp/pxe
   tar -xf Leap-15.1_appliance.x86_64-1.15.1.install.tar.xz
   ```

   b) Copy kernel and initrd used for pxe boot

   ```
   scp pxeboot.Leap-15.1_appliance.x86_64-1.15.1.initrd.xz PXE_
    ↪SERVER_IP:/srv/tftpboot/boot/initrd
   scp pxeboot.Leap-15.1_appliance.x86_64-1.15.1.kernel PXE_
    ↪SERVER_IP:/srv/tftpboot/boot/linux
   ```

3. Copy the disk image, MD5 file, system kernel, initrd and bootoptions to the PXE boot server:

   Activation of the deployed system is done via `kexec` of the kernel and initrd provided here.

   a) Copy system image and MD5 checksum

   ```
   scp Leap-15.1_appliance.x86_64-1.15.1.xz PXE_SERVER_IP:/srv/
    ↪tftpboot/image/
   scp Leap-15.1_appliance.x86_64-1.15.1.md5 PXE_SERVER_IP:/
    ↪srv/tftpboot/image/
   ```

   b) Copy kernel, initrd and bootoptions used for booting the system via kexec

   ```
   scp Leap-15.1_appliance.x86_64-1.15.1.initrd PXE_SERVER_IP:/
    ↪srv/tftpboot/image/
   scp Leap-15.1_appliance.x86_64-1.15.1.kernel PXE_SERVER_IP:/
    ↪srv/tftpboot/image/
   scp Leap-15.1_appliance.x86_64-1.15.1.config.bootoptions␣
    ↪PXE_SERVER_IP:/srv/tftpboot/image/
   ```

   Note: The config.bootoptions file is used together with kexec to boot the previously dumped image. The information in that file references the root of the dumped image and can also include any other type of boot options. The file provided with the KIWI NG built image is by default connected to the image present in the PXE TAR archive. If other images got deployed the contents of this file must be adapted to match the correct root reference.

4. Add/Update the kernel command line parameters

   Edit your PXE configuration (for example `pxelinux.cfg/default`) on the PXE server and add these parameters to the append line, typically looking like this:

```
append initrd=boot/initrd rd.kiwi.install.pxe rd.kiwi.install.
↪image=tftp://192.168.100.16/image/Leap-15.1_appliance.x86_64-
↪1.15.1.xz
```

The location of the image is specified as a source URI which can point to any location supported by the `curl` command. KIWI NG calls `curl` to fetch the data from this URI. This also means your image, MD5 file, system kernel and initrd could be fetched from any server and doesn't have to be stored on the `PXE_SERVER`.

By default KIWI NG does not use specific `curl` options or flags. However it is possible to add custom ones by adding the `rd.kiwi.install.pxe.curl_options` flag into the kernel command line. `curl` options are passed as comma separated values. Consider the following example:

```
rd.kiwi.install.pxe.curl_options=--retry,3,--retry-delay,3,--
↪speed-limit,2048
```

The above tells KIWI NG to call `curl` with:

```
curl --retry 3 --retry-delay 3 --speed-limit 2048 -f <url>
```

This is specially handy when the deployment infraestructure requires some fine tuned download behavior. For example, setting retries to be more robust over flaky network connections.

Note: KIWI NG just replaces commas with spaces and appends it to the `curl` call. This is relevant since command line options including commas will always fail.

Note: The initrd and Linux Kernel for pxe boot are always loaded via tftp from the `PXE_SERVER`.

4. Create a target disk

   Follow the steps above to create a virtual target disk

5. Connect the client to the network and boot QEMU with the target disk attached to the virtual machine.

```
$ qemu -boot n -hda target_disk -m 4096
```

Note: QEMU bridged networking

In order to let qemu connect to the network we recommend to setup a network bridge on the host system and let qemu connect to it via a custom /etc/qemu-ifup. For details see https://en.wikibooks.org/wiki/QEMU/Networking

10.3.5 OEM Customization

The deployment process of an oem image can be customized through the `oemconfig` element which is a child section of the `type` element like the following example shows:

```
<oemconfig>
  <oem-swapsize>512</oem-swapsize>
</oemconfig>
```

The following list of optional `oem` element settings exists:

oemconfig.oem-resize Element Specify if the disk has the capability to expand itself to a new disk geometry or not. By default, this feature is activated. The implementation of the resize capability is done in a dracut module packaged as `dracut-kiwi-oem-repart`. If `oem-resize` is set to false, the installation of the corresponding dracut package can be skipped as well.

oemconfig.oem-boot-title Element By default, the string OEM will be used as the boot manager menu entry when KIWI creates the GRUB configuration during deployment. The `oem-boot-title` element allows you to set a custom name for the grub menu entry. This value is represented by the `kiwi_oemtitle` variable in the initrd

oemconfig.oem-bootwait Element Specify if the system should wait for user interaction prior to continuing the boot process after the disk image has been dumped to the designated storage device (default value is false). This value is represented by the `kiwi_oembootwait` variable in the initrd

oemconfig.oem-reboot Element Specify if the system is to be rebooted after the disk image has been deployed to the designated storage device (default value is false). This value is represented by the `kiwi_oemreboot` variable in the initrd

oemconfig.oem-reboot-interactive Element Specify if the system is to be rebooted after the disk image has been deployed to the designated storage device (default value is false). Prior to reboot a message is posted and must be acknowledged by the user in order for the system to reboot. This value is represented by the `kiwi_oemrebootinteractive` variable in the initrd

oemconfig.oem-silent-boot Element Specify if the system should boot in silent mode after the disk image has been deployed to the designated storage device (default value is false). This value is represented by the `kiwi_oemsilentboot` variable in the initrd

oemconfig.oem-shutdown Element Specify if the system is to be powered down after the disk image has been deployed to the designated storage device (default value is false). This value is represented by the `kiwi_oemshutdown` variable in the initrd

oemconfig.oem-shutdown-interactive Element Specify if the system is to be powered down after the disk image has been deployed to the designated storage device (default value is false). Prior to shutdown a message is posted and must be acknowledged by the user in order for the system to power off. This value is represented by the `kiwi_oemshutdowninteractive` variable in the initrd

oemconfig.oem-swap Element Specify if a swap partition should be created. By default no swap partition will be created. This value is represented by the `kiwi_oemswap` variable in the initrd

oemconfig.oem-swapname Element Specify the name of the swap space. By default the name is set to `LVSwap`. The default already indicates that this setting is only useful in combination with the LVM volume manager. In this case the swapspace is setup as a volume in the volume group and any volume needs a name. The name set here is used to give the swap volume a name.

oemconfig.oem-swapsize Element Set the size of the swap partition. If a swap partition is to be created and the size of the swap partition is not specified with this optional element, KIWI will calculate the size of the swap partition and create a swap partition equal to two times the RAM installed on the system at initial boot time. This value is represented by the `kiwi_oemswapMB` variable in the initrd

oemconfig.oem-systemsize Element Set the size the operating system is allowed to consume on the target disk. The size limit does not include any consideration for swap space or a recovery partition. In a setup *without* a systemdisk element this value specifies the size of the root partition. In a setup *including* a systemdisk element this value specifies the size of the LVM partition which contains all specified volumes. Thus, the sum of all specified volume sizes plus the sum of the specified freespace for each volume must be smaller or equal to the size specified with the `oem-systemsize` element. This value is represented by the variable `kiwi_oemrootMB` in the initrd

oemconfig.oem-unattended Element The installation of the image to the target system occurs automatically without requiring user interaction. If multiple possible target devices are discovered the image is deployed to the first device. `kiwi_oemunattended` in the initrd

10.3.6 Installation Media Customization

The installation media created for OEM network or CD/DVD deployments can be customized with the `installmedia` section which is a child section of the `type` element as it appears in the following example:

```
<installmedia>
  <initrd action="omit">
    <dracut module="network-legacy"/>
  </initrd>
</installmedia>
```

The `installmedia` is only available for OEM image types that includes the request to create an installation media.

The `initrd` child element of `installmedia` lists dracut modules, they can be omitted, added or staticaly set the list of included ones. This is specified with the `action` attribute and can take `action="omit"`, `action="add"` or `action="set"` values.

10.4 Build a Docker Container Image

Abstract

This page explains how to build a Docker base image. It contains

- basic configuration explanation
- how to build a Docker image
- how to run it with the Docker daemon

KIWI NG is capable of building native Docker images, from scratch and derived ones. KIWI NG Docker images are considered to be native since the KIWI NG tarball image is ready to be loaded to a Docker daemon, including common container configurations.

The Docker configuration metadata is provided to KIWI NG as part of the *XML description file* using the `<containerconfig>` tag. The following configuration metadata can be specified:

`containerconfig` attributes:

- `name`: Specifies the repository name of the Docker image.
- `tag`: Sets the tag of the Docker image.
- `maintainer`: Specifies the author field of the container.
- `user`: Sets the user name or user id (UID) to be used when running `entrypoint` and `subcommand`. Equivalent of the `USER` directive of a Docker file.
- `workingdir`: Sets the working directory to be used when running `cmd` and `entrypoint`. Equivalent of the `WORKDIR` directive of a Docker file.

`containerconfig` child tags:

- `subcommand`: Provides the default execution parameters of the container. Equivalent of the `CMD` directive of a Docker file.
- `labels`: Adds custom metadata to an image using key-value pairs. Equivalent to one or more `LABEL` directives of a Docker file.
- `expose`: Informs at which ports is the container listening at runtime. Equivalent to one or more `EXPOSE` directives of a Docker file.
- `environment`: Sets an environment values using key-value pairs. Equivalent to one or more the `env` directives of a Docker file.
- `entrypoint`: Sets the command that the container will run, it can include parameters. Equivalent of the `ENTRYPOINT` directive of a Docker file.
- `volumes`: Create mountpoints with the given name and mark it to hold external volumes from the host or from other containers. Equivalent to one or more `VOLUME` directives of a Docker file.

Other Docker file directives such as `RUN`, `COPY` or `ADD`, can be mapped to KIWI NG by using the *config.sh* script file to run bash commands or the *overlay tree* to include extra files.

The following example shows how to build a Docker base image based on openSUSE Leap:

1. Make sure you have checked out the example image descriptions, see *Example Appliance Descriptions*.
2. Include the `Virtualization/containers` repository to your list:

   ```
   $ zypper addrepo http://download.opensuse.org/repositories/
   ↪Virtualization:/containers/<DIST> container-tools
   ```

   where the placeholder `<DIST>` is the preferred distribution.
3. Install **umoci** and **skopeo** tools

   ```
   $ zypper in umoci skopeo
   ```

4. Build the image with KIWI NG:

   ```
   $ sudo kiwi-ng --type docker system build \
       --description kiwi-descriptions/suse/x86_64/suse-tumbleweed-
   ↪docker \
       --target-dir /tmp/myimage
   ```

5. Test the Docker image.

   First load the new image

   ```
   $ docker load -i openSUSE-Tumbleweed-container-image.x86_64-1.0.
   ↪4.docker.tar.xz
   ```

   then run the loaded image:

   ```
   $ docker run -it opensuse:42.2 /bin/bash
   ```

10.5 Build a WSL Container Image

Abstract

This page explains how to build a WSL/Appx container image. WSL stands for Windows Subsystem Linux and is a zip based container format consumable by Windows 10 with enabled WSL functionality.

KIWI NG is capable of building WSL images using the **appx** utility. Make sure you have installed a package that provides this command on your build host.

Once the build host has the **appx** installed, the following image type setup is required in the XML description `config.xml`:

```
<type image="appx" metadata_path="/meta/data"/>
```

The `/meta/data` path specifies a path that provides additional information required for the **`WSL-DistroLauncher`**. This component consists out of a Windows(`exe`) executable file and an `AppxManifest.xml` file which references other files like icons and resource configurations for the startup of the container under Windows.

Note: /meta/data

Except for the root filesystem tarball KIWI NG is not responsible for providing the meta data required for the **`WSL-DistroLauncher`**. It is expected that the given metadata path contains all the needed information. Typically this information is delivered in a package provided by the Distribution and installed on the build host

10.5.1 Setup of the WSL-DistroLauncher

The contents of the `AppxManifest.xml` will be changed by KIWI NG if a `containerconfig` section is provided in the XML description. In the context of a WSL image the following container configuration parameters are taken into account:

```
<containerconfig name="my-wsl-container">
    <history
        created_by="Organisation"
        author="Name"
        application_id="AppIdentification"
        package_version="https://docs.microsoft.com/en-us/windows/
↪uwp/publish/package-version-numbering"
        launcher="WSL-DistroLauncher-exe-file"
    >Container Description Text</history>
</containerconfig>
```

All information provided here including the entire section is optional. If not provided the existing `AppxManifest.xml` stays untouched.

created_by Provides the name of a publisher organisation. An appx container needs to be signed off with a digital signature. If the image is build in the Open Build Service (OBS) this happens automatically. Outside of OBS one needs to make sure the given publisher organisation name matches the certificate used for signing.

author Provides the name of the author and maintainer of this container

application_id Provides an ID name for the container. The name must start with a letter and only allows alphanumeric characters. KIWI NG will not validate the given name string because there is no common criteria between the container architectures. KIWI NG just accepts any text.

package_version Provides the version identification for the container. KIWI NG validates this against the Microsoft Package Version Numbering rules.

launcher Provides the binary file name of the launcher `.exe` file.

> **Warning:** There is no validation by KIWI NG if the contents of `AppxManifest.xml` are valid or complete to run the container. Users will find out at call time, not before

The following example shows how to build a WSL image based on openSUSE TW:

1. Make sure you have checked out the example image descriptions, see *Example Appliance Descriptions*.

2. Include the `Virtualization/WSL` repository to your list:

   ```
   $ zypper addrepo http://download.opensuse.org/repositories/
   ↪Virtualization:/WSL/<DIST> WSL
   ```

   where the placeholder `<DIST>` is the preferred distribution.

3. Install **`fb-util-for-appx`** utility and a package that provides the **`WSL-DistroLauncher`** metadata. See the above note about `/meta/data`

   ```
   $ zypper in fb-util-for-appx DISTRO_APPX_METADATA_PACKAGE
   ```

   Note: If you are building in the Open Build Service make sure to add the packages from the zypper call above to your project config via **`osc meta -e prjconf`** and a line of the form `support:  PACKAGE_NAME` for each package that needs to be installed on the Open Build Service worker that runs the KIWI NG build process.

4. Setup the image type:

   Edit the XML description file: `kiwi-descriptions/suse/x86_64/suse-tumbleweed/config.xml` and add the following type and containerconfig:

   ```
   <type image="appx" metadata_path="/meta/data">
       <containerconfig name="Tumbleweed">
           <history
               created_by="SUSE"
               author="KIWI-Team"
               application_id="tumbleweed"
               package_version="2003.12.0.0"
               launcher="openSUSE-Tumbleweed.exe"
           >Tumbleweed Appliance text based</history>
       </containerconfig>
   </type>
   ```

   > **Warning:** If the configured metadata path does not exist the build will fail. Furthermore there is no validation by KIWI NG that the contents of the metadata path are valid or complete with respect to the requirements of the **`WSL-DistroLauncher`**

5. Build the image with KIWI NG:

```
$ sudo kiwi-ng --type appx system build \
    --description kiwi-descriptions/suse/x86_64/suse-tumbleweed␣
↪\
    --target-dir /tmp/myimage
```

10.5.2 Testing the WSL image

For testing the image a Windows 10 system is required. As a first step the optional feature named `Microsoft-Windows-Subsystem-Linux` must be enabled. For further details on how to setup the Windows machine see the following documentation: Windows Subsystem for Linux

10.6 Build KIS Image (Kernel, Initrd, System)

Abstract

This page explains how to build an image that consists out of three components. The kernel an initrd and an optional root filesystem image. In KIWI NG terminology this is called KIS.

A KIS image is a collection of image components that are not associated with a dedicated use case. This means from a KIWI NG perspective we don't know in which environment these components are later used. The predecessor of this image type was called `pxe` under the assumption that the components will be used in a pxe boot environment. However this assumption is not neccessarily true and the image components are used in a different way. Because there are so many possible deployment strategies for a `kernel` plus `initrd` and optional `system root filesystem`, KIWI NG provides this as generic `KIS` type that is generically usable.

The former `pxe` image type will continue to exist but is expected to be used only in combination with the legacy `netboot` infrastructure as described in *Build PXE Root File System Image for the legacy netboot infrastructure*.

To add a KIS build to your appliance, create a `type` element with `image` set to `kis` in your `config.xml` as shown below:

```
<preferences>
    <type image="kis"/>
</preferences>
```

With this image type setup KIWI NG will just build a kernel and initrd not associated to any system root file system. Usually such an image is only useful with some custom dracut extensions as part of the image description.

The following attributes of the `type` element are often used when building KIS images:

- filesystem: Specifies the root filesystem and triggers the build of an additional filesystem image of that filesystem. The generated kernel command line options file (append file) will then also include a root= parameter that references this filesystem image UUID. If the information from the append file should be used or not is optional.
- kernelcmdline: Specifies kernel command line options that will be part of the generated kernel command line options file (append file). By default the append file contains no information or the reference to the root UUID if the filesystem attribute is used.

All other attributes of the type element that applies to an optional root filesystem image will be effective in the system image of a KIS image as well.

With the appropriate settings present in config.xml KIWI NG can now build the image:

```
$ sudo kiwi-ng --profile KIS system build \
    --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
    --target-dir /tmp/myimage
```

The resulting image components are saved in the folder /tmp/myimage. Outside of a deployment infrastructure the example KIS image can be tested with QEMU as follows:

```
$ sudo qemu
    -kernel /tmp/myimage/*.kernel \
    -initrd /tmp/myimage/*.initrd.xz \
    -append $(cat /tmp/myimage/*.append) \
    -hda /tmp/myimage/Leap-15.1_appliance.*-1.15.1 \
```

Note: For testing the components of a KIS image a deployment infrastructure and also a deployment process is usually needed. One example of a deployment infrastructure using PXE is provided by KIWI NG with the netboot infrastructure. However that netboot infrastructure is no longer developed and only kept for compatibiliy reasons. For details see *Build PXE Root File System Image for the legacy netboot infrastructure*

CHAPTER

ELEVEN

WORKING WITH IMAGES

Note: This document provides a collection of worksheets which describes the creation and setup of appliances to work within a number of different target environments.

11.1 Deploy ISO Image on an USB Stick

Abstract

This page provides further information for handling ISO images built with KIWI NG and references the following articles:

- *Build an ISO Hybrid Live Image*

In KIWI NG all generated ISO images are created to be hybrid. This means, the image can be used as a CD/DVD or as a disk. This works because the ISO image also has a partition table embedded. With more and more computers delivered without a CD/DVD drive this becomes important.

The very same ISO image can be copied onto a USB stick and used as a bootable disk. The following procedure shows how to do this:

1. Plug in a USB stick

   Once plugged in, check which Unix device name the stick was assigned to. The following command provides an overview about all linux storage devices:

   ```
   $ lsblk
   ```

2. Dump the ISO image on the USB stick:

   Warning: Make sure the selected device really points to your stick because the following operation can not be revoked and will destroy all data on the selected device

```
$ dd if=Leap-15.1_appliance.x86_64-1.15.1.iso of=/dev/
↪<stickdevice>
```

3. Boot from your USB Stick

   Activate booting from USB in your BIOS/UEFI. As many firmware has different procedures on how to do it, look into your user manual. Many firmware offers a boot menu which can be activated at boot time.

11.2 Deploy ISO Image as File on a FAT32 Formated USB Stick

Abstract

This page provides further information for handling ISO images built with KIWI NG and references the following articles:

- *Build an ISO Hybrid Live Image*

In KIWI NG, all generated ISO images are created to be hybrid. This means, the image can be used as a CD/DVD or as a disk. The deployment of such an image onto a disk like an USB stick normally destroys all existing data on this device. Most USB sticks are pre-formatted with a FAT32 Windows File System and to keep the existing data on the stick untouched a different deployment needs to be used.

The following deployment process copies the ISO image as an additional file to the USB stick and makes the USB stick bootable. The ability to boot from the stick is configured through a SYSLINUX feature which allows to loopback mount an ISO file and boot the kernel and initrd directly from the ISO file.

The initrd loaded in this process must also be able to loopback mount the ISO file to access the root filesystem and boot the live system. The dracut initrd system used by KIWI NG provides this feature upstream called as “iso-scan”. Therefore all KIWI NG generated live ISO images supports this deployment mode.

For copying the ISO file on the USB stick and the setup of the SYSLINUX bootloader to make use of the “iso-scan” feature an extra tool named `live-grub-stick` exists. The following procedure shows how to setup the USB stick with `live-grub-stick`:

1. Install the `live-grub-stick` package from software.opensuse.org:
2. Plug in a USB stick

   Once plugged in, check which Unix device name the FAT32 partition was assigned to. The following command provides an overview about all storage devices and their filesystems:

```
$ sudo lsblk --fs
```

3. Call the `live-grub-stick` command as follows:

   Assuming "/dev/sdz1" was the FAT32 partition selected from the output of the `lsblk` command:

   ```
   $ sudo live-grub-stick Leap-15.1_appliance.x86_64-1.15.1.iso /
   ↪dev/sdz1
   ```

4. Boot from your USB Stick

   Activate booting from USB in your BIOS/UEFI. As many firmware has different procedures on how to do it, look into your user manual. EFI booting from iso image is not supported at the moment, for EFI booting use –isohybrid option with live-grub-stick, however note that all the data on the stick will be lost. Many firmware offers a boot menu which can be activated at boot time. Usually this can be reached by pressing the `Esc` or `F12` keys.

11.3 Image Description for Amazon EC2

Abstract

This page provides further information for handling Amazon EC2 images built with KIWI NG and references the following articles:

- *Build a Virtual Disk Image*

A virtual disk image which is able to boot in the Amazon EC2 cloud framework has to comply the following constraints:

- Xen tools and libraries must be installed
- cloud-init package must be installed
- cloud-init configuration for Amazon must be provided
- Grub bootloader modules for Xen must be installed
- AWS tools must be installed
- Disk size must be set to 10G
- Kernel parameters must allow for xen console

To meet this requirements add or update the KIWI NG image description as follows:

1. Software packages

   Make sure to add the following packages to the package list

Note: Package names used in the following list matches the package names of the SUSE distribution and might be different on other distributions.

```
<package name="aws-cli"/>
<package name="grub2-x86_64-xen"/>
<package name="xen-libs"/>
<package name="xen-tools-domU"/>
<package name="cloud-init"/>
```

2. Image Type definition

   Update the oem image type setup as follows

```
<type image="oem"
      filesystem="ext4"
      kernelcmdline="console=xvc0 multipath=off net.ifnames=0"
      devicepersistency="by-label"
      firmware="ec2">
  <bootloader name="grub2" timeout="1"/>
  <size unit="M">10240</size>
  <machine xen_loader="hvmloader"/>
  <oemconfig>
      <oem-resize>false</oem-resize>
  </oemconfig>
</type>
```

3. Cloud Init setup

   Cloud init is a service which runs at boot time and allows to customize the system by activating one ore more cloud init modules. For Amazon EC2 the following configuration file `/etc/cloud/cloud.cfg` needs to be provided as part of the overlay files in your KIWI NG image description

```
users:
  - default

disable_root: true
preserve_hostname: false
syslog_fix_perms: root:root

datasource_list: [ NoCloud, Ec2, None ]

cloud_init_modules:
  - migrator
  - bootcmd
  - write-files
  - growpart
  - resizefs
  - set_hostname
```

(continues on next page)

(continued from previous page)

```
  - update_hostname
  - update_etc_hosts
  - ca-certs
  - rsyslog
  - users-groups
  - ssh

cloud_config_modules:
  - mounts
  - ssh-import-id
  - locale
  - set-passwords
  - package-update-upgrade-install
  - timezone

cloud_final_modules:
  - scripts-per-once
  - scripts-per-boot
  - scripts-per-instance
  - scripts-user
  - ssh-authkey-fingerprints
  - keys-to-console
  - phone-home
  - final-message
  - power-state-change

system_info:
  default_user:
    name: ec2-user
    gecos: "cloud-init created default user"
    lock_passwd: True
    sudo: ["ALL=(ALL) NOPASSWD:ALL"]
    shell: /bin/bash
  paths:
    cloud_dir: /var/lib/cloud/
    templates_dir: /etc/cloud/templates/
  ssh_svcname: sshd
```

An image built with the above setup can be uploaded into the Amazon EC2 cloud and registered as image. For further information on how to upload to EC2 see: ec2uploadimg

11.4 Image Description for Microsoft Azure

Abstract

This page provides further information for handling Azure disk images built with KIWI NG and references the following articles:

- *Build a Virtual Disk Image*

A virtual disk image which is able to boot in the Microsoft Azure cloud framework has to comply the following constraints:

- Hyper-V tools must be installed
- Microsoft Azure Agent must be installed
- Disk size must be set to 30G
- Kernel parameters must allow for serial console

To meet this requirements update the KIWI NG image description as follows:

1. Software packages

   Make sure to add the following packages to the package list

   Note: Package names used in the following list matches the package names of the SUSE distribution and might be different on other distributions.

   ```
   <package name="hyper-v"/>
   <package name="python-azure-agent"/>
   ```

2. Image Type definition

   Update the oem image type setup as follows

   ```
   <type image="oem"
         filesystem="ext4"
         kernelcmdline="console=ttyS0 rootdelay=300 net.ifnames=0"
         devicepersistency="by-uuid"
         format="vhd-fixed"
         formatoptions="force_size"
         bootpartition="true"
         bootpartsize="1024">
     <bootloader name="grub2" timeout="1"/>
     <size unit="M">30720</size>
     <oemconfig>
         <oem-resize>false</oem-resize>
     </oemconfig>
   </type>
   ```

An image built with the above setup can be uploaded into the Microsoft Azure cloud and registered as image. For further information on how to upload to Azure see: azurectl

11.5 Image Description for Google Compute Engine

Abstract

This page provides further information for handling GCE images built with KIWI NG and references the following articles:

- *Build a Virtual Disk Image*

A virtual disk image which is able to boot in the Google Compute Engine cloud framework has to comply the following constraints:

- KIWI NG type must be an expandable disk
- Google Compute Engine init must be installed
- Disk size must be set to 10G
- Kernel parameters must allow for serial console

To meet this requirements update the KIWI NG image description as follows:

1. Software packages

   Make sure to add the following packages to the package list

   Note: Package names used in the following list matches the package names of the SUSE distribution and might be different on other distributions.

   ```
   <package name="google-compute-engine-init"/>
   ```

2. Image Type definition

   To allow the image to be expanded to the configured disk geometry of the instance started by Google Compute Engine it is suggested to let KIWI NG's OEM boot code take over that task. It would also be possible to try cloud-init's resize module but we found conflicts when two cloud init systems, `google-compute-engine-init` and `cloud-init` were used together. Thus for now we stick with KIWI NG's boot code which can resize the disk from within the initrd before the system gets activated through systemd.

   Update the oem image type setup to be changed into an expandable type as follows:

   ```
   <type image="oem"
         initrd_system="dracut"
         filesystem="ext4"
         kernelcmdline="console=ttyS0,38400n8 net.ifnames=0"
   ```

(continues on next page)

(continued from previous page)

```
      format="gce">
  <bootloader name="grub2" timeout="1"/>
  <size unit="M">10240</size>
  <oemconfig>
      <oem-resize>true</oem-resize>
      <oem-swap>false</oem-swap>
  </oemconfig>
</type>
```

An image built with the above setup can be uploaded into the Google Compute Engine cloud and registered as image. For further information on how to upload to Google see: `google-cloud-sdk` on software.opensuse.org

11.6 Image Description for Vagrant

Abstract

This page provides further information for handling Vagrant controlled disk images built with KIWI NG and references the following article:

- *Build a Virtual Disk Image*

Vagrant is a framework to implement consistent processing/testing work environments based on Virtualization technologies. To run a system, Vagrant needs so-called **boxes**. A box is a TAR archive containing a virtual disk image and some metadata.

To build Vagrant boxes, you can use Packer which is provided by Hashicorp itself. Packer is based on the official installation media (DVDs) as shipped by the distribution vendor.

The KIWI NG way of building images might be helpful, if such a media does not exist or does not suit your needs. For example, if the distribution is still under development or you want to use a collection of your own repositories. Note, that in contrast to Packer KIWI NG only supports the libvirt and VirtualBox providers. Other providers require a different box layout that is currently not supported by KIWI NG.

In addition, you can use the KIWI NG image description as source for the Open Build Service which allows building and maintaining boxes.

Vagrant expects boxes to be setup in a specific way (for details refer to the Vagrant box documentation.), applied to the referenced KIWI NG image description from *Build a Virtual Disk Image*, the following steps are required:

1. Update the image type setup

```
<type image="oem" filesystem="ext4" format="vagrant">
    <bootloader name="grub2" timeout="0"/>
    <vagrantconfig provider="libvirt" virtualsize="42"/>
```

(continues on next page)

(continued from previous page)

```
    <size unit="G">42</size>
    <oemconfig>
        <oem-resize>false</oem-resize>
    </oemconfig>
</type>
```

This modifies the type to build a Vagrant box for the libvirt provider including a pre-defined disk size. The disk size is optional, but recommended to provide some free space on disk.

For the VirtualBox provider, the additional attribute `virtualbox_guest_additions_present` can be set to `true` when the VirtualBox guest additions are installed in the KIWI NG image:

```
<type image="oem" filesystem="ext4" format="vagrant">
    <bootloader name="grub2" timeout="0"/>
    <vagrantconfig
      provider="virtualbox"
      virtualbox_guest_additions_present="true"
      virtualsize="42"
    />
    <size unit="G">42</size>
    <oemconfig>
        <oem-resize>false</oem-resize>
    </oemconfig>
</type>
```

The resulting Vagrant box then uses the `vboxfs` module for the synchronized folder instead of `rsync`, that is used by default.

2. Add mandatory packages

```
<package name="sudo"/>
<package name="openssh"/>
```

3. Add additional packages

If you have set the attribute `virtualbox_guest_additions_present` to `true`, add the VirtualBox guest additions. For openSUSE the following packages are required:

```
<package name="virtualbox-guest-tools"/>
<package name="virtualbox-guest-x11"/>
<package name="virtualbox-guest-kmp-default"/>
```

Otherwise, you must add `rsync`:

```
<package name="rsync"/>
```

Note that KIWI NG cannot verify whether these packages are installed. If they are missing, the resulting Vagrant box will be broken.

4. Add Vagrant user

   ```
   <users group='vagrant'>
       <user name='vagrant' password='vh4vw1N4alxKQ' home='/home/
   ↪vagrant'/>
   </users>
   ```

   This adds the **vagrant** user to the system and applies the name of the user as the password for login.

5. Configure SSH, the default shared folder and sudo permissions

   Vagrant expects that it can login as the user `vagrant` using an insecure public key[1]. Furthermore, vagrant also usually uses `/vagrant` as the default shared folder and assumes that the `vagrant` user can invoke commands via **`sudo`** without having to enter a password.

   This can be achieved using the function `baseVagrantSetup` in `config.sh`:

   ```
   baseVagrantSetup
   ```

6. Additional customizations:

   Additionally to `baseVagrantSetup`, you might want to also ensure the following:

   - If you have installed the Virtualbox guest additions into your box, then also load the `vboxsf` kernel module.
   - When building boxes for libvirt, then ensure that the default wired networking interface is called `eth0` and uses DHCP. This is necessary since libvirt uses `dnsmasq` to issue IPs to the VMs. This step can be omitted for Virtualbox boxes.

An image built with the above setup creates a Vagrant box file with the extension `.vagrant.libvirt.box` or `.vagrant.virtualbox.box`. Add the box file to Vagrant with the command:

```
vagrant box add my-box image-file.vagrant.libvirt.box
```

Note: Using the box with the libvirt provider requires alongside a correct Vagrant installation:

- the plugin `vagrant-libvirt` to be installed
- a running libvirtd daemon

Once added to Vagrant, boot the box and log in with the following sequence of **`vagrant`** commands:

```
vagrant init my-box
vagrant up --provider libvirt
vagrant ssh
```

[1] The insecure key is removed from the box when the it is first booted via Vagrant.

11.6.1 Customizing the embedded Vagrantfile

Warning: This is an advanced topic and not required for most users

Vagrant ship with an embedded `Vagrantfile` that carries settings specific to this box, for instance the synchronization mechanism for the shared folder. KIWI NG generates such a file automatically for you and it should be sufficient for most use cases.

If a box requires different settings in the embedded `Vagrantfile`, then the user can provide KIWI NG with a path to an alternative via the attribute `embebbed_vagrantfile` of the `vagrantconfig` element: it specifies a relative path to the `Vagrantfile` that will be included in the finished box.

In the following example snippet from `config.xml` we add a custom `MyVagrantfile` into the box (the file should be in the image description directory next to `config.sh`):

```xml
<type image="oem" filesystem="ext4" format="vagrant">
    <bootloader name="grub2" timeout="0"/>
    <vagrantconfig
      provider="libvirt"
      virtualsize="42"
      embedded_vagrantfile="MyVagrantfile"
    />
    <size unit="G">42</size>
    <oemconfig>
        <oem-resize>false</oem-resize>
    </oemconfig>
</type>
```

The option to provide a custom `Vagrantfile` can be combined with the usage of *profiles* (see *Image Profiles*), so that certain builds can use the automatically generated `Vagrantfile` (in the following example that is the Virtualbox build) and others get a customized one (the libvirt profile in the following example):

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

<image schemaversion="7.2" name="Leap-15.1_appliance">
  <!-- description goes here -->
  <profiles>
    <profile name="libvirt" description="Vagrant Box for Libvirt"/>
    <profile name="virtualbox" description="Vagrant Box for␣
↪VirtualBox"/>
  </profiles>

  <!-- general preferences go here -->

  <preferences profiles="libvirt">
    <type
```

(continues on next page)

(continued from previous page)

```
      image="oem"
      filesystem="ext4"
      format="vagrant">
        <bootloader name="grub2" timeout="0"/>
        <vagrantconfig
          provider="libvirt"
          virtualsize="42"
          embedded_vagrantfile="LibvirtVagrantfile"
        />
        <size unit="G">42</size>
        <oemconfig>
            <oem-resize>false</oem-resize>
        </oemconfig>
  </type>
  </preferences>
  <preferences profiles="virtualbox">
    <type
      image="oem"
      filesystem="ext4"
      format="vagrant">
        <bootloader name="grub2" timeout="0"/>
        <vagrantconfig
          provider="virtualbox"
          virtualbox_guest_additions_present="true"
          virtualsize="42"
        />
        <size unit="G">42</size>
        <oemconfig>
            <oem-resize>false</oem-resize>
        </oemconfig>
    </type>
  </preferences>

  <!-- remaining box description -->
</image>
```

11.7 Image Description Encrypted Disk

Abstract

This page provides further information for handling disk images with an encrypted root filesystem setup. The information here is based on top of the following article:

- *Build a Virtual Disk Image*

A virtual disk image can be partially or fully encrypted using the LUKS extension supported by KIWI NG. A fully encrypted image also includes the data in `/boot` to be encrypted. Such an image requests the passphrase for the master key to be entered at the bootloader stage. A partialy encrypted image keeps `/boot` unencrypted and on an extra boot partition. Such an image requests the passphrase for the master key later in the boot process when the root partition gets accessed by the systemd mount service. In any case the master passphrase is requested only once.

Update the KIWI NG image description as follows:

1. Software packages

   Make sure to add the following package to the package list

   Note: Package names used in the following list match the package names of the SUSE distribution and might be different on other distributions.

   ```
   <package name="cryptsetup"/>
   ```

2. Image Type definition

   Update the oem image type setup as follows

   Full disk encryption including `/boot`:

   ```
   <type image="oem" filesystem="ext4" luks="linux"␣
   ↪bootpartition="false">
       <oemconfig>
           <oem-resize>false</oem-resize>
       </oemconfig>
   </type>
   ```

   Encrypted root partition with an unencrypted extra `/boot` partition:

   ```
   <type image="oem" filesystem="ext4" luks="linux"␣
   ↪bootpartition="true">
       <oemconfig>
           <oem-resize>false</oem-resize>
       </oemconfig>
   </type>
   ```

   Note: The value for the `luks` attribute sets the master passphrase for the LUKS keyring. Therefore the XML description becomes security critical and should only be readable by trustworthy people

11.8 Deploy and Run System in a RamDisk

Abstract

This page provides further information for handling oem images built with KIWI NG and references the following articles:

- *Build an Expandable Disk Image*

If a machine should run the OS completely in memory without the need for any persistent storage, the approach to deploy the image into a ramdisk serves this purpose. KIWI NG allows to create a bootable ISO image which deploys the image into a ramdisk and activates that image with the following oem type definition:

```
<type image="oem" filesystem="ext4" installiso="true" initrd_system=
↪"dracut" installboot="install" kernelcmdline="rd.kiwi.ramdisk␣
↪ramdisk_size=2048000">
    <bootloader name="grub2" timeout="1"/>
    <oemconfig>
        <oem-skip-verify>true</oem-skip-verify>
        <oem-unattended>true</oem-unattended>
        <oem-unattended-id>/dev/ram1</oem-unattended-id>
        <oem-swap>false</oem-swap>
        <oem-multipath-scan>false</oem-multipath-scan>
     </oemconfig>
 </type>
```

The type specification above builds an installation ISO image which deploys the System Image into the specified ramdisk device (/dev/ram1). The setup of the ISO image boots with a short boot timeout of 1sec and just runs through the process without asking any questions. In a ramdisk deployment the optional target verification, swap space and multipath targets are out of scope and therefore disabled.

The configured size of the ramdisk specifies the size of the OS disk and must be at least of the size of the System Image. The disk size can be configured with the following value in the kernelcmdline attribute:

- ramdisk_size=kbyte-value"

An image built with the above setup can be tested in QEMU as follows:

```
$ qemu -cdrom Leap-15.1_appliance.x86_64-1.15.1.install.iso
```

Note: Enough Main Memory

The machine, no matter if it's a virtual machine like QEMU or a real machine, must provide enough RAM to hold the image in the ramdisk as well as have enough RAM available to operate the OS and its applications. The KIWI NG build image with the extension .raw provides the System Image which gets deployed into the RAM space. Substract the size of the System

Image from the RAM space the machine offers and make sure the result is still big enough for the use case of the appliance. In case of a virtual machine, attach enough main memory to fit this calculation. In case of QEMU this can be done with the `-m` option

Like all other oem KIWI NG images, also the ramdisk setup supports all the deployments methods as explained in *Deployment Methods* This means it's also possible to dump the ISO image on a USB stick let the system boot from it and unplug the stick from the machine because the system was deployed into RAM

Note: Limitations Of RamDisk Deployments

Only standard images which can be booted by a simple root mount and root switch can be used. Usually KIWI NG calls kexec after deployment such that the correct, for the image created dracut initrd, will boot the image. In case of a RAM only system kexec does not work because it would loose the ramdisk contents. Thus the dracut initrd driving the deployment is also the environment to boot the system. There are cases where this environment is not suitable to boot the system.

11.9 Custom Disk Partitions

Abstract

This page provides some details about what KIWI NG supports and does not support regarding customization over the partition scheme. It also provides some guidance in case the user requires some custom layout beyond KIWI NG supported features.

By design, KIWI NG does not support a customized partition table. Alternatively, KIWI NG supports the definition of user-defined volumes which covers most of common use cases. See *Custom Disk Volumes* for further details about that.

KIWI NG has its own partitioning schema which is defined according to several different user configurations: boot firmware, boot partition, expandable layouts, etc. Those supported features have an impact on the partitioning schema. MBR or GUID partition tables are not flexible, carry limitations and are tied to some specific disk geometry. Because of that the preferred alternative to disk layouts based on traditional partition tables is using flexible approaches like logic volumes.

As an example, expandable OEM images is a relevant KIWI NG feature that is incompatible with the idea of adding user defined partitions on the system area.

Despite no full customization is supported, some aspects of the partition schema can be customized. KIWI NG supports:

1. Adding a spare partition *before* the root (`/`) partition.

   It can be achieved by using the `spare_part` type attribute.

2. Leaving some unpartitioned area at the *end* of the disk.

   Setting some unpartitioned free space on the disk can be done using the `unpartitioned` attribute of `size` element in type's section. [LINK]

3. Expand built disks to a new size adding unpartitioned free space at the *end* of the disk.

   A built image can be resized by using the `kiwi-ng image resize` command and set a new extended size for the disk. See KIWI NG commands docs *here*.

11.9.1 Custom Partitioning at Boot Time

Adding additional partitions at boot time of KIWI NG images is also possible, however, setting the tools and scripts for doing so needs to be handled by the user. A possible strategy to add partitions on system area are described below.

The main idea consists on running a first boot service that creates the partitions that are needed. Adding custom services is simple, use the following steps:

1. Create a unit file for a systemd service:

   ```
   [Unit]
   Description=Add a data partition
   After=basic.target
   Wants=basic.target

   [Service]
   Type=oneshot
   ExecStart=/bin/bash /usr/local/bin/create_part
   ```

   This systemd unit file will run at boot time once systemd reaches the basic target. At this stage all basic services are up an running (devices mounted, network interfaces up, etc.). In case the service is required to run on earlier stages for some reason, default dependencies need to be disabled, see systemd man pages.

2. Create partitioner shell script matching your specific needs

   Consider the following steps for a partitioner shell script that creates a new partition. Following the above unit file example the `/usr/local/bin/create_part` script should cover the following steps:

   a. Verify partition exists

      Verify the required partition is not mounted neither exists. Exit zero (0) if is already there.

      Use tools such `findmnt` to find the root device and `blkid` or `lsblk` to find a partition with certain label or similar criteria.

   b. Create a new partition

      Create a new partition. On error, exit with non zero.

Use partitioner tools such as `sgdisk` that can be easily used in non interactive scripts. Using `partprobe` to reload partition table to make OS aware of the changes is handy.

c. Make filesystem

   Add the desired filesystem to the new partitions. On error, exit with non zero.

   Regular filesystem formatting tools (`mkfs.ext4` just to mention one) can be used to apply the desired filesystem to the just created new partition. At this stage it is handy to add a label to the filesystem for easy recognition on later stages or script reruns.

d. Update fstab file

   Just echo and append the desired entry in /etc/fstab.

e. Mount partition

   `mount --all` will try to mount all fstab volumes, it just omits any already mounted device.

3. Add additional files into the root overlay tree.

   The above described unit files and partition creation shell script need to be included into the overlay tree of the image, thus they should be placed into the expected paths in root folder (or in `root.tar.gz` tarball).

4. Activate the service in `config.sh`

   The service needs to be enabled during image built time to be run during the very first boot. In can be done by adding the following snipped inside the `config.sh`.

11.10 Custom Disk Volumes

Abstract

This chapter provides high level explanations on how to handle volumes or subvolumes definitions for disk images using KIWI NG.

KIWI NG supports defining custom volumes by using the logical volume manager (LVM) for the Linux kernel or by setting volumes at filesystem level when filesystem supports it (e.g. btrfs).

Volumes are defined in the KIWI NG description file `config.xml`, using `systemdisk`. This element is a child of the `type`. Volumes themselves are added via (multiple) `volume` child elements of the `systemdisk` element:

```
<image schemaversion="7.2" name="openSUSE-Leap-15.1">
  <type image="oem" filesystem="btrfs" preferlvm="true">
    <systemdisk name="vgroup">
      <volume name="usr/lib" size="1G" label="library"/>
      <volume name="@root" freespace="500M"/>
      <volume name="etc_volume" mountpoint="etc" copy_on_write=
↪"false"/>
      <volume name="bin_volume" size="all" mountpoint="/usr/bin"/>
    </systemdisk>
  </type>
</image>
```

Additional non-root volumes are created for each `volume` element. Volume details can be defined by setting the following `volume` attributes:

- `name`: Required attribute representing the volume's name. Additionally, this attribute is interpreted as the mountpoint if the `mountpoint` attribute is not used.
- `mountpoint`: Optional attribute that specifies the mountpoint of this volume.
- `size`: Optional attribute to set the size of the volume. If no suffix (`M` or `G`) is used, then the value is considered to be in megabytes.

  Note: Special name for the root volume

  One can use the `@root` name to refer to the volume mounted at `/`, in case some specific size attributes for the root volume have to be defined. For instance:

  ```
  <volume name="@root" size="4G"/>
  ```

  In addition to the custom size of the root volume it's also possible to setup the name of the root volume as follows:

  ```
  <volume name="@root=rootlv" size="4G"/>
  ```

  If no name for the root volume is specified the default name: **LVRoot** applies.

- `freespace`: Optional attribute defining the additional free space added to the volume. If no suffix (`M` or `G`) is used, the value is considered to be in megabytes.
- `label`: Optional attribute to set filesystem label of the volume.
- `copy_on_write`: Optional attribute to set the filesystem copy-on-write attribute for this volume.

Warning: The size attributes for filesystem volumes, as for btrfs, are ignored and have no effect.

The `systemdisk` element additionally supports the following optional attributes:

- name: The volume group name, by default kiwiVG is used. This setting is only relevant for LVM volumes.
- preferlvm: Boolean value instructing KIWI NG to prefer LVM even if the used filesystem has its own volume management system.

11.11 Setting Up a Network Boot Server

Abstract

This page provides general information how to setup a network boot server that provides all services needed for the PXE boot protocol

To be able to deploy a system through the PXE boot protocol, you need to set up a network boot server providing the services DHCP and tftp. With dnsmasq an utility exists which allows to setup all needed services at once:

11.11.1 Installing and Configuring DHCP and TFTP with dnsmasq

The following instructions can only serve as an example. Depending on your network structure, the IP addresses, ranges and domain settings needs to be adapted to allow the DHCP server to work within your network. If you already have a DHCP server running in your network, make sure that the filename and next-server directives are correctly set on this server.

The following steps describe how to set up dnsmasq to work as DHCP and TFTP server.

1. Install the dnsmasq package.
2. Create the file /etc/dnsmasq.conf and insert the following content

```
# Don't function as a DNS server.
port=0

# Log information about DHCP transactions.
log-dhcp

# Set the root directory for files available via FTP,
# usually "/srv/tftpboot":
tftp-root=TFTP_ROOT_DIR

enable-tftp

dhcp-range=BOOT_SERVER_IP,proxy
```

In the next step it's required to decide for the boot method. There is the PXE loader provided via pxelinux.0 from the syslinux package and there is the GRUB loader provided via the grub package.

Note: Placeholders

Replace all placeholders (written in uppercase) with data fitting your network setup.

2.1. insert the following content to use pxelinux.0:

```
# The boot filename, Server name, Server Ip Address
dhcp-boot=pxelinux.0,,BOOT_SERVER_IP

# Disable re-use of the DHCP servername and filename fields␣
↪as extra
# option space. That's to avoid confusing some old or broken
# DHCP clients.
dhcp-no-override

# PXE menu.  The first part is the text displayed to the␣
↪user.
# The second is the timeout, in seconds.
pxe-prompt="Booting FOG Client", 1

# The known types are x86PC, PC98, IA64_EFI, Alpha, Arc_x86,
# Intel_Lean_Client, IA32_EFI, BC_EFI, Xscale_EFI and X86-
↪64_EFI
# This option is first and will be the default if there is␣
↪no input
# from the user.
pxe-service=X86PC, "Boot to FOG", pxelinux.0
pxe-service=X86-64_EFI, "Boot to FOG UEFI", ipxe
pxe-service=BC_EFI, "Boot to FOG UEFI PXE-BC", ipxe
```

Note: On boot of a network client with that configuration the default pxelinux.0 config file is expected at `TFTP_ROOT_DIR/pxelinux.cfg/default`

2.2. insert the following content to use grub:

```
# The boot filename, Server name, Server Ip Address
dhcp-boot=boot/grub2/i386-pc/core.0,,BOOT_SERVER_IP
```

When using grub the referenced dhcp-boot grub module must be genereated. To do this change the directory to `TFTP_ROOT_DIR` and create the `setvars.conf` with the following content:

```
set root=(tftp)
set net_default_server=BOOT_SERVER_IP
set prefix=boot/grub2
```

Now call the following commands to create the grub module

```
$ grub2-mknetdir --net-directory=TFTP_ROOT_DIR --
↪subdir=boot/grub2
$ grub2-mkimage -O i386-pc-pxe \
    --output boot/grub2/i386-pc/core.0 \
    --prefix=/boot/grub2 \
    -c setvars.conf \
  pxe tftp
```

Note: On boot of a network client with that configuration the grub config file is expected at `TFTP_ROOT_DIR/boot/grub2/grub.cfg`

3. Run the dnsmasq server by calling:

```
systemctl start dnsmasq
```

11.12 Build PXE Root File System Image for the legacy netboot infrastructure

Abstract

This page explains how to build a file system image for use with KIWI NG's PXE boot infrastructure. It contains:

- how to build a PXE file system image
- how to setup the PXE file system image on the PXE server
- how to run it with QEMU

PXE is a network boot protocol that is shipped with most BIOS implementations. The protocol sends a DHCP request to get an IP address. When an IP address is assigned, it uses the TFTP protocol to download a Kernel and boot instructions. Contrary to other images built with KIWI NG, a PXE image consists of separate boot, kernel and root filesystem images, since those images need to be made available in different locations on the PXE boot server.

A root filesystem image which can be deployed via KIWI NG's PXE netboot infrastructure represents the system rootfs in a linux filesystem. A user could loop mount the image and access the contents of the root filesystem. The image does not contain any information about the system disk its partitions or the bootloader setup. All of these information is provided by a client configuration file on the PXE server which controlls how the root filesystem image should be deployed.

Many different deployment strategies are possible, e.g root over NBD (network block device), AoE (ATA over Ethernet), or NFS for diskless and diskfull clients. This particular example shows how to build an overlayfs-based union system based on openSUSE Leap for a diskless client which receives the squashfs compressed root file system image in a ramdisk overlayed via overlayfs and writes new data into another ramdisk on the same system. As diskless client, a QEMU virtual machine is used.

Things to know before

- To use the image, all image parts need to be copied to the PXE boot server. If you have not set up such a server, refer to *Setting Up a Network Boot Server* for instructions.
- The following example assumes you will create the PXE image on the PXE boot server itself (if not, use **scp** to copy the files on the remote host).
- To let QEMU connect to the network, we recommend to setup a network bridge on the host system and let QEMU connect to it via a custom `/etc/qemu-ifup`. For details, see https://en.wikibooks.org/wiki/QEMU/Networking
- The PXE root filesystem image approach is considered to be a legacy setup. The required netboot initrd code will be maintained outside of the KIWI NG appliance builder code base. If possible, we recommend to switch to the OEM disk image deployment via PXE.

1. Make sure you have checked out the example image descriptions, see *Example Appliance Descriptions*.
2. Build the image with KIWI NG:

   ```
   $ sudo kiwi-ng --type pxe system build \
       --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
       --target-dir /tmp/mypxe-result
   ```

3. Change into the build directory:

   ```
   $ cd /tmp/mypxe-result
   ```

4. Copy the initrd and the kernel to `/srv/tftpboot/boot`:

   ```
   $ cp *.initrd.xz /srv/tftpboot/boot/initrd
   $ cp *.kernel /srv/tftpboot/boot/linux
   ```

5. Copy the system image and its MD5 sum to `/srv/tftpboot/image`:

   ```
   $ cp Leap-15.1_appliance.x86_64-1.15.1 /srv/tftpboot/image
   $ cp Leap-15.1_appliance.x86_64-1.15.1.md5 /srv/tftpboot/image
   ```

6. Adjust the PXE configuration file. The configuration file controls which kernel and initrd is loaded and which kernel parameters are set. A template has been installed at `/srv/tftpboot/pxelinux.cfg/default` from the `kiwi-pxeboot` package. The minimal configuration required to boot the example image looks like to following:

   ```
   DEFAULT KIWI-Boot

   LABEL KIWI-Boot
       kernel boot/linux
   ```

(continues on next page)

(continued from previous page)

```
    append initrd=boot/initrd
    IPAPPEND 2
```

7. Create the image client configuration file:

   ```
   $ vi /srv/tftpboot/KIWI/config.default

   IMAGE=/dev/ram1;Leap-15.1_appliance.x86_64;1.15.1;192.168.100.2;
   ↪4096
   UNIONFS_CONFIG=/dev/ram2,/dev/ram1,overlay
   ```

   All PXE boot based deployment methods are controlled by a client configuration file. The above configuration tells the client where to find the image and how to activate it. In this case the image will be deployed into a ramdisk (ram1) and overlay mounted such that all write operations will land in another ramdisk (ram2). KIWI NG supports a variety of different deployment strategies based on the rootfs image created beforehand. For details, refer to *PXE Client Setup Configuration*

8. Connect the client to the network and boot. This can also be done in a virtualized environment using QEMU as follows:

   ```
   $ qemu -boot n -m 4096
   ```

11.12.1 PXE Client Setup Configuration

All PXE boot based deployment methods are controlled by configuration files located in `/srv/tftpboot/KIWI` on the PXE server. Such a configuration file can either be client-specific (config.MAC_ADDRESS, for example config.00.AB.F3.11.73.C8), or generic (config.default).

In an environment with heterogeneous clients, this allows to have a default configuration suitable for the majority of clients, to have configurations suitable for a group of clients (for example machines with similar or identical hardware), and individual configurations for selected machines.

The configuration file contains data about the image and about configuration, synchronization, and partition parameters. The configuration file has got the following general format:

```
IMAGE="device;name;version;srvip;bsize;compressed,...,"

DISK="device"
PART="size;id;Mount,...,size;id;Mount"
RAID="raid-level;device1;device2;..."

AOEROOT=ro-device[,rw-device]
NBDROOT="ip-address;export-name;device;swap-export-name;swap-device;
↪write-export-name;write-device"
NFSROOT="ip-address;path"

```

(continues on next page)

(continued from previous page)

```
UNIONFS_CONFIGURATION="rw-partition,compressed-partition,overlayfs"

CONF="src;dest;srvip;bsize;[hash],...,src;dest;srvip;bsize;[hash]"

KIWI_BOOT_TIMEOUT="seconds"
KIWI_KERNEL_OPTIONS="opt1 opt2 ..."

REBOOT_IMAGE=1
RELOAD_CONFIG=1
RELOAD_IMAGE=1
```

Note: Quoting the Values

The configuration file is sourced by Bash, so the same quoting rules as for Bash apply.

Not all configuration options needs to be specified. It depends on the setup of the client which configuration values are required. The following is a collection of client setup examples which covers all supported PXE client configurations.

Setup Client with Remote Root

To serve the image from a remote location and redirect all write operations on a tmpfs, the following setup is required:

```
# When using AoE, see vblade toolchain for image export

AOEROOT=/dev/etherd/e0.1
UNIONFS_CONFIG=tmpfs,aoe,overlay

# When using NFS, see exports manual page for image export

NFSROOT="192.168.100.2;/srv/tftpboot/image/root"
UNIONFS_CONFIG=tmpfs,nfs,overlay

# When using NBD, see nbd-server manual page for image export

NBDROOT=192.168.100.2;root_export;/dev/nbd0
UNIONFS_CONFIG=tmpfs,nbd,overlay
```

The above setup shows the most common use case where the image built with KIWI NG is populated over the network using either AoE, NBD or NFS in combination with overlayfs which redirects all write operations to be local to the client. In any case a setup of either AoE, NBD or NFS on the image server is required beforehand.

Setup Client with System on Local Disk

To deploy the image on a local disk the following setup is required:

Note: In the referenced suse-leap-15.1 XML description the pxe type must be changed as follows and the image needs to be rebuild:

```
<type image="pxe" filesystem="ext3" boot="netboot/suse-leap15.1"/>
```

```
IMAGE="/dev/sda2;Leap-15.1_appliance.x86_64;1.15.1;192.168.100.2;
↪4096"
DISK="/dev/sda"
PART="5;S;X,X;L;/"
```

The setup above will create a partition table on sda with a 5MB swap partition (no mountpoint) and the rest of the disk will be a Linux(L) partition with / as mountpoint. The (X) in the PART setup specifies a place holder to indicate the default behaviour.

Setup Client with System on Local MD RAID Disk

To deploy the image on a local disk with prior software RAID configuration, the following setup is required:

Note: In the referenced suse-leap-15.1 XML description the pxe type must be changed as follows and the image needs to be rebuild:

```
<type image="pxe" filesystem="ext3" boot="netboot/suse-leap15.1"/>
```

```
RAID="1;/dev/sda;/dev/sdb"
IMAGE="/dev/md1;Leap-15.1_appliance.x86_64;1.15.1;192.168.100.2;4096
↪"
PART="5;S;x,x;L;/"
```

The first parameter of the RAID line is the RAID level. So far only raid1 (mirroring) is supported. The second and third parameter specifies the raid disk devices which make up the array. If a RAID line is present all partitions in PART will be created as RAID partitions. The first RAID is named md0, the second one md1 and so on. It is required to specify the correct RAID partition in the IMAGE line according to the PART setup. In this case md0 is reserved for the SWAP space and md1 is reserved for the system.

Setup Loading of Custom Configuration File(s)

In order to load for example a custom `/etc/hosts` file on the client, the following setup is required:

```
CONF="hosts;/etc/hosts;192.168.1.2;4096;ffffffff"
```

On boot of the client KIWI NG's boot code will fetch the `hosts` file from the root of the server (192.168.1.2) with 4k blocksize and deploy it as `/etc/hosts` on the client. The protocol is by default tftp but can be changed via the `kiwiservertype` kernel commandline option. For details, see *Setup a Different Download Protocol and Server*

Setup Client to Force Reload Image

To force the reload of the system image even if the image on the disk is up-to-date, the following setup is required:

```
RELOAD_IMAGE=1
```

The option only applies to configurations with a DISK/PART setup

Setup Client to Force Reload Configuration Files

To force the reload of all configuration files specified in CONF, the following setup is required:

```
RELOAD_CONFIG=1
```

By default only configuration files which has changed according to their md5sum value will be reloaded. With the above setup all files will be reloaded from the PXE server. The option only applies to configurations with a DISK/PART setup

Setup Client for Reboot After Deployment

To reboot the system after the initial deployment process is done the following setup is required:

```
REBOOT_IMAGE=1
```

Setup custom kernel boot options

To deactivate the kernel mode setting on local boot of the client the following setup is required:

```
KIWI_KERNEL_OPTIONS="nomodeset"
```

Note: This does not influence the kernel options passed to the client if it boots from the network. In order to setup those the PXE configuration on the PXE server needs to be changed

Setup a Custom Boot Timeout

To setup a 10sec custom timeout for the local boot of the client the following setup is required.

```
KIWI_BOOT_TIMEOUT="10"
```

Note: This does not influence the boot timeout if the client boots off from the network.

Setup a Different Download Protocol and Server

By default all downloads controlled by the KIWI NG linuxrc code are performed by an atftp call using the TFTP protocol. With PXE the download protocol is fixed and thus you cannot change the way how the kernel and the boot image (`initrd`) is downloaded. As soon as Linux takes over, the download protocols http, https and ftp are supported too. KIWI NG uses the curl program to support the additional protocols.

To select one of the additional download protocols the following kernel parameters need to be specified

```
kiwiserver=192.168.1.1 kiwiservertype=ftp
```

To set up this parameters edit the file `/srv/tftpboot/pxelinux.cfg/default` on your PXE boot server and change the append line accordingly.

Note: Once configured all downloads except for kernel and initrd are now controlled by the given server and protocol. You need to make sure that this server provides the same directory and file structure as initially provided by the `kiwi-pxeboot` package

11.13 Booting a Root Filesystem from Network

Abstract

This page provides further information for handling KIS images built with KIWI NG and references the following article:

- *Build KIS Image (Kernel, Initrd, System)*

In KIWI NG, the `kiwi-overlay` dracut module can be used to boot from a remote exported root filesystem. The exported device is visible as block device on the network client. KIWI NG supports the two export backends `NBD` (Network Block Device) and `AoE` (ATA over Ethernet) for this purpose. A system that is booted in this mode will read the contents of the root filesystem from a remote location and targets any write action into RAM by default. The kernel cmdline option `rd.root.overlay.write` can be used to specify an alternative device to use for writing. The two layers (read and write) are combined using the `overlayfs` filesystem.

For remote boot of a network client, the PXE boot protocol is used. This functionality requires a network boot server setup on the system. Details how to setup such a server can be found in *Setting Up a Network Boot Server*.

Before the KIS image can be build, the following configuration step is required:

- Create dracut configuration to include the `kiwi-overlay` module

```
$ cd kiwi-descriptions/suse/x86_64/suse-leap-15.1
$ mkdir -p root/etc/dracut.conf.d
$ cd root/etc/dracut.conf.d
$ echo 'add_dracutmodules+=" kiwi-overlay "' > overlay.
↪conf
```

Now the KIS image can be build as shown in *Build KIS Image (Kernel, Initrd, System)*. After the build, the following configuration steps are required to boot from the network:

1. Copy initrd/kernel from the KIS build to the PXE server

   The PXE boot process loads the configured kernel and initrd from the PXE server. For this reason, the following files must be copied to the PXE server as follows:

```
$ cp *.initrd.xz /srv/tftpboot/boot/initrd
$ cp *.kernel /srv/tftpboot/boot/linux
```

2. Export Root FileSystem to the Network

   Access to the root filesystem is implemented using either the AoE or the NBD protocol. This requires the export of the root filesystem image as remote block device:

   Export via AoE: Install the `vblade` package on the system which is expected to export the root filesystem

   Note: Not all versions of AoE are compatible with any kernel. This means the kernel on the system exporting the root filesystem image must provide a compatible aoe kernel module compared to the kernel used inside of the root filesystem image.

   Once done, export the filesystem from the KIS build above as follows:

```
$ vbladed 0 1 IFACE Leap-15.1_appliance.x86_64-1.15.1
```

   The above command exports the given filesystem image file as a block storage device to the network of the given IFACE. On any machine except the one exporting the file, it will appear as `/dev/etherd/e0.1` once the **aoe** kernel module was loaded. The two numbers, 0 and 1 in the above example, classifies a major and minor number which is used in the device node name on the reading side, in this case `e0.1`.

Note: Only machines in the same network of the given INTERFACE can see the exported block device.

Export via NBD: Install the `nbd` package on the system which is expected to export the root filesystem

Once done, export the filesystem from the KIS build above as follows:

```
$ losetup /dev/loop0 Leap-15.1_appliance.x86_64-1.15.1

$ vi /etc/nbd-server/config

  [generic]
      user = root
      group = root
  [export]
      exportname = /dev/loop0

$ nbd-server
```

3. Setup boot entry in the PXE configuration

Note: The following step assumes that the pxelinux.0 loader has been configured on the boot server to boot up network clients

Edit the file `/srv/tftpboot/pxelinux.cfg/default` and create a boot entry of the form:

Using NBD:

```
LABEL Overlay-Boot
    kernel boot/linux
    append initrd=boot/initrd root=overlay:nbd=server-
↪ip:export
```

The boot parameter `root=overlay:nbd=server-ip:export` specifies the NBD server IP address and the name of the export as used in `/etc/nbd-server/config`

Using AoE:

```
LABEL Overlay-Boot
    kernel boot/linux
    append initrd=boot/initrd root=overlay:aoe=AOEINTERFACE
```

The boot parameter `root=overlay:aoe=AOEINTERFACE` specifies the interface name as it was exported by the **`vbladed`** command

4. Boot from the Network

Within the network which has access to the PXE server and the exported root filesystem image, any network client can now boot the system. A test based on QEMU can be done as follows:

```
$ qemu -boot n
```

11.14 Booting a Live ISO Image from Network

Abstract

This page provides further information for handling ISO images built with KIWI NG and references the following articles:

- *Build an ISO Hybrid Live Image*

In KIWI NG, live ISO images can be configured to boot via the PXE boot protocol. This functionality requires a network boot server setup on the system. Details how to setup such a server can be found in *Setting Up a Network Boot Server*.

After the live ISO was built as shown in *Build an ISO Hybrid Live Image*, the following configuration steps are required to boot from the network:

1. Extract initrd/kernel From Live ISO

   The PXE boot process loads the configured kernel and initrd from the PXE server. For this reason, those two files must be extracted from the live ISO image and copied to the PXE server as follows:

   ```
   $ mount Leap-15.1_appliance.x86_64-1.15.1.iso /mnt
   $ cp /mnt/boot/x86_64/loader/initrd /srv/tftpboot/boot/initrd
   $ cp /mnt/boot/x86_64/loader/linux /srv/tftpboot/boot/linux
   $ umount /mnt
   ```

   Note: This step must be repeated with any new build of the live ISO image

2. Export Live ISO To The Network

   Access to the live ISO file is implemented using the AoE protocol in KIWI NG. This requires the export of the live ISO file as remote block device which is typically done with the **vblade** toolkit. Install the following package on the system which is expected to export the live ISO image:

   ```
   $ zypper in vblade
   ```

   Note: Not all versions of AoE are compatible with any kernel. This means the kernel on the system exporting the live ISO image must provide a compatible aoe kernel module compared to the kernel used in the live ISO image.

Once done, export the live ISO image as follows:

```
$ vbladed 0 1 INTERFACE Leap-15.1_appliance.x86_64-1.15.1.iso
```

The above command exports the given ISO file as a block storage device to the network of the given INTERFACE. On any machine except the one exporting the file, it will appear as `/dev/etherd/e0.1` once the **aoe** kernel module was loaded. The two numbers, 0 and 1 in the above example, classifies a major and minor number which is used in the device node name on the reading side, in this case `e0.1`. The numbers given at export time must match the AOEINTERFACE name as described in the next step.

Note: Only machines in the same network of the given INTERFACE can see the exported live ISO image. If virtual machines are the target to boot the live ISO image they could all be connected through a bridge. In this case INTERFACE is the bridge device. The availability scope of the live ISO image on the network is in general not influenced by KIWI NG and is a task for the network administrators.

3. Setup live ISO boot entry in PXE configuration

Note: The following step assumes that the pxelinux.0 loader has been configured on the boot server to boot up network clients

Edit the file `/srv/tftpboot/pxelinux.cfg/default` and create a boot entry of the form:

```
LABEL Live-Boot
    kernel boot/linux
    append initrd=boot/initrd rd.kiwi.live.pxe␣
↪root=live:AOEINTERFACE=e0.1
```

- The boot parameter `rd.kiwi.live.pxe` tells the KIWI NG boot process to setup the network and to load the required `aoe` kernel module.
- The boot parameter `root=live:AOEINTERFACE=e0.1` specifies the interface name as it was exported by the **vbladed** command from the last step. Currently only AoE (Ata Over Ethernet) is supported.

4. Boot from the Network

Within the network which has access to the PXE server and the exported live ISO image, any network client can now boot the live system. A test based on QEMU is done as follows:

```
$ qemu -boot n
```

11.15 Setting Up YaST at First Boot

Abstract

This page provides information how to setup the KIWI NG XML description to start the SUSE YaST system setup utility at first boot of the image

To be able to use YaST in a non interactive way, create a YaST profile which tells the autoyast module what to do. To create the profile, run:

```
yast autoyast
```

Once the YaST profile exists, update the KIWI NG XML description as follows:

1. Edit the KIWI NG XML file and add the following package to the `<packages type="image">` section:

   ```
   <package name="yast2-firstboot"/>
   ```

2. Copy the YaST profile file as overlay file to your KIWI NG image description overlay directory:

   ```
   cd IMAGE_DESCRIPTION_DIRECTORY
   mkdir -p root/etc/YaST2
   cp PROFILE_FILE root/etc/YaST2/firstboot.xml
   ```

3. Copy and activate the YaST firstboot template. This is done by the following instructions which needs to be written into the KIWI NG `config.sh` which is stored in the image description directory:

   ```
   sysconfig_firsboot=/etc/sysconfig/firstboot
   sysconfig_template=/var/adm/fillup-templates/sysconfig.firstboot
   if [ ! -e "${sysconfig_firsboot}" ]; then
       cp "${sysconfig_template}" "${sysconfig_firsboot}"
   fi

   touch /var/lib/YaST2/reconfig_system
   ```

11.16 Add or Update the Fstab File

Abstract

This page provides further information for customizing the `/etc/fstab` file which controls the mounting of filesystems at boot time.

In KIWI NG, all major filesystems that were created at image build time are handled by KIWI NG itself and setup in `/etc/fstab`. Thus there is usually no need to add entries or change the ones added by KIWI NG. However depending on where the image is deployed later it might be required to pre-populate fstab entries that are unknown at the time the image is build.

Possible use cases are for example:

- Adding NFS locations that should be mounted at boot time. Using autofs would be an alternative to avoid additional entries to fstab. The information about the NFS location will make this image specific to the target network. This will be independent of the mount method, either fstab or the automount map has to provide it.
- Adding or changing entries in a read-only root system which becomes effective on first boot but can't be added at that time because of the read-only characteristics.

Note: Modifications to the fstab file are a critical change. If done wrongly the risk exists that the system will not boot. In addition this type of modification makes the image specific to its target and creates a dependency to the target hardware, network, etc… Thus this feature should be used with care.

The optimal way to provide custom fstab information is through a package. If this can't be done the files can also be provided via the overlay file tree of the image description.

KIWI NG supports three ways to modify the contents of the `/etc/fstab` file:

Providing an `/etc/fstab.append` file If that file exists in the image root tree, KIWI NG will take its contents and append it to the existing `/etc/fstab` file. The provided `/etc/fstab.append` file will be deleted after successful modification.

Providing an `/etc/fstab.patch` file The `/etc/fstab.patch` represents a patch file that will be applied to `/etc/fstab` using the `patch` program. This method also allows to change the existing contents of `/etc/fstab`. On success `/etc/fstab.patch` will be deleted.

Providing an `/etc/fstab.script` file The `/etc/fstab.script` represents an executable which is called as chrooted process. This method is the most flexible one and allows to apply any change. On success `/etc/fstab.script` will be deleted.

Note: All three variants to handle the fstab file can be used together. Appending happens first, patching afterwards and the script call is last. When using the script call, there is no validation that checks if the script actually handles fstab or any other file in the image rootfs.

11.17 Building Images with Profiles

KIWI NG supports so-called *profiles* inside the XML image description. Profiles act as namespaces for additional settings to be applied on top of the defaults. For further details, see *Image Profiles*.

11.17.1 Local Builds

To execute local KIWI NG builds with a specific, selected profile, add the command line flag `--profile=$PROFILE_NAME`:

```
$ sudo kiwi-ng --type iso --profile=workstation system build \
      --description kiwi-descriptions/suse/x86_64/suse-leap-15.1 \
      --target-dir /tmp/myimage
```

Consult the manual page of `kiwi` for further details: *SYNOPSIS*.

11.17.2 Building with the Open Build Service

The Open Build Service (OBS) support profiles via the multibuild feature. An example project using this feature is the openSUSE-Tumbleweed image.

To enable and use the profiles, follow these steps:

1. Add the following XML comment to your `config.xml`:

   ```
   <!-- OBS-Profiles: @BUILD_FLAVOR@ -->
   ```

   It must be added before the opening `<image>` element and after the `<?xml?>` element, e.g.:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <!-- OBS-Profiles: @BUILD_FLAVOR@ -->
   <image schemaversion="7.2" name="Leap-15.1_appliance">
     <!-- snip -->
   </image>
   ```

2. Add a file `_multibuild` into your package's repository with the following contents:

   ```
   <multibuild>
     <flavor>profile_1</flavor>
     <flavor>profile_2</flavor>
   </multibuild>
   ```

   Add a line `<flavor>$PROFILE</flavor>` for each profile that you want OBS to build.

Note, by default, OBS excludes the build **without** any profile enabled.

Running a build of a multibuild enabled repository via `osc` can be achieved via the `-M $PROFILE` flag:

```
$ osc build -M $PROFILE
```

11.18 Building in the Open Build Service

Note: Abstract

This document gives a brief overview how to build images with KIWI NG in version 9.23.20 inside of the Open Build Service. A tutorial on the Open Buildservice itself can be found here: https://en.opensuse.org/openSUSE:Build_Service_Tutorial

The next generation KIWI NG is fully integrated with the Open Build Service. In order to start it's best to checkout one of the integration test image build projects from the base Testing project `Virtualization:Appliances:Images:Testing_$ARCH:$DISTRO` at:

https://build.opensuse.org

For example the test images for SUSE on x86 can be found here.

11.18.1 Advantages of using the Open Build Service (OBS)

The Open Build Service offers multiple advantages over running KIWI NG locally:

- OBS will host the latest successful build for you without having to setup a server yourself.
- As KIWI NG is fully integrated into OBS, OBS will automatically rebuild your images if one of the included packages or one of its dependencies or KIWI NG itself get updated.
- The builds will no longer have to be executed on your own machine, but will run on OBS, thereby saving you resources. Nevertheless, if a build fails, you get a notification via email (if enabled in your user's preferences).

11.18.2 Differences Between Building Locally and on OBS

Note, there is a number of differences when building images with KIWI NG using the Open Build Service. Your image that build locally just fine, might not build without modifications.

The notable differences to running KIWI NG locally include:

- OBS will pick the KIWI NG package from the repositories configured in your project, which will most likely not be the same version that you are running locally. This is especially relevant when building images for older versions like SUSE Linux Enterprise. Therefore, include the custom appliances repository as described in the following section: *Recommendations*.

- When KIWI NG runs on OBS, OBS will extract the list of packages from `config.xml` and use it to create a build root. In contrast to a local build (where your distributions package manager will resolve the dependencies and install the packages), OBS will **not** build your image if there are multiple packages that could be chosen to satisfy the dependencies of your packages[1]. This shows errors like this:

  ```
  unresolvable: have choice for SOMEPACKAGE: SOMEPAKAGE_1␣
   ↪SOMEPACKAGE_2
  ```

  This can be solved by explicitly specifying one of the two packages in the project configuration via the following setting:

  ```
  Prefer: SOMEPACKAGE_1
  ```

  Place the above line into the project configuration, which can be accessed either via the web interface (click on the tab `Project Config` on your project's main page) or via `osc meta -e prjconf`.

  Warning: We strongly encourage you to remove your repositories from `config.xml` and move them to the repository configuration in your project's settings. This usually prevents the issue of having the choice for multiple package version and results in a much smoother experience when using OBS.

- By default, OBS builds only a single build type and the default profile. If your appliance uses multiple build types, put each build type into a profile, as OBS cannot handle multiple build types.

  There are two options to build multiple profiles on OBS:

  1. Use the `<image>` element and add it bellow the XML declaration (`<?xml ..?>`):

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

     <!-- OBS-Profiles: foo_profile bar_profile -->

     <image schemaversion="7.2" name="openSUSE-Leap-15.1">
       <!-- image description with the profiles foo_profile and␣
      ↪bar_profile
     </image>
     ```

  2. Use the multibuild feature.

  The first option is simpler to use, but has the disadvantage that your appliances are built sequentially. The `multibuild` feature allows to build each profile as a single package, thereby enabling parallel execution, but requires an additional _`multibuild` file. For the above example `config.xml` would have to be adapted as follows:

[1] This is a design decision made by OBS: as it's purpose is to build packages in a reproducible fashion it cannot make a decision which package to choose from multiple available ones. A package manager build for end-users on the other hand **must** make an a choice, as it would be otherwise hardly usable.

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

<!-- OBS-Profiles: @BUILD_FLAVOR@ -->

<image schemaversion="7.2" name="openSUSE-Leap-15.1">
  <!-- image description with the profiles foo_profile and bar_
↪profile
</image>
```

The file `_multibuild` would have the following contents:

```
<multibuild>
  <flavor>foo_profile</flavor>
  <flavor>bar_profile</flavor>
</multibuild>
```

- Subfolders in OBS projects are ignored by default by **osc** and must be explicitly added via **osc add $FOLDER**[2]. Bear that in mind when adding the overlay files inside the `root/` directory to your project.
- OBS ignores file permissions. Therefore `config.sh` and `images.sh` will **always** be executed through BASH (see also: *User Defined Scripts*).

11.18.3 Recommendations

Working with OBS

Although OBS is an online service, it is not necessary to test every change by uploading it. OBS will use the same process as `osc build` does, so if your image builds locally via `osc build` it should also build online on OBS.

Repository Configuration

When setting up the project, enable the `images` repository: the `images` repository's checkbox can be found at the bottom of the selection screen that appears when clicking `Add from a Distribution` in the `Repositories` tab. Or specify it manually in the project configuration (it can be accessed via `osc meta -e prj`):

```
<repository name="images">
  <arch>x86_64</arch>
</repository>
```

Furthermore, OBS requires additional repositories from which it obtains your dependent packages. These repositories can be provided in two ways:

[2] `osc` compresses added folders into a cpio archive and decompresses it before running your builds. The only downside of this is, that the contents of your overlay is not conveniently visible via the web interface.

1. Add the repositories to the project configuration on OBS and omit them from `config.xml`. Provide only the following repository inside the image description:

   ```
   <repository type="rpm-md">
     <source path="obsrepositories:/"/>
   </repository>
   ```

   This instructs OBS to inject the repositories from your project into your appliance.

   Additional repositories can be added by invoking `osc meta -e prj` and adding a line of the following form as a child of `<repository name="images">`:

   ```
   <path project="$OBS_PROJECT" repository="$REPOSITORY_NAME"/>
   ```

   The order in which you add repositories matters: if a package is present in multiple repositories, then it is taken from the **first** repository. The **last** repository is subject to path expansion: its repository paths are included as well.

   Don't forget to add the repository from the `Virtualization:Appliances:Builder` project, providing the latest stable version of KIWI NG (which you are very likely using for your local builds).

   The following example repository configuration[3] adds the repositories from the `Virtualization:Appliances:Builder` project and those from the latest snapshot of openSUSE Tumbleweed:

   ```
   <project name="Virtualization:Appliances:Images:openSUSE-
   ↪Tumbleweed">
     <title>Tumbleweed JeOS images</title>
     <description>Host JeOS images for Tumbleweed</description>
     <repository name="images">
       <path project="Virtualization:Appliances:Builder"␣
   ↪repository="Factory"/>
       <path project="openSUSE:Factory" repository="snapshot"/>
       <arch>x86_64</arch>
     </repository>
   </project>
   ```

   The above can be simplified further using the path expansion of the last repository to:

   ```
   <project name="Virtualization:Appliances:Images:openSUSE-
   ↪Tumbleweed">
     <title>Tumbleweed JeOS images</title>
     <description>Host JeOS images for Tumbleweed</description>
     <repository name="images">
       <path project="Virtualization:Appliances:Builder"␣
   ↪repository="Factory"/>
       <arch>x86_64</arch>
     </repository>
   </project>
   ```

[3] Taken from the project Virtualization:Appliances:Images:openSUSE-Tumbleweed

Now `Virtualization:Appliances:Builder` is the last repository, which' repositories are included into the search path. As `openSUSE:Factory/snapshot` is among these, it can be omitted from the repository list.

2. Keep the repositories in your `config.xml` configuration file. If you have installed the latest stable KIWI NG as described in *Installation* then you should add the following repository to your projects configuration (accessible via **`osc meta -e prjconf`**), so that OBS will pick the latest stable KIWI NG version too:

```
<repository name="images">
  <path project="Virtualization:Appliances:Builder" repository="
↪$DISTRO"/>
  <arch>x86_64</arch>
</repository>
```

Replace `$DISTRO` with the appropriate name for the distribution that you are currently building and optionally adjust the architecture.

We recommend to use the first method, as it integrates better into OBS. Note that your image description will then no longer build outside of OBS though. If building locally is required, use the second method.

Warning: Adding the repositories to project's configuration makes it impossible to build images for different distributions from the same project.

Since the repositories are added for every package in your project, all your image builds will share the same repositories, thereby resulting in conflicts for different distributions.

We recommend to create a separate project for each distribution. If that is impossible, you can keep all your repositories (including `Virtualization:Appliances:Builder`) in `config.xml`. That however usually requires a large number of workarounds via `Prefer:` settings in the project configuration and is thus **not** recommended.

Project Configuration

The Open Build Service will by default create the same output file as KIWI NG when run locally, but with a custom filename ending (that is unfortunately unpredictable). This has the consequence that the download URL of your image will change with every rebuild (and thus break automated scripts). OBS can create symbolic links with static names to the latest build by adding the following line to the project configuration:

```
Repotype: staticlinks
```

If build Vagrant images (see *Image Description for Vagrant*) add the repository-type `vagrant`. OBS creates a `boxes/` subdirectory in your download repositories, which contains JSON files for Vagrant[4].

[4] Vagrant uses these JSON files for automatic updates of your Vagrant boxes.

If you have added your repositories to `config.xml`, you probably see errors of the following type:

```
unresolvable: have choice for SOMEPACKAGE: SOMEPAKAGE_1 SOMEPACKAGE_
↪2
```

Instead of starting from scratch and manually adding `Prefer:` statements to the project configuration, we recommend to copy the current project configuration of the testing project `Virtualization:Appliances:Images:Testing_$ARCH:$DISTRO` into your own project. It provides a good starting point and can be adapted to your OBS project.

11.19 Using SUSE Product ISO To Build

Abstract

This page provides information how to use the SUSE media ISO with KIWI NG

When building an image with KIWI NG, the image description usually points to a number of public/private package source repositories from which the new image root tree will be created. Alternatively the vendor provided product ISO image(s) can be used. The contents of the ISO (DVD) media also provides package source repositories but organized in a vendor specific structure. As a user it's important to know about this structure such that the KIWI NG image description can consume it.

To use a SUSE product media the following steps are required:

1. Mount the ISO media from file or DVD drive:

```
$ sudo mount Product_ISO_file.iso|DVD_drive /media/suse
```

2. Lookup all `Product` and `Module` directories:

   Below `/media/suse` there is a directory structure which provides package repositories in directories starting with `Product-XXX` and `Module-XXX`. It depends on the package list in the KIWI NG image description from which location a package or a dependency of the package is taken. Therefore it is best practice to browse through all the directories and create a `<repository>` definition for each of them in the KIWI NG image description like the following example shows:

   ```
   <repository alias="DVD-1-Product-SLES">
       <source path="file:///media/suse/Product-SLES"/>
   </repository>

   <repository alias="DVD-1-Module-Basesystem">
       <source path="file:///media/suse/Module-Basesystem"/>
   </repository>
   ```

Once all the individual product and module repos has been created in the KIWI NG image description, the build process can be started as usual.

Note: Because of the manual mount process the `/media/suse` location stays busy after KIWI NG has created the image. The cleanup of this resource is a responsibility of the user and not done by KIWI NG

CHAPTER TWELVE

CONTRIBUTING

Note: Abstract

This document describes the development process of KIWI NG and how you can be part of it. This description applies to version 9.23.20.

12.1 Using KIWI NG in a Python Project

Note: Abstract

KIWI NG is provided as python module under the **kiwi** namespace. It is available for the python 3 version. The following description applies for KIWI NG version 9.23.20.

KIWI NG can also function as a module for other Python projects. The following example demonstrates how to read an existing image description, add a new repository definition and export the modified description on stdout.

```python
import sys
import logging

from kiwi.xml_description import XMLDescription
from kiwi.xml_state import XMLState

description = XMLDescription('path/to/kiwi/XML/config.xml')

xml_data = description.load()

xml_state = XMLState(
    xml_data=xml_data, profiles=[], build_type='iso'
)

xml_state.add_repository(
    repo_source='http://repo',
```

(continues on next page)

(continued from previous page)

```
    repo_type='rpm-md',
    repo_alias='myrepo',
    repo_prio=99
)

xml_data.export(
    outfile=sys.stdout, level=0
)
```

All classes are written in a way to care for a single responsibility in order to allow for re-use on other use cases. Therefore it is possible to use KIWI NG outside of the main image building scope to manage e.g the setup of loop devices, filesystems, partitions, etc. . .

This means KIWI NG provides you a way to describe a system but you are free to make use of the kiwi description format or not. The following example shows how to use kiwi to create a simple filesystem image which contains your host `tmp` directory.

```
import logging

from kiwi.storage.loop_device import LoopDevice
from kiwi.filesystem import FileSystem

loop_provider = LoopDevice(
    filename='my_tmp.ext4', filesize_mbytes=100
)
loop_provider.create()

filesystem = FileSystem.new(
    'ext4', loop_provider, '/tmp/'
)
filesystem.create_on_device(
    label='TMP'
)
filesystem.sync_data()
```

12.2 Plugin Architecture

Each command provided by KIWI NG is written as a task plugin under the **kiwi.tasks** namespace. As a developer you can extend KIWI NG with custom task plugins if the following conventions are taken into account:

12.2.1 Naming Conventions

Task Plugin File Name The file name of a task plugin must follow the pattern `<service>_<command>.py`. This allows to invoke the task with **`kiwi-ng service command ...`**

Task Plugin Option Handling KIWI NG uses the docopt module to handle options. Each task plugin must use docopt to allow option handling.

Task Plugin Class The implementation of the plugin must be a class that matches the naming convention: `<Service><Command>Task`. The class must inherit from the `CliTask` base class. On startup of the plugin, KIWI NG expects an implementation of the `process` method.

Task Plugin Entry Point Registration of the plugin must be done in `setup.py` using the `entry_points` concept from Python's setuptools.

```
'packages': ['kiwi_plugin'],
'entry_points': {
    'kiwi.tasks': [
        'service_command=kiwi_plugin.tasks.service_command'
    ]
}
```

12.2.2 Example Plugin

Note: The following example assumes an existing Python project which was set up according to the Python project rules and standards.

1. Assuming the project namespace is **kiwi_relax_plugin**.

   Create the task plugin directory `kiwi_relax_plugin/tasks`

2. Create the entry point in **`setup.py`**.

   Assuming we want to create the service named **relax** providing the command **justdoit** this would be the following entry point definition in `setup.py`:

   ```
   'packages': ['kiwi_relax_plugin'],
   'entry_points': {
       'kiwi.tasks': [
           'relax_justdoit=kiwi_relax_plugin.tasks.relax_justdoit'
       ]
   }
   ```

3. Create the plugin code in the file `kiwi_relax_plugin/tasks/relax_justdoit.py` with the following content:

```
"""
usage: kiwi-ng relax justdoit -h | --help
       kiwi-ng relax justdoit --now

commands:
    justdoit
        time to relax

options:
    --now
        right now. For more details about docopt
        see: http://docopt.org
"""
# These imports requires kiwi to be part of your environment
# It can be either installed from pip into a virtual development
# environment or from the distribution package manager
from kiwi.tasks.base import CliTask
from kiwi.help import Help

class RelaxJustdoitTask(CliTask):
    def process(self):
        self.manual = Help()
        if self.command_args.get('help') is True:
            # The following will invoke man to show the man page
            # for the requested command. Thus for the call to
            # succeed a manual page needs to be written and
            # installed by the plugin
            return self.manual.show('kiwi::relax::justdoit')

        print(
            'https://genius.com/Frankie-goes-to-hollywood-relax-
↪lyrics'
        )
```

4. Test the plugin

```
$ ./setup.py develop
$ kiwi-ng relax justdoit --now
```

12.3 Extending KIWI NG with Custom Operations

Note: Abstract

Users building images with KIWI NG need to implement their own infrastructure if the image description does not provide a way to embed custom information which is outside of the scope of the general schema as it is provided by KIWI NG today.

This document describes how to create an extension plugin for the KIWI NG schema to add and validate additional information in the KIWI NG image description.

Such a schema extension can be used in an additional KIWI NG task plugin to provide a new subcommand for KIWI NG. As of today there is no other plugin interface except for providing additional KIWI NG commands implemented.

Depending on the demand for custom plugins, the interface to hook in code into other parts of the KIWI NG processing needs to be extended.

This description applies for version 9.23.20.

12.3.1 The <extension> Section

The main KIWI NG schema supports an extension section which allows to specify any XML structure and attributes as long as they are connected to a namespace. According to this any custom XML structure can be implemented like the following example shows:

```
<image>
    ...
    <extension xmlns:my_plugin="http://www.my_plugin.com">
        <my_plugin:my_feature>
            <my_plugin:title name="cool stuff"/>
        </my_plugin:my_feature>
    </extension>
</image>
```

- Any toplevel namespace must exist only once
- Multiple different toplevel namespaces are allowed, e.g my_plugin_a, my_plugin_b

12.3.2 RELAX NG Schema for the Extension

If an extension section is found, KIWI NG looks up its namespace and asks the main XML catalog for the schema file to validate the extension data. The schema file must be a RELAX NG schema in the .rng format. We recommend to place the schema as `/usr/share/xml/kiwi/my_plugin.rng`

For the above example the RELAX NG Schema in the compressed format `my_plugin.rnc` would look like this:

```
namespace my_plugin = "http://www.my_plugin.com"

start =
    k.my_feature

div {
    k.my_feature.attlist = empty
    k.my_feature =
```

(continues on next page)

(continued from previous page)

```
        element my_plugin:my_feature {
            k.my_feature.attlist &
            k.title
        }
}

div {
    k.title.name.attribute =
        attribute name { text }
    k.title.attlist = k.title.name.attribute
    k.title =
        element my_plugin:title {
            k.title.attlist
        }
}
```

In order to convert this schema to the .rng format just call:

```
$ trang -I rnc -O rng my_plugin.rnc /usr/share/xml/kiwi/my_plugin.
↪rng
```

12.3.3 Extension Schema in XML catalog

As mentioned above the mapping from the extension namespace to the correct RELAX NG schema file is handled by a XML catalog file. The XML catalog for the example use here looks like this:

```
<?xml version="1.0"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
    <system
        systemId="http://www.my_plugin.com"
        uri="file:////usr/share/xml/kiwi/my_plugin.rng"/>
</catalog>
```

For resolving the catalog KIWI NG uses the **`xmlcatalog`** command and the main XML catalog from the system which is `/etc/xml/catalog`.

Note: It depends on the distribution and its version how the main catalog gets informed about the existence of the KIWI NG extension catalog file. Please consult the distribution manual about adding XML catalogs.

If the following command provides the information to the correct RELAX NG schema file you are ready for a first test:

```
$ xmlcatalog /etc/xml/catalog http://www.my_plugin.com
```

12.3.4 Using the Extension

In order to test your extension place the example extension section from the beginning of this document into one of your image description's `config.xml` file

The following example will read the name attribute from the title section of the my_feature root element and prints it:

```
import logging

from kiwi.xml_description import XMLDescription

description = XMLDescription('path/to/kiwi/XML/config.xml')
description.load()

my_plugin = description.get_extension_xml_data('my_plugin')

print(my_plugin.getroot()[0].get('name'))
```

12.4 The Basics

The core appliance builder is developed in Python and follows the test driven development rules.

If you want to implement a bigger feature, consider opening an issue on GitHub first to discuss the changes. Or join the discussion in the `#kiwi` channel on Riot.im.

12.5 Fork the upstream repository

1. On GitHub, navigate to: https://github.com/OSInside/kiwi
2. In the top-right corner of the page, click **`Fork`**.

12.6 Create a local clone of the forked repository

```
$ git clone https://github.com/YOUR-USERNAME/kiwi

$ git remote add upstream https://github.com/OSInside/kiwi.git
```

12.7 Install Required Operating System Packages

KIWI NG requires the following additional packages which are not provided by **pip**:

XML processing libraries `libxml2` and `libxslt` (for `lxml`)

Python header files, GCC compiler and glibc-devel header files Required for python modules that hooks into shared library context

Spell Checking library Provided by the `enchant` library

ShellCheck ShellCheck script linter.

ISO creation program One of `xorriso` (preferred) or `genisoimage`.

LaTeX documentation build environment A full LaTeX installation is required to build the PDF documentation[1].

Host Requirements To Build Images A full set of tools needed to build images and provided by the `kiwi-systemdeps` package

The above mentioned system packages will be installed by calling the `install_devel_packages.sh` helper script from the checked out Git repository as follows:

```
$ sudo helper/install_devel_packages.sh
```

Note: The helper script checks for the package managers `zypper` and `dnf` and associates a distribution with it. If you use a distribution that does not use one of those package managers the script will not install any packages and exit with an error message. In this case we recommend to take a look at the package list encoded in the script and adapt to your distribution and package manager as needed.

12.8 Create a Python Virtual Development Environment

The following commands initializes and activates a development environment for Python 3:

```
$ tox -e devel
$ source .tox/3/bin/activate
```

The commands above automatically creates the application script called **kiwi-ng**, which allows you to run KIWI NG from the Python sources inside the virtual environment:

```
$ kiwi-ng --help
```

[1] Sphinx requires a plethora of additional LaTeX packages. Unfortunately there is currently no comprehensive list available. On Ubuntu/Debian installing `texlive-latex-extra` should be sufficient. For Fedora, consult the package list from `.gitlab-ci.yml`.

Warning: The virtualenv's `$PATH` will not be taken into account when calling KIWI NG via **sudo**! Use the absolute path to the KIWI NG executable to run an actual build using your local changes:

```
$ sudo $PWD/.tox/3/bin/kiwi-ng system build ...
```

To leave the development mode, run:

```
$ deactivate
```

To resume your work, **cd** into your local Git repository and call:

```
$ source .tox/3/bin/activate
```

12.9 Running the Unit Tests

We use **tox** to run the unit tests. Tox sets up its own virtualenvs inside the `.tox` directory for multiple Python versions and should thus **not** be invoked from inside your development virtualenv.

Before submitting your changes via a pull request, ensure that all tests pass and that the code has the required test coverage via the command:

```
$ tox
```

We also include `pytest-xdist` in the development virtualenv which allows to run the unit tests in parallel. It is turned off by default but can be enabled via:

```
$ tox "-n NUMBER_OF_PROCESSES"
```

where you can insert an arbitrary number as `NUMBER_OF_PROCESSES` (or a shell command like `$(nproc)`). Note that the double quotes around `-n NUMBER_OF_PROCESSES` are required (otherwise **tox** will consume this command line flag instead of forwarding it to **pytest**).

The previous call would run the unit tests for different Python versions, check the source code for errors and build the documentation.

If you want to see the available targets, use the option `-l` to let **tox** print a list of them:

```
$ tox -l
```

To only run a special target, use the `-e` option. The following example runs the test cases for the Python 3.6 interpreter only:

```
$ tox -e unit_py3_6
```

12.10 Create a Branch for each Feature or Bugfix

Code changes should be done in an extra Git branch. This allows for creating GitHub pull requests in a clean way. See also: Collaborating with issues and pull requests

```
$ git checkout -b my-topic-branch
```

Make and commit your changes.

Note: You can make multiple commits which is generally useful to give your changes a clear structure and to allow us to better review your work.

Note: Your work is important and must be signed to ensure the integrity of the repository and the code. Thus we recommend to setup a signing key as documented in *Signing Git Patches*.

```
$ git commit -S -a
```

Run the tests and code style checks. All of these are also performed by GitLab CI when a pull request is created.

```
$ tox
```

Once everything is done, push your local branch to your forked repository and create a pull request into the upstream repository.

```
$ git push origin my-topic-branch
```

Thank you much for contributing to KIWI NG. Your time and work effort is very much appreciated!

12.11 Coding Style

KIWI NG follows the general PEP8 guidelines with the following exceptions:

- We do not use free functions at all. Even utility functions must be part of a class, but should be either prefixed with the `@classmethod` or `@staticmethod` decorators (whichever is more appropriate).
- Do not set module and class level variables, put these into the classes' `__init__` method.
- The names of constants are not written in all capital letters.

12.11.1 Documentation

KIWI NG uses Sphinx for the API and user documentation.

In order to build the HTML documentation call:

```
tox -e doc
```

or to build the full documentation (including a PDF generated by LaTeX[3]):

```
tox -e packagedoc
```

Document all your classes, methods, their parameters and their types using the standard reStructuredText syntax as supported by Sphinx, an example class is documented as follows:

```
class Example:
    """
    **Example class**

    :param str param: A parameter
    :param bool : Source file name to compress
    :param list supported_zipper: List of supported compression␣
↪tools
    :attr Optional[str] attr: A class attribute
    """
    def __init__(self, param, param_w_default=False):
        self.attr = param if param_w_default else None

    def method(self, param):
        """
        A method that takes a parameter.

        :param list param: a parameter
        :return: whether param is very long
        :rtype: bool
        """
        return len(param) > 50
```

Try to stick to the following guidelines when documenting source code:

- Classes should be documented directly in their main docstring and not in `__init__`.
- Document **every** function parameter and every public attribute including their types.
- Only public methods should be documented, private methods don't have to, unless they are complex and it is not easy to grasp what they do (which should be avoided anyway).

Please also document any user-facing changes that you implementing (e.g. adding a new build type) in the user documentation, which can be found in `doc/source`. General documentation should be put into the `working_with_kiwi/` subfolder, whereas documentation about more specialized topics would belong into the `building/` subfolder.

[3] Requires a full LaTeX installation.

Adhere to a line limit of 75 characters when writing the user facing documentation[2].

12.12 Additional Information

The following sections provides further information about the repository integrity, version, package and documentation management.

12.12.1 Signing Git Patches

To ensure the integrity of the repository and the code base, patches sent for inclusion should be signed with a GPG key.

To prepare Git to sign commits, follow these instructions:

1. Create a key suitable for signing (it is not recommended to use existing keys to not mix it with your email environment):

   ```
   $ gpg2 --expert --full-gen-key
   ```

2. Either choose a RSA key for signing (option `(4)`) or an ECC key for signing (option `(10)`). For a RSA key choose a key size of 4096 bits and for a ECC key choose Curve 25519 (option `(1)`). Enter a reasonable validity period (we recommend 2 to 5 years). Complete the key generation by entering your name and email address.

3. Add the key ID to your git configuration, by running the following **`git config`** commands:

   ```
   $ git config --local user.signingkey $YOUR_SIGN_KEY_ID
   $ git config --local commit.gpgSign true
   ```

   Omitting the flag `--local` will make these settings global for all repositories (they will be added to `~/.gitconfig`). You can find your signkey's ID via:

   ```
   $ gpg2 --list-keys --keyid-format long $YOUR_EMAIL
   pub   rsa4096/AABBCCDDEEFF0011 2019-04-26 [S] [expires: 2021-04-
   ↪16]
   AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBB
   uid                     [ultimate] YOU <$YOUR_EMAIL>
   ```

   The key's ID in this case would be `AABBCCDDEEFF0011`. Note that your signkey will have only a `[S]` after the creation date, not a `[SC]` (then you are looking at your ordinary GPG key that can also encrypt).

[2] Configure your editor to automatically break lines and/or reformat paragraphs. For Emacs you can use `M-x set-fill-column RET 75` and `M-x auto-fill-mode RET` for automatic filling of paragraphs in conjunction with `M-x fill-paragraph` (usually bound to `M-q`) to reformat a paragraph to adhere to the current column width. For editing reStructuredText we recommend `rst-mode` (built-in to Emacs since version `23.1`). Vim users can set the text width via `:tw 75` and then use the commands `gwip` or `gq`.

12.12.2 Bumping the Version

The KIWI NG project follows the Semantic Versioning scheme. We use the **bumpversion** tool for consistent versioning.

Follow these instructions to bump the major, minor, or patch part of the KIWI NG version. Ensure that your repository is clean (i.e. no modified and unknown files exist) beforehand running **bumpversion**.

- For backwards-compatible bug fixes:

  ```
  $ bumpversion patch
  ```

- For additional functionality in a backwards-compatible manner. When changed, the patch level is reset to zero:

  ```
  $ bumpversion minor
  ```

- For incompatible API changes. When changed, the patch and minor levels are reset to zero:

  ```
  $ bumpversion major
  ```

12.12.3 Creating a RPM Package

We provide a template for a RPM spec file in `package/python-kiwi-spec-template` alongside with a rpmlint configuration file and an automatically updated `python-kiwi.changes`.

To create the necessary files to build a RPM package via `rpmbuild`, run:

```
$ make build
```

The sources are collected in the `dist/` directory. These can be directly build it with **rpmbuild**, **fedpkg**, or submitted to the Open Build Service using **osc**.

CHAPTER

THIRTEEN

PYTHON API

Note: This API documentation covers KIWI NG 9.23.20

13.1 kiwi Package

13.1.1 Submodules

13.1.2 `kiwi.app` Module

class `kiwi.app.`**`App`**

Bases: `object`

Implements creation of task instances

Each task class implements a process method which is called when constructing an instance of App

13.1.3 `kiwi.cli` Module

class `kiwi.cli.`**`Cli`**

Bases: `object`

Implements the main command line interface

An instance of the Cli class builds the entry point for the application and implements methods to load further command plugins which itself provides their own command line interface

`get_command``()`

Extract selected command name

Returns command name

Return type str

get_command_args()
Extract argument dict for selected command

Returns

Contains dictionary of command arguments

```
{
    '--command-option': 'value'
}
```

Return type dict

get_global_args()
Extract argument dict for global arguments

Returns

Contains dictionary of global arguments

```
{
    '--global-option': 'value'
}
```

Return type dict

get_servicename()
Extract service name from argument parse result

Returns service name

Return type str

invoke_kiwicompat(*compat_args*)
Execute kiwicompat with provided legacy KIWI command line arguments

Example:

```
invoke_kiwicompat(
    '--build', 'description', '--type', 'vmx',
    '-d', 'destination'
)
```

Parameters **compat_args** (*list*) – legacy kiwi command arguments

load_command()
Loads task class plugin according to service and command name

Returns loaded task module

Return type object

show_and_exit_on_help_request()
Execute man to show the selected manual page

13.1.4 `kiwi.command` Module

class kiwi.command.**Command**

Bases: `object`

Implements command invocation

An instance of Command provides methods to invoke external commands in blocking and non blocking mode. Control of stdout and stderr is given to the caller

static call(*command*, *custom_env=None*)

Execute a program and return an io file handle pair back. stdout and stderr are both on different channels. The caller must read from the output file handles in order to actually run the command. This can be done using the CommandIterator from command_process

Example:

```
process = Command.call(['ls', '-l'])
```

Parameters

- **command** (*list*) – command and arguments
- **custom_env** (*list*) – custom os.environ

Returns

Contains process results in command type

```
command(
    output='string', output_available=bool,
    error='string', error_available=bool,
    process=subprocess
)
```

Return type namedtuple

static run(*command*, *custom_env=None*, *raise_on_error=True*, *stderr_to_stdout=False*)

Execute a program and block the caller. The return value is a hash containing the stdout, stderr and return code information. Unless raise_on_error is set to false an exception is thrown if the command exits with an error code not equal to zero

Example:

```
result = Command.run(['ls', '-l'])
```

Parameters

- **command** (*list*) – command and arguments
- **custom_env** (*list*) – custom os.environ

- **raise_on_error** (*bool*) – control error behaviour
- **stderr_to_stdout** (*bool*) – redirects stderr to stdout

Returns

Contains call results in command type

```
command(output='string', error='string',␣
↪returncode=int)
```

Return type namedtuple

class kiwi.command.**command_type**(*output*, *error*, *returncode*)

Bases: tuple

property error
Alias for field number 1

property output
Alias for field number 0

property returncode
Alias for field number 2

13.1.5 kiwi.command_process Module

class kiwi.command_process.**CommandIterator**(*command*)

Bases: object

Implements an Iterator for Instances of Command

Parameters **command** (*subprocess*) – instance of subprocess

get_error_code()
Provide return value from processed command

Returns errorcode

Return type int

get_error_output()
Provide data which was sent to the stderr channel

Returns stderr data

Return type str

get_pid()
Provide process ID of command while running

Returns pid

Return type int

kill()
Send kill signal SIGTERM to command process

class kiwi.command_process.**CommandProcess**(*command*, *log_topic='system'*)

Bases: `object`

Implements processing of non blocking Command calls

Provides methods to iterate over non blocking instances of the Command class with and without progress information

Parameters

- **command** (*subprocess*) – instance of subprocess
- **log_topic** (*string*) – topic string for logging

create_match_method(*method*)

create a matcher function pointer which calls the given method as method(item_to_match, data) on dereference

Parameters **method** (*function*) – function reference

Returns function pointer

Return type object

poll()

Iterate over process, raise on error and log output

poll_and_watch()

Iterate over process don't raise on error and log stdout and stderr

poll_show_progress(*items_to_complete*, *match_method*)

Iterate over process and show progress in percent raise on error and log output

Parameters

- **items_to_complete** (*list*) – all items
- **match_method** (*function*) – method matching item

returncode()

13.1.6 `kiwi.defaults` Module

class kiwi.defaults.**Defaults**

Bases: `object`

Implements default values

Provides static methods for default values and state information

get(*key*)

Implements get method for profile elements

Parameters **key** (*string*) – profile keyname

Returns key value

Return type str

static get_archive_image_types()
Provides list of supported archive image types

Returns archive names

Return type list

static get_bios_image_name()
Provides bios core boot binary name

Returns name

Return type str

static get_bios_module_directory_name()
Provides x86 BIOS directory name which stores the pc binaries

Returns directory name

Return type str

static get_boot_image_description_path()
Provides the path to find custom kiwi boot descriptions

Returns directory path

Return type str

static get_boot_image_strip_file()
Provides the file path to bootloader strip metadata. This file contains information about the files and directories automatically striped out from the kiwi initrd

Returns file path

Return type str

static get_buildservice_env_name()
Provides the base name of the environment file in a buildservice worker

Returns file basename

Return type str

static get_common_functions_file()
Provides the file path to config functions metadata.

This file contains bash functions used for system configuration or in the boot code from the kiwi initrd

Returns file path

Return type str

static get_container_base_image_tag()
Provides the tag used to identify base layers during the build of derived images.

Returns tag

Return type str

static get_container_compression()
Provides default container compression algorithm

Returns name

Return type str

static get_container_image_types()
Provides list of supported container image types

Returns container names

Return type list

static get_custom_rpm_bootstrap_macro_name()
Returns the rpm bootstrap macro file name created in the custom rpm macros path

Returns filename

Return type str

static get_custom_rpm_image_macro_name()
Returns the rpm image macro file name created in the custom rpm macros path

Returns filename

Return type str

static get_custom_rpm_macros_path()
Returns the custom macros directory for the rpm database.

Returns path name

Return type str

static get_default_boot_mbytes()
Provides default boot partition size in mbytes

Returns mbsize value

Return type int

static get_default_boot_timeout_seconds()
Provides default boot timeout in seconds

Returns seconds

Return type int

static get_default_bootloader()
Return default bootloader name which is grub2

Returns bootloader name

Return type str

static get_default_container_created_by()
Provides the default 'created by' history entry for containers.

Returns the specific kiwi version used for the build

Return type str

static get_default_container_name()
Provides the default container name.

Returns name

Return type str

static get_default_container_subcommand()
Provides the default container subcommand.

Returns command as a list of arguments

Return type list

static get_default_container_tag()
Provides the default container tag.

Returns tag

Return type str

static get_default_disk_start_sector()
Provides the default initial disk sector for the first disk partition.

Returns sector value

Return type int

static get_default_efi_boot_mbytes()
Provides default EFI partition size in mbytes

Returns mbsize value

Return type int

static get_default_efi_partition_table_type()
Provides the default partition table type for efi firmwares.

Returns partition table type name

Return type str

static get_default_firmware(*arch*)
Provides default firmware for specified architecture

Parameters **arch** (*string*) – machine architecture name

Returns firmware name

Return type str

static get_default_inode_size()
Provides default size of inodes in bytes. This is only relevant for inode based filesystems

Returns bytesize value

Return type int

static get_default_legacy_bios_mbytes()
Provides default size of bios_grub partition in mbytes

Returns mbsize value

Return type int

static get_default_live_iso_root_filesystem()
Provides default live iso root filesystem type

Returns filesystem name

Return type str

static get_default_live_iso_type()
Provides default live iso union type

Returns live iso type

Return type str

static get_default_package_manager() → str
Returns the default package manager name if none is configured in the image description

Returns package manager name

Return type str

static get_default_packager_tool(*package_manager*)
Provides the packager tool according to the package manager

Parameters **package_manager** (*string*) – package manger name

Returns packager tool binary name

Return type str

static get_default_prep_mbytes()
Provides default size of prep partition in mbytes

Returns mbsize value

Return type int

static get_default_uri_type()
Provides default URI type

Absolute path specifications used in the context of an URI will apply the specified default mime type

Returns URI mime type

Return type str

static get_default_video_mode()
Uses auto mode for default video. See get_video_mode_map for details on the value depending which bootloader is used

Returns auto

Return type str

static get_default_volume_group_name()
Provides default LVM volume group name

Returns name

Return type str

static get_disk_format_types()
Provides supported disk format types

Returns disk types

Return type list

static get_disk_image_types()
Provides supported disk image types

Returns disk image type names

Return type list

static get_dracut_conf_name()
Provides file path of dracut config file to be used with KIWI

Returns file path name

Return type str

static get_ec2_capable_firmware_names()
Provides list of EC2 capable firmware names. These are those for which kiwi supports the creation of disk images bootable within the Amazon EC2 public cloud

Returns firmware names

Return type list

static get_efi_capable_firmware_names()
Provides list of EFI capable firmware names. These are those for which kiwi supports the creation of an EFI bootable disk image

Returns firmware names

Return type list

static get_efi_image_name(*arch*)
Provides architecture specific EFI boot binary name

Parameters **arch** (*string*) – machine architecture name

Returns name

Return type str

static get_efi_module_directory_name(*arch*)
Provides architecture specific EFI directory name which stores the EFI binaries for the desired architecture.

Parameters **arch** (*string*) – machine architecture name

Returns directory name

Return type str

static get_efi_vendor_directory(*efi_path*)

Provides EFI vendor directory if present

Looks up distribution specific EFI vendor directory

Parameters **root_path** (*string*) – path to efi mountpoint

Returns directory path or None

Return type str

static get_exclude_list_for_non_physical_devices()

Provides the list of folders that are not associated with a physical device. KIWI returns the basename of the folders typically used as mountpoint for those devices.

Returns list of file and directory names

Return type list

static get_exclude_list_for_root_data_sync()

Provides the list of files or folders that are created by KIWI for its own purposes. Those files should be not be included in the resulting image.

Returns list of file and directory names

Return type list

static get_failsafe_kernel_options()

Provides failsafe boot kernel options

Returns

list of kernel options

```
['option=value', 'option']
```

Return type list

static get_filesystem_image_types()

Provides list of supported filesystem image types

Returns filesystem names

Return type list

static get_firmware_types()

Provides supported architecture specific firmware types

Returns firmware types per architecture

Return type dict

static get_grub_basic_modules(*multiboot*)

Provides list of basic grub modules

Parameters **multiboot** (*bool*) – grub multiboot mode

Returns list of module names

Return type list

static get_grub_bios_core_loader(*root_path*)
Provides grub bios image

Searches distribution specific locations to find the core bios image below the given root path

Parameters **root_path** (*string*) – image root path

Returns file path or None

Return type str

static get_grub_bios_modules(*multiboot=False*)
Provides list of grub bios modules

Parameters **multiboot** (*bool*) – grub multiboot mode

Returns list of module names

Return type list

static get_grub_boot_directory_name(*lookup_path*)
Provides grub2 data directory name in boot/ directory

Depending on the distribution the grub2 boot path could be either boot/grub2 or boot/grub. The method will decide for the correct base directory name according to the name pattern of the installed grub2 tools

Returns directory basename

Return type str

static get_grub_efi_font_directory(*root_path*)
Provides distribution specific EFI font directory used with grub.

Parameters **root_path** (*string*) – image root path

Returns file path or None

Return type str

static get_grub_efi_modules(*multiboot=False*)
Provides list of grub efi modules

Parameters **multiboot** (*bool*) – grub multiboot mode

Returns list of module names

Return type list

static get_grub_ofw_modules()
Provides list of grub ofw modules (ppc)

Returns list of module names

Return type list

static get_grub_path(*root_path*, *filename*, *raise_on_error=True*)

Provides grub path to given search file

Depending on the distribution grub could be installed below a grub2 or grub directory. grub could also reside in /usr/lib as well as in /usr/share. Therefore this information needs to be dynamically looked up

Parameters

- **root_path** (*string*) – root path to start the lookup from
- **filename** (*string*) – filename to search
- **raise_on_error** (*bool*) – raise on not found, defaults to True

The method returns the path to the given grub search file. By default it raises a KiwiBootLoaderGrubDataError exception if the file could not be found in any of the search locations. If raise_on_error is set to False and no file could be found the function returns None

Returns filepath

Return type str

static get_grub_s390_modules()

Provides list of grub ofw modules (s390)

Returns list of module names

Return type list

static get_imported_root_image(*root_dir*)

Provides the path to an imported root system image

If the image description specified a derived_from attribute the file from this attribute is copied into the root_dir using the name as provided by this method

Parameters **root_dir** (*string*) – image root directory

Returns file path name

Return type str

static get_install_volume_id()

Provides default value for ISO volume ID for install media

Returns name

Return type str

static get_iso_boot_path()

Provides arch specific relative path to boot files on kiwi iso filesystems

Returns relative path name

Return type str

static get_iso_tool_category()
Provides default iso tool category

Returns name

Return type str

static get_isolinux_bios_grub_loader()
Return name of eltorito grub image used as isolinux loader in BIOS mode if isolinux.bin should not be used

Returns file base name

Return type str

static get_kis_image_types()
Provides supported kis image types

Returns kis image type names

Return type list

static get_live_dracut_modules_from_flag(*flag_name*)
Provides flag_name to dracut modules name map

Depending on the value of the flag attribute in the KIWI image description specific dracut modules need to be selected

Returns dracut module names as list

Return type list

static get_live_image_types()
Provides supported live image types

Returns live image type names

Return type list

static get_live_iso_persistent_boot_options(*persistent_filesystem=None*)
Provides list of boot options passed to the dracut kiwi-live module to setup persistent writing

Returns list of boot options

Return type list

static get_luks_key_length()
Provides key length to use for random luks keys

static get_lvm_overhead_mbytes()
Provides empiric LVM overhead size in mbytes

Returns mbsize value

Return type int

static get_min_partition_mbytes()
Provides default minimum partition size in mbytes

Returns mbsize value

Return type int

static get_min_volume_mbytes()
Provides default minimum LVM volume size in mbytes

Returns mbsize value

Return type int

static get_obs_api_server_url()
Provides the default API server url to access the public open buildservice API

Returns url path

Return type str

static get_obs_download_server_url()
Provides the default download server url hosting the public open buildservice repositories

Returns url path

Return type str

static get_oci_archive_tool()
Provides the default OCI archive tool name.

Returns name

Return type str

static get_platform_name()

static get_preparer()
Provides ISO preparer name

Returns name

Return type str

static get_profile_file(*root_dir*)
Return name of profile file for given root directory

static get_publisher()
Provides ISO publisher name

Returns name

Return type str

static get_recovery_spare_mbytes()
Provides spare size of recovery partition in mbytes

Returns mbsize value

Return type int

static get_schema_file()

Provides file path to kiwi RelaxNG schema

Returns file path

Return type str

static get_shared_cache_location()

Provides the shared cache location

This is a directory which shares data from the image buildsystem host with the image root system. The location is returned as an absolute path stripped off by the leading '/'. This is because the path is transparently used on the host /<cache-dir> and inside of the image imageroot/<cache-dir>

Returns directory path

Return type str

static get_shim_loader(*root_path*)

Provides shim loader file path

Searches distribution specific locations to find shim.efi below the given root path

Parameters **root_path** (*string*) – image root path

Returns file path or None

Return type str

static get_shim_vendor_directory(*root_path*)

Provides shim vendor directory

Searches distribution specific locations to find shim.efi below the given root path and return the directory name to the file found

Parameters **root_path** (*string*) – image root path

Returns directory path or None

Return type str

static get_signed_grub_loader(*root_path*)

Provides shim signed grub loader file path

Searches distribution specific locations to find grub.efi below the given root path

Parameters **root_path** (*string*) – image root path

Returns file path or None

Return type str

static get_snapper_config_template_file()

Provides the default configuration template file for snapper

Returns file

Return type str

static get_solvable_location()
Provides the directory to store SAT solvables for repositories. The solvable files are used to perform package dependency and metadata resolution

Returns directory path

Return type str

static get_swapsize_mbytes()
Provides swapsize in MB

static get_sync_options()
Provides list of default data sync options

Returns list of rsync options

Return type list

static get_syslinux_modules()
Returns list of syslinux modules to include on ISO images that boots via isolinux

Returns base file names

Return type list

static get_syslinux_search_paths()
syslinux is packaged differently between distributions. This method returns a list of directories to search for syslinux data

Returns directory names

Return type list

static get_unsigned_grub_loader(*root_path*)
Provides unsigned grub efi loader file path

Searches distribution specific locations to find grub.efi below the given root path

Parameters **root_path** (`string`) – image root path

Returns file path or None

Return type str

static get_vagrant_config_virtualbox_guest_additions()
Provides the default value for `vagrantconfig.virtualbox_guest_additions_present`

Returns whether guest additions are expected to be present in the vagrant box

Return type bool

static get_vendor_grubenv(*efi_path*)

static get_video_mode_map()
Provides video mode map

Assign a tuple to each kernel vesa hex id for each of the supported bootloaders

Returns

video type map

```
{'kernel_hex_mode': video_type(grub2='mode',␣
↪isolinux='mode')}
```

Return type dict

static get_volume_id()

Provides default value for ISO volume ID

Returns name

Return type str

static get_xsl_stylesheet_file()

Provides the file path to the KIWI XSLT style sheets

Returns file path

Return type str

static get_xz_compression_options()

Provides compression options for the xz compressor

Returns

Contains list of options

```
['--option=value']
```

Return type list

static is_buildservice_worker()

Checks if build host is an open buildservice machine

The presence of /.buildenv on the build host indicates we are building inside of the open buildservice

Returns True if obs worker, else False

Return type bool

static is_x86_arch(*arch*)

Checks if machine architecture is x86 based

Any arch that matches 32bit and 64bit x86 architecture causes the method to return True. Anything else will cause the method to return False

Return type bool

static project_file(*filename*)

Provides the python module base directory search path

The method uses the resource_filename method to identify files and directories from the application

Parameters **filename** (*string*) – relative project file

Returns absolute file path name

Return type str

static set_custom_runtime_config_file(*filename*)
Sets the runtime config file once

Parameters **filename** (*str*) – a file path name

static set_shared_cache_location(*location*)
Sets the shared cache location once

Parameters **location** (*str*) – a location path

to_profile(*profile*)
Implements method to add list of profile keys and their values to the specified instance of a Profile class

Parameters **profile** (*object*) – Profile instance

13.1.7 `kiwi.exceptions` Module

exception kiwi.exceptions.**KiwiAnyMarkupPluginError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the python anymarkup module failed to load.

exception kiwi.exceptions.**KiwiArchiveSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an unsupported image archive type is used.

exception kiwi.exceptions.**KiwiArchiveTarError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if impossible to determine which tar command version is installed on the underlying system

exception kiwi.exceptions.**KiwiBootImageDumpError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an instance of BootImage* can not be serialized on as file via pickle dump

exception kiwi.exceptions.**KiwiBootImageSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an unsupported initrd system type is used.

exception kiwi.exceptions.**KiwiBootLoaderConfigSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a configuration for an unsupported bootloader is requested.

exception kiwi.exceptions.**KiwiBootLoaderGrubDataError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if no grub installation was found.

exception kiwi.exceptions.**KiwiBootLoaderGrubFontError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if no grub unicode font was found.

exception kiwi.exceptions.**KiwiBootLoaderGrubInstallError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if grub install to master boot record has failed.

exception kiwi.exceptions.**KiwiBootLoaderGrubModulesError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the synchronisation of modules from the grub installation to the boot space has failed.

exception kiwi.exceptions.**KiwiBootLoaderGrubPlatformError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to use grub on an unsupported platform.

exception kiwi.exceptions.**KiwiBootLoaderGrubSecureBootError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the Microsoft signed shim loader or grub2 loader could not be found in the image root system

exception kiwi.exceptions.**KiwiBootLoaderInstallSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an installation for an unsupported bootloader is requested.

exception kiwi.exceptions.**KiwiBootLoaderTargetError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the target to read the bootloader path from is not a disk or an iso image.

exception kiwi.exceptions.**KiwiBootLoaderZiplInstallError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the installation of zipl has failed.

exception kiwi.exceptions.**KiwiBootLoaderZiplPlatformError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a configuration for an unsupported zipl architecture is requested.

exception kiwi.exceptions.**KiwiBootLoaderZiplSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the data set to configure the zipl bootloader is incomplete.

exception kiwi.exceptions.**KiwiBootStrapPhaseFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the bootstrap phase of the system prepare command has failed.

exception kiwi.exceptions.**KiwiBuildahError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised on inconsistent buildah class calls

exception kiwi.exceptions.**KiwiBundleError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the system bundle command has failed.

exception kiwi.exceptions.**KiwiCommandCapabilitiesError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception is raised when some the CommandCapabilities methods fails, usually meaning there is some issue trying to parse some command output.

exception kiwi.exceptions.**KiwiCommandError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an external command called via a Command instance has returned with an exit code != 0 or could not be called at all.

exception kiwi.exceptions.**KiwiCommandNotFound**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if any executable command cannot be found in the evironment PATH variable.

exception kiwi.exceptions.**KiwiCommandNotLoaded**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a kiwi command task module could not be loaded.

exception kiwi.exceptions.**KiwiCompatError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the given kiwi compatibility command line could not be understood by the compat option parser.

exception kiwi.exceptions.**KiwiCompressionFormatUnknown**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the compression format of the data could not be detected.

exception kiwi.exceptions.**KiwiConfigFileFormatNotSupported**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if kiwi description file format is not supported.

exception kiwi.exceptions.**KiwiConfigFileNotFound**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if no kiwi XML description was found.

exception kiwi.exceptions.**KiwiContainerBuilderError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception is raised when something fails during a container image build procedure.

exception kiwi.exceptions.**KiwiContainerImageSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt to create a container instance for an unsupported container type is performed.

exception kiwi.exceptions.**KiwiContainerSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an error in the creation of the container archive happened.

exception kiwi.exceptions.**KiwiDataStructureError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the XML description failed to parse the data structure.

exception kiwi.exceptions.**KiwiDebootstrapError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if not enough user data to call debootstrap were provided or the debootstrap has failed.

exception kiwi.exceptions.**KiwiDecodingError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception is raised on decoding literals failure

exception kiwi.exceptions.**KiwiDescriptionInvalid**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the XML description failed to validate the XML schema.

exception kiwi.exceptions.**KiwiDeviceProviderError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a storage provide is asked for its managed device but no such device exists.

exception kiwi.exceptions.**KiwiDiskBootImageError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a kiwi boot image does not provide the requested data, e.g kernel, or hypervisor files.

exception kiwi.exceptions.**KiwiDiskFormatSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to create a disk format instance of an unsupported disk format.

exception kiwi.exceptions.**KiwiDiskGeometryError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the disk geometry (partition table) could not be read or evaluated against their expected geometry and capabilities.

exception kiwi.exceptions.**KiwiDistributionNameError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the distribution name could not be found. The information is extracted from the boot attribute of the XML description. If no boot attribute is present or does not match the naming conventions the exception is raised.

exception `kiwi.exceptions.`**`KiwiError`**(*message*)
Bases: `Exception`

Base class to handle all known exceptions

Specific exceptions are implemented as sub classes of KiwiError

Attributes

Parameters **`message`** (*`string`*) – Exception message text

exception `kiwi.exceptions.`**`KiwiExtensionError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an extension section of the same namespace is used multiple times as toplevel section within the extension section. Each extension must have a single toplevel entry point qualified by its namespace

exception `kiwi.exceptions.`**`KiwiFileAccessError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if accessing a file or its metadata failed

exception `kiwi.exceptions.`**`KiwiFileNotFound`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the requested file could not be found.

exception `kiwi.exceptions.`**`KiwiFileSystemSetupError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an attempt was made to build an unsupported or unspecified filesystem.

exception `kiwi.exceptions.`**`KiwiFileSystemSyncError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the data sync from the system into the loop mounted filesystem image failed.

exception `kiwi.exceptions.`**`KiwiFormatSetupError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the requested disk format could not be created.

exception `kiwi.exceptions.`**`KiwiHelpNoCommandGiven`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the request for the help page is executed without a command to show the help for.

exception `kiwi.exceptions.`**`KiwiImageResizeError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the request to resize a disk image failed. Reasons could be a missing raw disk reference or a wrong size specification.

exception `kiwi.exceptions.`**`KiwiImportDescriptionError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the XML description data and scripts could not be imported into the root of the image.

exception `kiwi.exceptions.`**`KiwiInstallBootImageError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the required files to boot an installation image could not be found, e.g kernel or hypervisor.

exception `kiwi.exceptions.`**`KiwiInstallMediaError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if a request for an installation media is made but the system image type is not an oem type.

exception `kiwi.exceptions.`**`KiwiInstallPhaseFailed`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the install phase of a system prepare command has failed.

exception `kiwi.exceptions.`**`KiwiIsoLoaderError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if no isolinux loader file could be found.

exception `kiwi.exceptions.`**`KiwiIsoMetaDataError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an inconsistency in the ISO header was found such like invalid eltorito specification or a broken path table.

exception `kiwi.exceptions.`**`KiwiIsoToolError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an iso helper tool such as isoinfo could not be found on the build system.

exception `kiwi.exceptions.`**`KiwiKernelLookupError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the search for the kernel image file failed

exception `kiwi.exceptions.`**`KiwiKisBootImageError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if a required boot file e.g the kernel could not be found in the process of building a kis image.

exception `kiwi.exceptions.`**`KiwiLiveBootImageError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an attempt was made to use an unsupported live iso type.

exception kiwi.exceptions.**KiwiLoadCommandUndefined**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if no command is specified for a given service on the commandline.

exception kiwi.exceptions.**KiwiLogFileSetupFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the log file could not be created.

exception kiwi.exceptions.**KiwiLoopSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if not enough user data to create a loop device is specified.

exception kiwi.exceptions.**KiwiLuksSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if not enough user data is provided to setup the luks encryption on the given device.

exception kiwi.exceptions.**KiwiMappedDeviceError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the device to become mapped does not exist.

exception kiwi.exceptions.**KiwiMarkupConversionError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the markup format conversion is not possible.

exception kiwi.exceptions.**KiwiMountKernelFileSystemsError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a kernel filesystem such as proc or sys could not be mounted.

exception kiwi.exceptions.**KiwiMountSharedDirectoryError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the host <-> image shared directory could not be mounted.

exception kiwi.exceptions.**KiwiNotImplementedError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a functionality is not yet implemented.

exception kiwi.exceptions.**KiwiOCIArchiveToolError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the requested OCI archive tool is not supported

exception kiwi.exceptions.**KiwiPackageManagerSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to create a package manager instance for an unsupported package manager.

exception kiwi.exceptions.**KiwiPackagesDeletePhaseFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the packages deletion phase in system prepare fails.

exception kiwi.exceptions.**KiwiPartitionerGptFlagError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to set an unknown partition flag for an entry in the GPT table.

exception kiwi.exceptions.**KiwiPartitionerMsDosFlagError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to set an unknown partition flag for an entry in the MSDOS table.

exception kiwi.exceptions.**KiwiPartitionerSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to create an instance of a partitioner for an unsupporte partitioner.

exception kiwi.exceptions.**KiwiPrivilegesError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if root privileges are required but not granted.

exception kiwi.exceptions.**KiwiProfileNotFound**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a specified profile does not exist in the XML configuration.

exception kiwi.exceptions.**KiwiRaidSetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if invalid or not enough user data is provided to create a raid array on the specified storage device.

exception kiwi.exceptions.**KiwiRepositorySetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to create an instance of a repository for an unsupported package manager.

exception kiwi.exceptions.**KiwiRequestError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a package request could not be processed by the corresponding package manager instance.

exception kiwi.exceptions.**KiwiRequestedTypeError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to build an image for an unsupported image type.

exception kiwi.exceptions.**KiwiResizeRawDiskError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if an attempt was made to resize the image disk to a smaller size than the current one. Simply shrinking a disk image file is not possible without data corruption because the partitions were setup to use the entire disk geometry as it fits into the file. A successful shrinking operation would require the filesystems and the partition table to be reduced which is not done by the provided simple storage resize method. In addition without the user overwriting the disk size in the XML setup, kiwi will calculate the minimum required size in order to store the data. Thus in almost all cases it will not be possible to store the data in a smaller disk.

exception `kiwi.exceptions.`**`KiwiResultError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the image build result pickle information could not be created or loaded.

exception `kiwi.exceptions.`**`KiwiRootDirExists`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the specified image root directory already exists and should not be re-used.

exception `kiwi.exceptions.`**`KiwiRootImportError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception is raised when something fails during the root import procedure.

exception `kiwi.exceptions.`**`KiwiRootInitCreationError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the initialization of a new image root directory has failed.

exception `kiwi.exceptions.`**`KiwiRpmDirNotRemoteError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the provided rpm-dir repository is not local

exception `kiwi.exceptions.`**`KiwiRuntimeConfigFileError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the provided custom runtime config file could not be found

exception `kiwi.exceptions.`**`KiwiRuntimeConfigFormatError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the expected format in the yaml KIWI runtime config file does not match

exception `kiwi.exceptions.`**`KiwiRuntimeError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if a runtime check has failed.

exception `kiwi.exceptions.`**`KiwiSatSolverJobError`**`(`*message*`)`
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if a sat solver job can not be done, e.g because the requested package or collection does not exist in the registered repository metadata

exception kiwi.exceptions.**KiwiSatSolverJobProblems**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the sat solver operations returned with solver problems e.g package conflicts

exception kiwi.exceptions.**KiwiSatSolverPluginError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the python solv module failed to load. The solv module is provided by SUSE's rpm package python-solv and provides a python binding to the libsolv C library

exception kiwi.exceptions.**KiwiSchemaImportError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the schema file could not be read by lxml.RelaxNG.

exception kiwi.exceptions.**KiwiScriptFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a user script returned with an exit code != 0.

exception kiwi.exceptions.**KiwiSetupIntermediateConfigError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the setup of the temporary image system configuration for the duration of the build process has failed.

exception kiwi.exceptions.**KiwiShellVariableValueError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if a given python value cannot be converted into a string representation for use in shell scripts

exception kiwi.exceptions.**KiwiSizeError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception is raised when the convertion from a given size in string format to a number.

exception kiwi.exceptions.**KiwiSolverRepositorySetupError**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the repository type is not supported for the creation of a SAT solvable

exception kiwi.exceptions.**KiwiSystemDeletePackagesFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the deletion of a package has failed in the corresponding package manager instance.

exception kiwi.exceptions.**KiwiSystemInstallPackagesFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the installation of a package has failed in the corresponding package manager instance.

exception kiwi.exceptions.**KiwiSystemUpdateFailed**(*message*)
Bases: *kiwi.exceptions.KiwiError*

Exception raised if the package upgrade has failed in the corresponding package manager instance.

exception `kiwi.exceptions.`**`KiwiTargetDirectoryNotFound`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the specified target directory to store the image results was not found.

exception `kiwi.exceptions.`**`KiwiTemplateError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the substitution of variables in a configuration file template has failed.

exception `kiwi.exceptions.`**`KiwiTypeNotFound`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if no build type was found in the XML description.

exception `kiwi.exceptions.`**`KiwiUnknownServiceName`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an unknown service name was provided on the commandline.

exception `kiwi.exceptions.`**`KiwiUriOpenError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the urllib urlopen request has failed

exception `kiwi.exceptions.`**`KiwiUriStyleUnknown`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if an unsupported URI style was used in the source definition of a repository.

exception `kiwi.exceptions.`**`KiwiUriTypeUnknown`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the protocol type of an URI is unknown in the source definition of a repository.

exception `kiwi.exceptions.`**`KiwiValidationError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the XML validation against the schema has failed.

exception `kiwi.exceptions.`**`KiwiVhdTagError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the GUID tag is not provided in the expected format.

exception `kiwi.exceptions.`**`KiwiVolumeGroupConflict`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the requested LVM volume group already is in use on the build system.

exception `kiwi.exceptions.`**`KiwiVolumeManagerSetupError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the preconditions for volume mangement support are not met or an attempt was made to create an instance of a volume manager for an unsupported volume management system.

exception `kiwi.exceptions.`**`KiwiVolumeRootIDError`**(*message*)
Bases: *`kiwi.exceptions.KiwiError`*

Exception raised if the root volume can not be found. This concept currently exists only for the btrfs subvolume system.

13.1.8 `kiwi.firmware` Module

class `kiwi.firmware.`**`FirmWare`**(*xml_state*)
Bases: `object`

Implements firmware specific methods

According to the selected firmware some parameters in a disk image changes. This class provides methods to provide firmware dependant information

- **param object xml_state** instance of `XMLState`

`bios_mode`()
Check if BIOS mode is requested

Returns True or False

Return type bool

`ec2_mode`()
Check if EC2 mode is requested

Returns True or False

Return type bool

`efi_mode`()
Check if EFI mode is requested

Returns The requested EFI mode or None if no EFI mode requested

Return type str

`get_efi_partition_size`()
Size of EFI partition. Returns 0 if no such partition is needed

Returns mbsize value

Return type int

`get_legacy_bios_partition_size`()
Size of legacy bios_grub partition if legacy BIOS mode is required. Returns 0 if no such partition is needed

Returns mbsize value

Return type int

get_partition_table_type()

Provides partition table type according to architecture and firmware

Returns partition table name

Return type str

get_prep_partition_size()

Size of Prep partition if OFW mode is requested. Returns 0 if no such partition is needed

Returns mbsize value

Return type int

legacy_bios_mode()

Check if the legacy boot from BIOS systems should be activated

Returns True or False

Return type bool

ofw_mode()

Check if OFW mode is requested

Returns True or False

Return type bool

opal_mode()

Check if Opal mode is requested

Returns True or False

Return type bool

13.1.9 `kiwi.help` Module

class kiwi.help.**Help**

Bases: `object`

Implements man page help for kiwi commands

Each kiwi command implements their own manual page, which is shown if the positional argument 'help' is passed to the command.

show(*command=None*)

Call man to show the command specific manual page

All kiwi commands store their manual page in the section '8' of the man system. The calling process is replaced by the man process

Parameters **command** (*string*) – man page name

13.1.10 kiwi.kiwi Module

`kiwi.kiwi.`**`extras`**`(`*help_*, *version*, *options*, *doc*`)`

Overwritten method from docopt

Shows our own usage message for -h|–help

Parameters

- **help** (*bool*) – indicate to show help
- **version** (*string*) – version string
- **options** (*list*) –

list of option tuples

```
[option(name='name', value='value')]
```

Parameters **doc** (*string*) – docopt doc string

`kiwi.kiwi.`**`main`**`()`

kiwi - main application entry point

Initializes a global log object and handles all errors of the application. Every known error is inherited from KiwiError, everything else is passed down until the generic Exception which is handled as unexpected error including the python backtrace

`kiwi.kiwi.`**`usage`**`(`*command_usage*`)`

Instead of the docopt way to show the usage information we provide a kiwi specific usage information. The usage data now always consists out of:

1. the generic call kiwi-ng [global options] service <command> [<args>]
2. the command specific usage defined by the docopt string short form by default, long form with -h | –help
3. the global options

Parameters **command_usage** (*string*) – usage data

13.1.11 kiwi.logger Module

class `kiwi.logger.`**`Logger`**`(`*name: str*`)`

Bases: `logging.Logger`

Extended logging facility based on Python logging

Parameters **name** (*str*) – name of the logger

`getLogLevel``()` → int

Return currently used log level

Returns log level number

Return type int

`get_logfile``()` → Optional[str]

Return file path name of logfile

Returns file path

Return type str

`static progress``(`*current: int*, *total: int*, *prefix: str*, *bar_length: int = 40*`)` → None

Custom progress log information. progress information is intentionally only logged to stdout and will bypass any handlers. We don't want this information to show up in the log file

Parameters

- **`current`** (*`int`*) – current item
- **`total`** (*`int`*) – total number of items
- **`prefix`** (*`string`*) – prefix name
- **`bar_length`** (*`int`*) – length of progress bar

`setLogLevel``(`*level: int*`)` → None

Set custom log level for all console handlers

Parameters **`level`** (*`int`*) – log level number

`set_color_format``()` → None

Set color format for all console handlers

`set_logfile``(`*filename: str*`)` → None

Set logfile handler

Parameters **`filename`** (*`str`*) – logfile file path

13.1.12 `kiwi.logger_color_formatter` Module

`class` `kiwi.logger_color_formatter.`**`ColorFormatter`**`(`*fmt=None*, *datefmt=None*, *style='%'*`)`

Bases: `logging.Formatter`

Extended standard logging Formatter

Extended format supporting text with color metadata

Example:

```
ColorFormatter(message_format, '%H:%M:%S')
```

`format``(`*record: logging.LogRecord*`)` → str

Creates a logging Formatter with support for color messages

Parameters **`record`** (*`tuple`*) – logging message record

Returns result from format_message

Return type str

class kiwi.logger_color_formatter.**ColorMessage**

Bases: object

Implements color messages for Python logging facility

Has to implement the format_message method to serve as message formatter

format_message(*level: str*, *message: str*) → str

Message formatter with support for embedded color sequences

The Message is allowed to contain the following color metadata:

- $RESET, reset to no color mode
- $BOLD, bold
- $COLOR, color the following text
- $LIGHTCOLOR, light color the following text

The color of the message depends on the level and is defined in the ColorMessage constructor

Parameters

- **level** (*str*) – color level name
- **message** (*str*) – text

Returns color message with escape sequences

Return type str

13.1.13 kiwi.logger_filter Module

class kiwi.logger_filter.**DebugFilter**(*name=''*)

Bases: logging.Filter

Extended standard debug logging Filter

filter(*record: logging.LogRecord*) → bool

Only messages with record level DEBUG can pass for messages with another level an extra handler is used

Parameters **record** (*tuple*) – logging message record

Returns True|False

Return type bool

class kiwi.logger_filter.**ErrorFilter**(*name=''*)

Bases: logging.Filter

Extended standard error logging Filter

filter(*record: logging.LogRecord*) → bool
Only messages with record level DEBUG can pass for messages with another level an extra handler is used

Parameters **record** (*tuple*) – logging message record

Returns True|False

Return type bool

class kiwi.logger_filter.**InfoFilter**(*name=''*)
Bases: logging.Filter

Extended standard logging Filter

filter(*record: logging.LogRecord*) → bool
Only messages with record level INFO can pass for messages with another level an extra handler is used

Parameters **record** (*tuple*) – logging message record

Returns True|False

Return type bool

class kiwi.logger_filter.**LoggerSchedulerFilter**(*name=''*)
Bases: logging.Filter

Extended standard logging Filter

filter(*record: logging.LogRecord*) → bool
Messages from apscheduler scheduler instances are filtered out They conflict with console progress information

Parameters **record** (*tuple*) – logging message record

Returns True|False

Return type bool

class kiwi.logger_filter.**WarningFilter**(*name=''*)
Bases: logging.Filter

Extended standard warning logging Filter

filter(*record: logging.LogRecord*) → bool
Only messages with record level WARNING can pass for messages with another level an extra handler is used

Parameters **record** (*tuple*) – logging message record

Returns True|False

Return type bool

13.1.14 `kiwi.mount_manager` Module

class `kiwi.mount_manager.`**MountManager**(*device*, *mountpoint=None*)

Bases: `object`

Implements methods for mounting, umounting and mount checking

If a MountManager instance is used to mount a device the caller must care for the time when umount needs to be called. The class does not automatically release the mounted device, which is intentional

- **param string device** device node name
- **param string mountpoint** mountpoint directory name

bind_mount()

Bind mount the device to the mountpoint

is_mounted()

Check if mounted

Returns True or False

Return type bool

mount(*options=None*)

Standard mount the device to the mountpoint

Parameters **options** (`list`) – mount options

umount()

Umount by the mountpoint directory

If the resource is busy the call will return False

Returns True or False

Return type bool

umount_lazy()

Umount by the mountpoint directory in lazy mode

Release the mount in any case, however the time when the mounted resource is released by the kernel depends on the when the resource enters the non busy state

13.1.15 `kiwi.path` Module

class `kiwi.path.`**Path**

Bases: `object`

Directory path helpers

static access(*path*, *mode*, ***kwargs*)

Check whether path can be accessed with the given mode.

Parameters

- **path** (*str*) – The path that should be checked for access.
- **mode** (*int*) – Which access mode should be checked. This value must be a bit-wise or of one or more of the following constants: `os.F_OK` (note that this one is zero), `os.X_OK`, `os.R_OK` and `os.W_OK`
- **kwargs** – further keyword arguments are forwarded to `os.access()`

Returns Boolean value whether this access mode is allowed

Return type bool

Raises

- **ValueError** – if the supplied mode is invalid
- ***kiwi.exceptions.KiwiFileNotFound*** – if the path does not exist or is not accessible by the current user

static create(*path*)

Create path and all sub directories to target

Parameters path (*string*) – path name

static move_to_root(*root, elements*)

Change the given path elements to a new root directory

Parameters

- **root** (*str*) – the root path to trim
- **elements** (*list*) – list of path names

Returns changed elements

Return type list

static rebase_to_root(*root, elements*)

Include the root prefix for the given paths elements

Parameters

- **root** (*str*) – the new root path
- **elements** (*list*) – list of path names

Returns changed elements

Return type list

static remove(*path*)

Delete empty path, causes an error if target is not empty

Parameters path (*string*) – path name

static remove_hierarchy(*root, path*)

Recursively remove an empty path and its sub directories starting at a given root directory. Ignore non empty or protected paths and leave them untouched

Parameters

- **root** (*string*) – start at directory
- **path** (*string*) – path name below root

static sort_by_hierarchy(*path_list*)

Sort given list of path names by their hierachy in the tree

Example:

```
result = Path.sort_by_hierarchy(['/var/lib', '/var'])
```

Parameters **path_list** (*list*) – list of path names

Returns hierachy sorted path_list

Return type list

static which(*filename*, *alternative_lookup_paths=None*, *custom_env=None*, *access_mode=None*, *root_dir=None*)

Lookup file name in PATH

Parameters

- **filename** (*string*) – file base name
- **alternative_lookup_paths** (*list*) – list of additional lookup paths
- **custom_env** (*list*) – a custom os.environ
- **access_mode** (*int*) – one of the os access modes or a combination of them (os.R_OK, os.W_OK and os.X_OK). If the provided access mode does not match the file is considered not existing
- **root_dir** (*str*) – the root path to look at

Returns absolute path to file or None

Return type str

static wipe(*path*)

Delete path and all contents

Parameters **path** (*string*) – path name

13.1.16 `kiwi.privileges` Module

class `kiwi.privileges.`**`Privileges`**

Bases: `object`

Implements check for root privileges

static **`check_for_root_permissions`**`()`

Check if we are effectively root on the system. If not an exception is thrown

Returns True or raise an Exception

Return type bool

13.1.17 `kiwi.runtime_checker` Module

class `kiwi.runtime_checker.`**`RuntimeChecker`**`(`*xml_state*`)`

Bases: `object`

Implements build consistency checks at runtime

The schema of an image description covers structure and syntax of the provided data. The RuntimeChecker provides methods to perform further semantic checks which allows to recognize potential build or boot problems early.

- **param object xml_state** Instance of XMLState

`check_appx_naming_conventions_valid``()`

When building wsl images there are some naming conventions that must be fulfilled to run the container on Microsoft Windows

`check_architecture_supports_iso_firmware_setup``()`

For creating ISO images a different bootloader setup is performed depending on the configured firmware. If the firmware is set to bios, isolinux is used and that limits the architecture to x86 only. In any other case the appliance configured bootloader is used. This check examines if the host architecture is supported with the configured firmware on request of an ISO image.

`check_boot_description_exists``()`

If a kiwi initrd is used, a lookup to the specified boot description is done and fails early if it does not exist

`check_consistent_kernel_in_boot_and_system_image``()`

If a kiwi initrd is used, the kernel used to build the kiwi initrd and the kernel used in the system image must be the same in order to avoid an inconsistent boot setup

`check_container_tool_chain_installed``()`

When creating container images the specific tools are used in order to import and export OCI or Docker compatible images. This check searches for those tools to be installed in the build system and fails if it can't find them

`check_dracut_module_for_disk_oem_in_package_list``()`

OEM images if configured to use dracut as initrd system requires the KIWI provided dracut-kiwi-oem-repart module to be installed at the time dracut is called. Thus this runtime check examines if the required package is part of the package list in the image description.

check_dracut_module_for_disk_overlay_in_package_list()

Disk images configured to use a root filesystem overlay requires the KIWI provided kiwi-overlay dracut module to be installed at the time dracut is called. Thus this runtime check examines if the required package is part of the package list in the image description.

check_dracut_module_for_live_iso_in_package_list()

Live ISO images uses a dracut initrd to boot and requires the KIWI provided kiwi-live dracut module to be installed at the time dracut is called. Thus this runtime check examines if the required package is part of the package list in the image description.

check_dracut_module_for_oem_install_in_package_list()

OEM images if configured to use dracut as initrd system and configured with one of the installiso, installstick or installpxe attributes requires the KIWI provided dracut-kiwi-oem-dump module to be installed at the time dracut is called. Thus this runtime check examines if the required package is part of the package list in the image description.

check_dracut_module_versions_compatible_to_kiwi(*root_dir*)

KIWI images which makes use of kiwi dracut modules has to use module versions compatible with the version of this KIWI builder code base. This is important to avoid inconsistencies between the way how kiwi includes its own dracut modules and former version of those dracut modules which could be no longer compatible with the builder. Therefore this runtime check maintains a min_version constraint for which we know this KIWI builder to be compatible with.

check_efi_mode_for_disk_overlay_correctly_setup()

Disk images configured to use a root filesystem overlay only supports the standard EFI mode and not secure boot. That's because the shim setup performs changes to the root filesystem which can not be applied during the bootloader setup at build time because at that point the root filesystem is a read-only squashfs source.

check_image_include_repos_publicly_resolvable()

Verify that all repos marked with the imageinclude attribute can be resolved into a http based web URL

check_image_type_unique()

Verify that the selected image type is unique within the range of the configured types and profiles.

check_image_version_provided()

Kiwi requires a <version> element to be specified as part of at least one <preferences> section.

check_initrd_selection_required()

If the boot attribute is used without selecting kiwi as the initrd_system, the setting of the boot attribute will not have any effect. We assume that configurations which explicitly specify the boot attribute wants to use the custom kiwi initrd system and not dracut.

check_mediacheck_installed()

If the image description enables the mediacheck attribute the required tools to run this check must be installed on the image build host

check_repositories_configured()

Verify that that there are repositories configured

check_swap_name_used_with_lvm()

The optional oem-swapname is only effective if used together with the LVM volume manager. A name for the swap space can only be set if it is created as a LVM volume. In any other case the name does not apply to the system

check_syslinux_installed_if_isolinux_is_used()

ISO images that are configured to use isolinux requires the host to provide a set of syslinux binaries.

check_target_directory_not_in_shared_cache(*target_dir*)

The target directory must be outside of the kiwi shared cache directory in order to avoid busy mounts because kiwi bind mounts the cache directory into the image root tree to access host caching information

Parameters **target_dir** (*string*) – path name

check_volume_label_used_with_lvm()

The optional volume label in a systemdisk setup is only effective if the LVM, logical volume manager system is used. In any other case where the filesystem itself offers volume management capabilities there are no extra filesystem labels which can be applied per volume

check_volume_setup_defines_multiple_fullsize_volumes()

The volume size specification 'all' makes this volume to take the rest space available on the system. It's only allowed to specify one all size volume

check_volume_setup_defines_reserved_labels()

check_volume_setup_has_no_root_definition()

The root volume in a systemdisk setup is handled in a special way. It is not allowed to setup a custom name or mountpoint for the root volume. Therefore the size of the root volume can be setup via the @root volume name. This check looks up the volume setup and searches if there is a configuration for the '/' mountpoint which would cause the image build to fail

check_xen_uniquely_setup_as_server_or_guest()

If the image is classified to be used as Xen image, it can be either a Xen Server(dom0) or a Xen guest. The image configuration is checked if the information uniquely identifies the image as such

class kiwi.runtime_checker.**dracut_module_type**(*package*, *min_version*)

Bases: tuple

property min_version

Alias for field number 1

property package
Alias for field number 0

13.1.18 `kiwi.runtime_config` Module

class `kiwi.runtime_config.`**`RuntimeConfig`**(*reread=False*)
Bases: `object`

Implements reading of runtime configuration file:

1. Check for –config provided from the CLI
2. ~/.config/kiwi/config.yml
3. /etc/kiwi.yml

The KIWI runtime configuration file is a yaml formatted file containing information to control the behavior of the tools used by KIWI.

Parameters **`reread`** (*`bool`*) – reread runtime config

`get_bundle_compression`(*default=True*)
Return boolean value to express if the image bundle should contain XZ compressed image results or not.

bundle:

- compress: true|false

If compression of image build results is activated the size of the bundle is smaller and the download speed increases. However the image must be uncompressed before use

If no compression is explicitly configured, the provided default value applies

Parameters **`default`** (*`bool`*) – Default value

Returns True or False

Return type bool

`get_container_compression`()
Return compression algorithm to use for compression of container images

container:

- compress: xz|none

if no or invalid configuration data is provided, the default compression algorithm from the Defaults class is returned

Returns A name

Return type str

`get_disabled_runtime_checks`()
Returns disabled runtime checks. Checks can be disabled with:

runtime_checks:

- disable: check_container_tool_chain_installed

if the provided string does not match any RuntimeChecker method it is just ignored.

get_iso_tool_category()

Return tool category which should be used to build iso images

iso:

- tool_category: cdrtools|xorriso

if no or invalid configuration exists the default tool category from the Defaults class is returned

Returns A name

Return type str

get_max_size_constraint()

Returns the maximum allowed size of the built image. The value is returned in bytes and it is specified in build_constraints element with the max_size attribute. The value can be specified in bytes or it can be specified with m=MB or g=GB.

build_constraints:

- max_size: 700m

if no configuration exists None is returned

Returns byte value or None

Return type int

get_obs_api_credentials()

Return OBS API credentials if configured:

obs:

- **user:**
  - user_name: user_credentials

Returns List of Dicts with credentials per user

Return type list

get_obs_api_server_url()

Return URL of buildservice API server in:

obs:

- api_url: …

if no configuration exists the API server from the Defaults class is returned

Returns URL type data

Return type str

get_obs_download_server_url()

Return URL of buildservice download server in:

obs:

- download_url: . . .

if no configuration exists the downloadserver from the Defaults class is returned

Returns URL type data

Return type str

get_oci_archive_tool()

Return OCI archive tool which should be used on creation of container archives for OCI compliant images, e.g docker

oci:

- archive_tool: umoci

if no configuration exists the default tool from the Defaults class is returned

Returns A name

Return type str

get_package_changes(*default=True*)

Return boolean value to express if the image build and bundle should contain a .changes file. The .changes file contains the package changelog information from all packages installed into the image.

bundle:

- has_package_changes: true|false

By default the creation is switched on. When building in the Open Build Service the default is switched off because obs provides a .report file containing the same information.

Parameters **default** (*bool*) – Default value

Returns True or False

Return type bool

get_xz_options()

Return list of XZ compression options in:

xz:

- options: . . .

if no configuration exists None is returned

Returns

Contains list of options

```
['--option=value']
```

Return type list

is_obs_public()

Check if the buildservice configuration is public or private in:

obs:

- public: true|false

if no configuration exists we assume to be public

Returns True or False

Return type bool

13.1.19 kiwi.version Module

Global version information used in kiwi and the package

13.1.20 kiwi.xml_description Module

class kiwi.xml_description.**XMLDescription**(*description: str = '', derived_from: str = None*)

Bases: object

Implements data management for the image description

Supported description markup languages are XML, YAML, JSON and INI. The provided input file is converted into XML, transformed to the current RelaxNG schema via XSLT and validated against this result.

- XSLT Style Sheet processing to apply on this version of kiwi
- Schema Validation based on RelaxNG schema
- Loading XML data into internal data structures

Attributes

Parameters

- **description** (*str*) – path to description file
- **derived_from** (*str*) – path to base description file

get_extension_xml_data(*namespace_name: str*) → Any

Return the xml etree parse result for the specified extension namespace

Parameters **namespace_name** (*str*) – name of the extension namespace

Returns result of etree.parse

Return type object

load() → Any

Read XML description, validate it against the schema and the schematron rules and pass it to the autogenerated(generateDS) parser.

Returns instance of XML toplevel domain (image)

Return type object

13.1.21 `kiwi.xml_state` Module

class kiwi.xml_state.**XMLState**(*xml_data: Any*, *profiles: List = None*, *build_type: Any = None*)

Bases: `object`

Implements methods to get stateful information from the XML data

Parameters

- **xml_data** (*object*) – parse result from XMLDescription.load()
- **profiles** (*list*) – list of used profiles
- **build_type** (*object*) – build <type> section reference

add_container_config_label(*label_name: str*, *value: str*) → None

Adds a new label in the containerconfig section, if a label with the same name is already defined in containerconfig it gets overwritten by this method.

Parameters

- **label_name** (*str*) – the string representing the label name
- **value** (*str*) – the value of the label

add_repository(*repo_source: str*, *repo_type: str*, *repo_alias: str = None*, *repo_prio: str = ''*, *repo_imageinclude: bool = False*, *repo_package_gpgcheck: Optional[bool] = None*) → None

Add a new repository section at the end of the list

Parameters

- **repo_source** (*str*) – repository URI
- **repo_type** (*str*) – type name defined by schema
- **repo_alias** (*str*) – alias name
- **repo_prio** (*str*) – priority number, package manager specific
- **repo_imageinclude** (*bool*) – setup repository inside of the image
- **repo_package_gpgcheck** (*bool*) – enable/disable package gpg checks

copy_bootdelete_packages(*target_state: Any*) → None
Copy packages marked as bootdelete to the packages type=delete section in the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_bootincluded_archives(*target_state: Any*) → None
Copy archives marked as bootinclude to the packages type=bootstrap section in the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_bootincluded_packages(*target_state: Any*) → None
Copy packages marked as bootinclude to the packages type=image (or type=bootstrap if no type=image was found) section in the target xml state. The package will also be removed from the packages type=delete section in the target xml state if present there

Parameters **target_state** (*object*) – XMLState instance

copy_bootloader_section(*target_state: Any*) → None
Copy bootloader section from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_build_type_attributes(*attribute_names: List*, *target_state: Any*) → None
Copy specified attributes from this build type section to the target xml state build type section

Parameters

- **attribute_names** (*list*) – type section attributes
- **target_state** (*object*) – XMLState instance

copy_displayname(*target_state: Any*) → None
Copy image displayname from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_drivers_sections(*target_state: Any*) → None
Copy drivers sections from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_machine_section(*target_state: Any*) → None
Copy machine sections from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_name(*target_state: Any*) → None
Copy image name from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_oemconfig_section(*target_state: Any*) → None
Copy oemconfig sections from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_preferences_subsections (*section_names: List*, *target_state: Any*) → None

Copy subsections of the preferences sections, matching given section names, from this xml state to the target xml state

Parameters

- **section_names** (*list*) – preferences subsection names
- **target_state** (*object*) – XMLState instance

copy_repository_sections (*target_state: Any*, *wipe: bool = False*) → None

Copy repository sections from this xml state to the target xml state

Parameters

- **target_state** (*object*) – XMLState instance
- **wipe** (*bool*) – delete all repos in target prior to copy

copy_strip_sections (*target_state: Any*) → None

Copy strip sections from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

copy_systemdisk_section (*target_state: Any*) → None

Copy systemdisk sections from this xml state to the target xml state

Parameters **target_state** (*object*) – XMLState instance

delete_repository_sections () → None

Delete all repository sections matching configured profiles

delete_repository_sections_used_for_build () → None

Delete all repository sections used to build the image matching configured profiles

get_bootstrap_archives () → List

List of archive names from the type="bootstrap" packages section(s)

Returns archive names

Return type list

get_bootstrap_collection_type () → str

Collection type for packages sections matching type="bootstrap"

Returns collection type name

Return type str

get_bootstrap_collections () → List

List of collection names from the packages sections matching type="bootstrap"

Returns collection names

Return type list

get_bootstrap_packages(*plus_packages: List = None*) → List

List of package names from the type="bootstrap" packages section(s)

The list gets the selected package manager appended if there is a request to install packages inside of the image via a chroot operation

Parameters **plus_packages** (*list*) – list of additional packages

Returns package names

Return type list

get_bootstrap_packages_sections() → List

List of packages sections matching type="bootstrap"

Returns list of <packages> section reference(s)

Return type list

get_bootstrap_products() → List

List of product names from the packages sections matching type="bootstrap"

Returns product names

Return type list

get_build_type_bootloader_console() → Optional[str]

Return bootloader console setting for selected build type

Returns console string

Return type str

get_build_type_bootloader_name() → str

Return bootloader name for selected build type

Returns bootloader name

Return type str

get_build_type_bootloader_section() → Any

First bootloader section from the build type section

Returns <bootloader> section reference

Return type xml_parse::bootloader

get_build_type_bootloader_serial_line_setup() → Optional[str]

Return bootloader serial line setup parameters for the selected build type

Returns serial line setup

Return type str

get_build_type_bootloader_targettype() → Optional[str]

Return bootloader target type setting. Only relevant for the zipl bootloader because zipl is installed differently depending on the storage target it runs later

Returns target type string

Return type str

get_build_type_bootloader_timeout() → Optional[str]
Return bootloader timeout setting for selected build type

Returns timeout string

Return type str

get_build_type_bootloader_timeout_style() → Optional[str]
Return bootloader timeout style setting for selected build type

Returns timeout_style string

Return type str

get_build_type_containerconfig_section() → Any
First containerconfig section from the build type section

Returns <containerconfig> section reference

Return type xml_parse::containerconfig

get_build_type_format_options() → Dict
Disk format options returned as a dictionary

Returns format options

Return type dict

get_build_type_machine_section() → Any
First machine section from the build type section

Returns <machine> section reference

Return type xml_parse::machine

get_build_type_name() → str
Default build type name

Returns Content of image attribute from build type

Return type str

get_build_type_oemconfig_section() → Any
First oemconfig section from the build type section

Returns <oemconfig> section reference

Return type xml_parse::oemconfig

get_build_type_size(*include_unpartitioned: bool = False*) → Optional[*kiwi.xml_state.size_type*]
Size information from the build type section. If no unit is set the value is treated as mbytes

Parameters **include_unpartitioned** (*bool*) – sets if the unpartitioned area should be included in the computed size or not

Returns mbytes

Return type int

get_build_type_spare_part_fs_attributes() → Optional[List]
Build type specific list of filesystem attributes applied to the spare partition.

Returns list of strings or empty list

Return type list

get_build_type_spare_part_size() → Optional[int]
Size information for the spare_part size from the build type. If no unit is set the value is treated as mbytes

Returns mbytes

Return type int

get_build_type_system_disk_section() → Any
First system disk section from the build type section

Returns <systemdisk> section reference

Return type xml_parse::systemdisk

get_build_type_unpartitioned_bytes() → int
Size of the unpartitioned area for image in megabytes

Returns mbytes

Return type int

get_build_type_vagrant_config_section() → Any
First vagrantconfig section from the build type section

Returns <vagrantconfig> section reference

Return type xml_parse::vagrantconfig

get_build_type_vmconfig_entries() → List
List of vmconfig-entry section values from the first machine section in the build type section

Returns <vmconfig_entry> section reference(s)

Return type list

get_build_type_vmdisk_section() → Any
First vmdisk section from the first machine section in the build type section

Returns <vmdisk> section reference

Return type xml_parse::vmdisk

get_build_type_vmdvd_section() → Any
First vmdvd section from the first machine section in the build type section

Returns <vmdvd> section reference

Return type xml_parse::vmdvd

get_build_type_vmnic_entries() → List
vmnic section(s) from the first machine section in the build type section

Returns list of <vmnic> section reference(s)

Return type list

get_collection_type(*section_type: str = 'image'*) → str
Collection type from packages sections matching given section type.

If no collection type is specified the default collection type is set to: onlyRequired

Parameters **section_type** (*str*) – type name from packages section

Returns collection type name

Return type str

get_collections(*section_type: str = 'image'*) → List
List of collection names from the packages sections matching type=section_type and type=build_type

Returns collection names

Return type list

get_container_config() → Dict
Dictionary of containerconfig information

Takes attributes and subsection data from the selected <containerconfig> section and stores it in a dictionary

get_derived_from_image_uri() → Optional[*kiwi.system.uri.Uri*]
Uri object of derived image if configured

Specific image types can be based on a master image. This method returns the location of this image when configured in the XML description

Returns Instance of Uri

Return type object

get_description_section() → *kiwi.xml_state.description_type*
The description section

Returns description_type tuple providing the elements author contact and specification

Return type tuple

get_disk_start_sector() → int
First disk sector number to be used by the first disk partition.

Returns number

Return type int

get_distribution_name_from_boot_attribute() → str
Extract the distribution name from the boot attribute of the build type section.

If no boot attribute is configured or the contents does not match the kiwi defined naming schema for boot image descriptions, an exception is thrown

Returns lowercase distribution name

Return type str

`get_drivers_list()` → List

List of driver names from all drivers sections matching configured profiles

Returns driver names

Return type list

`get_fs_create_option_list()` → List

List of root filesystem creation options

The list contains elements with the information from the fscreateoptions attribute string that got split into its substring components

Returns list with create options

Return type list

`get_fs_mount_option_list()` → List

List of root filesystem mount options

The list contains one element with the information from the fsmountoptions attribute. The value there is passed along to the -o mount option

Returns max one element list with mount option string

Return type list

`get_image_packages_sections()` → List

List of packages sections matching type="image"

Returns list of <packages> section reference(s)

Return type list

`get_image_version()` → str

Image version from preferences section.

Multiple occurences of version in preferences sections are not forbidden, however only the first version found defines the final image version

Returns Content of <version> section

Return type str

`get_initrd_system()` → str

Name of initrd system to use

Depending on the image type a specific initrd system is either pre selected or free of choice according to the XML type setup

Returns dracut, kiwi or None

Return type str

get_installmedia_initrd_modules(*action: str*) → List[str]
Gets the list of modules to append in installation initrds

Returns a list of dracut module names

Return type list

get_locale() → Optional[List]
Gets list of locale names if configured. Takes the first locale setup from the existing preferences sections into account.

Returns List of names or None

Return type list|None

get_oemconfig_oem_multipath_scan() → bool
State value to activate multipath maps. Returns a boolean value if specified or False

Returns Content of <oem-multipath-scan> section value

Return type bool

get_oemconfig_oem_resize() → bool
State value to activate/deactivate disk resize. Returns a boolean value if specified or True to set resize on by default

Returns Content of <oem-resize> section value

Return type bool

get_oemconfig_swap_mbytes() → Optional[int]
Return swapsize in MB if requested or None

Operates on the value of oem-swap and if set to true returns the given size or the default value.

Returns Content of <oem-swapsize> section value or default

Return type int

get_oemconfig_swap_name() → str
Return the swap space name

Operates on the value of oem-swapname and if set returns the configured name or the default name: LVSwap

The name of the swap space is used only if the image is configured to use the LVM volume manager. In this case swap is a volume and the volume takes a name. In any other case the given name will have no effect.

Returns Content of <oem-swapname> section value or default

Return type str

get_package_manager() → str
Get configured package manager from selected preferences section

Returns Content of the <packagemanager> section

Return type str

get_package_sections (*packages_sections: List*) → List[*kiwi.xml_state.package_type*]
List of package sections from the given packages sections. Each list element contains a tuple with the <package> section reference and the <packages> section this package belongs to

If a package entry specfies an architecture, it is only taken if the host architecture matches the configured architecture

Parameters **packages_sections** (*list*) – <packages>

Returns

Contains list of package_type tuples

```
[package_type(packages_section=object, package_
↪section=object)]
```

Return type list

get_packages_sections (*section_types: List*) → List
List of packages sections matching given section type(s)

Parameters **section_types** (*list*) – type name(s) from packages sections

Returns list of <packages> section reference(s)

Return type list

get_preferences_sections () → List
All preferences sections for the selected profiles that match the host architecture

Returns list of <preferences> section reference(s)

Return type list

get_products (*section_type: str = 'image'*) → List
List of product names from the packages sections matching type=section_type and type=build_type

Parameters **section_type** (*str*) – type name from packages section

Returns product names

Return type list

get_repository_sections () → List
List of all repository sections matching configured profiles

Returns <repository> section reference(s)

Return type list

get_repository_sections_used_for_build () → List
List of all repositorys sections used to build the image and matching configured profiles.

Returns <repository> section reference(s)

Return type list

get_repository_sections_used_in_image() → List
List of all repositorys sections to be configured in the resulting image matching configured profiles.

Returns <repository> section reference(s)

Return type list

get_root_filesystem_uuid() → Optional[str]
Return preserved UUID

get_root_partition_uuid() → Optional[str]
Return preserved PARTUUID

get_rpm_check_signatures() → bool
Gets the rpm-check-signatures configuration flag. Returns False if not present.

Returns True or False

Return type bool

get_rpm_excludedocs() → bool
Gets the rpm-excludedocs configuration flag. Returns False if not present.

Returns True or False

Return type bool

get_rpm_locale() → Optional[List]
Gets list of locale names to filter out by rpm if rpm-locale-filtering is switched on the the list always contains: [POSIX, C, C.UTF-8] and is extended by the optionaly configured locale

Returns List of names or None

Return type list|None

get_rpm_locale_filtering() → bool
Gets the rpm-locale-filtering configuration flag. Returns False if not present.

Returns True or False

Return type bool

get_strip_files_to_delete() → List
Items to delete from strip section

Returns item names

Return type list

get_strip_libraries_to_keep() → List
Libraries to keep from strip section

Returns librarie names

Return type list

get_strip_list(*section_type: str*) → List
List of strip names matching the given section type and profiles

Parameters **section_type** (*str*) – type name from packages section

Returns strip names

Return type list

get_strip_tools_to_keep() → List
Tools to keep from strip section

Returns tool names

Return type list

get_system_archives() → List
List of archive names from the packages sections matching type="image" and type=build_type

Returns archive names

Return type list

get_system_collection_type() → str
Collection type for packages sections matching type="image"

Returns collection type name

Return type str

get_system_collections() → List
List of collection names from the packages sections matching type="image"

Returns collection names

Return type list

get_system_ignore_packages() → List
List of ignore package names from the packages sections matching type="image" and type=build_type

Returns package names

Return type list

get_system_packages() → List
List of package names from the packages sections matching type="image" and type=build_type

Returns package names

Return type list

get_system_products() → List
List of product names from the packages sections matching type="image"

Returns product names

Return type list

`get_to_become_deleted_packages`(*force: bool = True*) → List
List of package names from the type="delete" or type="uninstall" packages section(s)

Parameters **`force`** (*`bool`*) – return "delete" type if True, "uninstall" type otherwise

Returns package names

Return type list

`get_user_groups`(*user_name*) → List
List of group names matching specified user

Each entry in the list is the name of a group that the specified user belongs to. The first item in the list is the login or primary group. The list will be empty if no groups are specified in the description file.

Returns groups data for the given user

Return type list

`get_users`() → List
List of configured users.

Each entry in the list is a single xml_parse::user instance.

Returns list of <user> section reference(s)

Return type list

`get_users_sections`() → List
All users sections for the selected profiles

Returns list of <users> section reference(s)

Return type list

`get_vagrant_config_virtualbox_guest_additions`() → bool
Attribute virtualbox_guest_additions_present from the first vagrantconfig section.

Returns True|False

Return type bool

`get_volume_group_name`() → str
Volume group name from selected <systemdisk> section

Returns volume group name

Return type str

`get_volume_management`() → Optional[str]
Provides information which volume management system is used

Returns name of volume manager

Return type str

get_volumes() → List[*kiwi.xml_state.volume_type*]

List of configured systemdisk volumes.

Each entry in the list is a tuple with the following information

- name: name of the volume
- size: size of the volume
- realpath: system path to lookup volume data. If no mountpoint is set the volume name is used as data path.
- mountpoint: volume mount point and volume data path
- fullsize: takes all space True|False
- attributes: list of volume attributes handled via chattr

Returns

Contains list of volume_type tuples

```
[
    volume_type(
        name=volume_name,
        size=volume_size,
        realpath=path,
        mountpoint=path,
        fullsize=True,
        label=volume_label,
        attributes=['no-copy-on-write'],
        is_root_volume=True|False
    )
]
```

Return type list

is_xen_guest() → bool

Check if build type setup specifies a Xen Guest (domX) The check is based on the architecture, the firmware and xen_loader configuration values:

- We only support Xen setup on the x86_64 architecture
- Firmware pointing to ec2 means the image is targeted to run in Amazon EC2 which is a Xen guest
- Machine setup with a xen_loader attribute also indicates a Xen guest target

Returns True or False

Return type bool

is_xen_server() → bool

Check if build type domain setup specifies a Xen Server (dom0)

Returns True or False

Return type bool

`package_matches_host_architecture`(*package: Any*) → bool

Tests if the given package section is applicable for the current host architecture. If no architecture is specified within the section it is considered as a match returning True.

Note: The XML section pointer must provide an arch attribute

Parameters **`section`** – XML section object

Returns True or False

Return type bool

`preferences_matches_host_architecture`(*preferences: Any*) → bool

Tests if the given preferences section is applicable for the current host architecture. If no architecture is specified within the section it is considered as a match returning True.

Note: The XML section pointer must provide an arch attribute

Parameters **`section`** – XML section object

Returns True or False

Return type bool

`profile_matches_host_architecture`(*profile: Any*) → bool

Tests if the given profile section is applicable for the current host architecture. If no architecture is specified within the section it is considered as a match returning True.

Note: The XML section pointer must provide an arch attribute

Parameters **`section`** – XML section object

Returns True or False

Return type bool

`set_container_config_tag`(*tag: str*) → None

Set new tag name in containerconfig section

In order to set a new tag value an existing containerconfig and tag setup is required

Parameters **`tag`** (`str`) – tag name

`set_derived_from_image_uri`(*uri: str*) → None

Set derived_from attribute to a new value

In order to set a new value the derived_from attribute must be already present in the image configuration

Parameters **`uri`** (`str`) – URI

set_repository(*repo_source: str*, *repo_type: str*, *repo_alias: str*, *repo_prio: str*, *repo_imageinclude: bool = False*, *repo_package_gpgcheck: Optional[bool] = None*) → None

Overwrite repository data of the first repository

Parameters

- **repo_source** (*str*) – repository URI
- **repo_type** (*str*) – type name defined by schema
- **repo_alias** (*str*) – alias name
- **repo_prio** (*str*) – priority number, package manager specific
- **repo_imageinclude** (*bool*) – setup repository inside of the image
- **repo_package_gpgcheck** (*bool*) – enable/disable package gpg checks

set_root_filesystem_uuid(*uuid: str*) → None

Store UUID provided in uuid as state information

Parameters **uuid** (*str*) – UUID

set_root_partition_uuid(*uuid: str*) → None

Store PARTUUID provided in uuid as state information

Parameters **uuid** (*str*) – PARTUUID

class kiwi.xml_state.**description_type**(*author*, *contact*, *specification*)

Bases: tuple

property author

Alias for field number 0

property contact

Alias for field number 1

property specification

Alias for field number 2

class kiwi.xml_state.**package_type**(*packages_section*, *package_section*)

Bases: tuple

property package_section

Alias for field number 1

property packages_section

Alias for field number 0

class kiwi.xml_state.**size_type**(*mbytes*, *additive*)

Bases: tuple

property additive

Alias for field number 1

property mbytes
Alias for field number 0

class kiwi.xml_state.**volume_type**(*name, size, realpath, mountpoint, fullsize, label, attributes, is_root_volume*)
Bases: `tuple`

property attributes
Alias for field number 6

property fullsize
Alias for field number 4

property is_root_volume
Alias for field number 7

property label
Alias for field number 5

property mountpoint
Alias for field number 3

property name
Alias for field number 0

property realpath
Alias for field number 2

property size
Alias for field number 1

13.1.22 Module Contents

13.2 kiwi.archive Package

13.2.1 Submodules

13.2.2 `kiwi.archive.cpio` Module

class kiwi.archive.cpio.**ArchiveCpio**(*filename*)
Bases: `object`

Extraction/Creation of cpio archives

Parameters filename (`string`) – filename to use for archive extraction or creation

create(*source_dir, exclude=None*)
Create cpio archive

Parameters

- **source_dir** (`string`) – data source directory

- **`exclude`** (*`list`*) – list of excluded items

`extract` (*dest_dir*)
Extract cpio archive contents

Parameters **`dest_dir`** (*`string`*) – target data directory

13.2.3 `kiwi.archive.tar` Module

class `kiwi.archive.tar.`**`ArchiveTar`**(*filename*, *create_from_file_list=True*, *file_list=None*)
Bases: `object`

Extraction/Creation of tar archives

The tarfile python module is not used by that class, since it does not provide support for some relevant features in comparison to the GNU tar command (e.g. numeric-owner). Moreover tarfile lacks support for xz compression under Python v2.7.

Parameters

- **`filename`** (*`string`*) – filename to use for archive extraction or creation
- **`create_from_file_list`** (*`bool`*) – use file list not entire directory to create the archive
- **`file_list`** (*`list`*) – list of files and directorie names to archive

`append_files`(*source_dir*, *files_to_append*, *options=None*)
Append files to an already existing uncompressed tar archive

Parameters

- **`source_dir`** (*`string`*) – data source directory
- **`files_to_append`** (*`list`*) – list of items to append
- **`options`** (*`list`*) – custom options

`create`(*source_dir*, *exclude=None*, *options=None*)
Create uncompressed tar archive

Parameters

- **`source_dir`** (*`string`*) – data source directory
- **`exclude`** (*`list`*) – list of excluded items
- **`options`** (*`list`*) – custom creation options

`create_gnu_gzip_compressed`(*source_dir*, *exclude=None*)
Create gzip compressed tar archive

Parameters

- **`source_dir`** (*`string`*) – data source directory

- **exclude** (*list*) – list of excluded items

create_xz_compressed(*source_dir, exclude=None, options=None, xz_options=None*)
Create XZ compressed tar archive

Parameters

- **source_dir** (*string*) – data source directory
- **exclude** (*list*) – list of excluded items
- **options** (*list*) – custom tar creation options
- **xz_options** (*list*) – custom xz compression options

extract(*dest_dir*)
Extract tar archive contents

Parameters **dest_dir** (*string*) – target data directory

13.2.4 Module Contents

13.3 kiwi.boot.image Package

13.3.1 Submodules

13.3.2 `kiwi.boot.image.base` Module

class kiwi.boot.image.base.**BootImageBase**(*xml_state, target_dir, root_dir=None, signing_keys=None*)
Bases: `object`

Base class for boot image(initrd) task

Parameters

- **xml_state** (*object*) – Instance of `XMLState`
- **target_dir** (*string*) – target dir to store the initrd
- **root_dir** (*string*) – system image root directory
- **signing_keys** (*list*) – list of package signing keys

cleanup()
Cleanup temporary boot image data if any

create_initrd(*mbrid=None, basename=None, install_initrd=False*)
Implements creation of the initrd

Parameters

- **mbrid** (*object*) – instance of ImageIdentifier

- **basename** (*string*) – base initrd file name
- **install_initrd** (*bool*) – installation media initrd

Implementation in specialized boot image class

dump (*filename*)
Pickle dump this instance to a file. If the object dump is requested the destructor code will also be disabled in order to preserve the generated data

Parameters **filename** (*string*) – file path name

get_boot_description_directory ()
Provide path to the boot image XML description

Returns path name

Return type str

get_boot_names ()
Provides kernel and initrd names for the boot image

Returns

Contains boot_names_type tuple

```
boot_names_type(
    kernel_name='INSTALLED_KERNEL',
    initrd_name='DRACUT_OUTPUT_NAME'
)
```

Return type tuple

import_system_description_elements ()
Copy information from the system image relevant to create the boot image to the boot image state XML description

include_file (*filename*, *install_media=False*)
Include file to boot image

For kiwi boot images this is done by adding package or archive definitions with the bootinclude attribute. Thus for kiwi boot images the method is a noop

Parameters

- **filename** (*string*) – file path name
- **install_media** (*bool*) – include also for installation media initrd

include_module (*module*, *install_media=False*)
Include module to boot image

For kiwi boot no modules configuration is required. Thus in such a case this method is a noop.

Parameters

- **module** (*string*) – module to include

- **install_media** (*bool*) – include the module for install initrds

is_prepared()
Check if initrd system is prepared.

Returns True or False

Return type bool

load_boot_xml_description()
Load the boot image description referenced by the system image description boot attribute

omit_module(*module*, *install_media=False*)
Omit module to boot image

For kiwi boot no modules configuration is required. Thus in such a case this method is a noop.

Parameters

- **module** (*string*) – module to omit
- **install_media** (*bool*) – omit the module for install initrds

post_init()
Post initialization method

Implementation in specialized boot image class

prepare()
Prepare new root system to create initrd from. Implementation is only needed if there is no other root system available

Implementation in specialized boot image class

set_static_modules(*modules*, *install_media=False*)
Set static modules list for boot image

For kiwi boot no modules configuration is required. Thus in such a case this method is a noop.

Parameters

- **modules** (*list*) – list of modules to include
- **install_media** (*bool*) – lists the modules for install initrds

write_system_config_file(*config*, *config_file=None*)
Writes relevant boot image configuration into configuration file that will be part of the system image.

This is used to configure any further boot image rebuilds after deployment. For instance, initrds recreated on kernel update.

For kiwi boot no specific configuration is required for initrds recreation, thus this method is a noop in that case.

Parameters

- **config** (*dict*) – dictonary including configuration parameters
- **config_file** (*string*) – configuration file to write

13.3.3 kiwi.boot.image.dracut Module

class kiwi.boot.image.dracut.**BootImageDracut**(*xml_state*, *target_dir*, *root_dir=None*, *signing_keys=None*)

Bases: *kiwi.boot.image.base.BootImageBase*

Implements creation of dracut boot(initrd) images.

create_initrd(*mbrid=None*, *basename=None*, *install_initrd=False*)

Create kiwi .profile environment to be included in dracut initrd. Call dracut as chroot operation to create the initrd and move the result into the image build target directory

Parameters

- **mbrid** (*object*) – unused
- **basename** (*string*) – base initrd file name
- **install_initrd** (*bool*) – installation media initrd

include_file(*filename*, *install_media=False*)

Include file to dracut boot image

Parameters **filename** (*string*) – file path name

include_module(*module*, *install_media=False*)

Include module to dracut boot image

Parameters

- **module** (*string*) – module to include
- **install_media** (*bool*) – include the module for install initrds

omit_module(*module*, *install_media=False*)

Omit module to dracut boot image

Parameters

- **module** (*string*) – module to omit
- **install_media** (*bool*) – omit the module for install initrds

post_init()

Post initialization method

Initialize empty list of dracut caller options

prepare()
Prepare dracut caller environment

- Setup machine_id(s) to be generic and rebuild by dracut on boot

set_static_modules(*modules*, *install_media=False*)
Set static dracut modules list for boot image

Parameters

- **modules** (*list*) – list of the modules to include
- **install_media** (*bool*) – lists the modules for install initrds

write_system_config_file(*config*, *config_file=None*)
Writes modules configuration into a dracut configuration file.

Parameters

- **config** (*dict*) – a dictionary containing the modules to add and omit
- **conf_file** (*string*) – configuration file to write

13.3.4 kiwi.boot.image.builtin_kiwi Module

class kiwi.boot.image.builtin_kiwi.**BootImageKiwi**(*xml_state*, *target_dir*, *root_dir=None*, *signing_keys=None*)

Bases: *kiwi.boot.image.base.BootImageBase*

Implements preparation and creation of kiwi boot(initrd) images

The kiwi initrd is a customized first boot initrd which allows to control the first boot an appliance. The kiwi initrd replaces itself after first boot by the result of dracut.

cleanup()
Cleanup temporary boot image data if any

create_initrd(*mbrid=None*, *basename=None*, *install_initrd=False*)
Create initrd from prepared boot system tree and compress the result

Parameters

- **mbrid** (*object*) – instance of ImageIdentifier
- **basename** (*string*) – base initrd file name
- **install_initrd** (*bool*) – installation media initrd

post_init()
Post initialization method

Creates custom directory to prepare the boot image root filesystem which is a separate image to create the initrd from

prepare()
Prepare new root system suitable to create a kiwi initrd from it

13.3.5 Module Contents

class kiwi.boot.image.**BootImage**
Bases: `object`

BootImge Factory

Parameters

- **xml_state** (*object*) – Instance of `XMLState`
- **target_dir** (*string*) – target dir to store the initrd
- **root_dir** (*string*) – system image root directory
- **signing_keys** (*list*) – list of package signing keys

static new(*xml_state: object, target_dir: str, root_dir: str = None, signing_keys: List = None*)

13.4 kiwi.bootloader.config Package

13.4.1 Submodules

13.4.2 `kiwi.bootloader.config.base` Module

class kiwi.bootloader.config.base.**BootLoaderConfigBase**(*xml_state, root_dir, boot_dir=None, custom_args=None*)
Bases: `object`

Base class for bootloader configuration

Parameters

- **xml_state** (*object*) – instance of `XMLState`
- **root_dir** (*string*) – root directory path name
- **custom_args** (*dict*) – custom bootloader arguments dictionary

create_efi_path(*in_sub_dir='boot/efi'*)
Create standard EFI boot directory structure

Parameters **in_sub_dir** (*string*) – toplevel directory

Returns Full qualified EFI boot path

Return type str

failsafe_boot_entry_requested()
Check if a failsafe boot entry is requested

Returns True or False

Return type bool

get_boot_cmdline(*uuid=None*)
Boot commandline arguments passed to the kernel

Parameters **uuid** (*string*) – boot device UUID

Returns kernel boot arguments

Return type str

get_boot_path(*target='disk'*)
Bootloader lookup path on boot device

If the bootloader reads the data it needs to boot, it does that from the configured boot device. Depending if that device is an extra boot partition or the root partition or or based on a non standard filesystem like a btrfs snapshot, the path name varies

Parameters **target** (*string*) – target name: disk|iso

Returns path name

Return type str

get_boot_theme()
Bootloader Theme name

Returns theme name

Return type str

get_boot_timeout_seconds()
Bootloader timeout in seconds

If no timeout is specified the default timeout applies

Returns timeout seconds

Return type int

get_continue_on_timeout()
Check if the boot should continue after boot timeout or not

Returns True or False

Return type bool

get_gfxmode(*target*)
Graphics mode according to bootloader target

Bootloaders which support a graphics mode can be configured to run graphics in a specific resolution and colors. There is no standard for this setup which causes kiwi to create a mapping from the kernel vesa mode number to the corresponding bootloader graphics mode setup

Parameters **target** (*string*) – bootloader name

Returns boot graphics mode

Return type str

get_install_image_boot_default(*loader=None*)

Provide the default boot menu entry identifier for install images

The install image can be configured to provide more than one boot menu entry. Menu entries configured are:

- [0] Boot From Hard Disk
- [1] Install
- [2] Failsafe Install

The installboot attribute controlls which of these are used by default. If not specified the boot from hard disk entry will be the default. Depending on the specified loader type either an entry number or name will be returned.

Parameters **loader** (*string*) – bootloader name

Returns menu name or id

Return type str

get_menu_entry_install_title()

Prefixed menu entry title for install images

If no displayname is specified in the image description, the menu title is constructed from the image name

Returns title text

Return type str

get_menu_entry_title(*plain=False*)

Prefixed menu entry title

If no displayname is specified in the image description, the menu title is constructed from the image name and build type

Parameters **plain** (*bool*) – indicate to add built type into title text

Returns title text

Return type str

post_init(*custom_args*)

Post initialization method

Store custom arguments by default

Parameters **custom_args** (*dict*) – custom bootloader arguments

quote_title(*name*)
Quote special characters in the title name

Not all characters can be displayed correctly in the bootloader environment. Therefore a quoting is required

Parameters **name** (*string*) – title name

Returns quoted text

Return type str

setup_disk_boot_images(*boot_uuid*, *lookup_path=None*)
Create bootloader images for disk boot

Some bootloaders requires to build a boot image the bootloader can load from a specific offset address or from a standardized path on a filesystem.

Parameters

- **boot_uuid** (*string*) – boot device UUID
- **lookup_path** (*string*) – custom module lookup path

Implementation in specialized bootloader class required

setup_disk_image_config(*boot_uuid*, *root_uuid*, *hypervisor*, *kernel*, *initrd*, *boot_options={}*)
Create boot config file to boot from disk.

Parameters

- **boot_uuid** (*string*) – boot device UUID
- **root_uuid** (*string*) – root device UUID
- **hypervisor** (*string*) – hypervisor name
- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name
- **boot_options** (*dict*) – custom options dictionary required to setup the bootloader. The scope of the options covers all information needed to setup and configure the bootloader and gets effective in the individual implementation. boot_options should not be mixed up with commandline options used at boot time. This information is provided from the get_*_cmdline methods. The contents of the dictionary can vary between bootloaders or even not be needed

Implementation in specialized bootloader class required

setup_install_boot_images(*mbrid*, *lookup_path=None*)
Create bootloader images for ISO boot an install media

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **lookup_path** (*string*) – custom module lookup path

Implementation in specialized bootloader class required

setup_install_image_config(*mbrid*, *hypervisor*, *kernel*, *initrd*)
Create boot config file to boot from install media in EFI mode.

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **hypervisor** (*string*) – hypervisor name
- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name

Implementation in specialized bootloader class required

setup_live_boot_images(*mbrid*, *lookup_path=None*)
Create bootloader images for ISO boot a live ISO image

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **lookup_path** (*string*) – custom module lookup path

Implementation in specialized bootloader class required

setup_live_image_config(*mbrid*, *hypervisor*, *kernel*, *initrd*)
Create boot config file to boot live ISO image in EFI mode.

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **hypervisor** (*string*) – hypervisor name
- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name

Implementation in specialized bootloader class required

setup_sysconfig_bootloader()
Create or update etc/sysconfig/bootloader by parameters required according to the bootloader setup

Implementation in specialized bootloader class required

write()
Write config data to config file.

Implementation in specialized bootloader class required

write_meta_data(*root_uuid=None*, *boot_options=''*)
Write bootloader setup meta data files

Parameters

- **root_uuid** (*string*) – root device UUID

- **boot_options** (*string*) – kernel options as string

Implementation in specialized bootloader class optional

13.4.3 kiwi.bootloader.config.grub2 Module

class kiwi.bootloader.config.grub2.**BootLoaderConfigGrub2** (*xml_state, root_dir, boot_dir=None, custom_args=None*)

Bases: *kiwi.bootloader.config.base.BootLoaderConfigBase*

grub2 bootloader configuration.

post_init (*custom_args*)
grub2 post initialization method

Parameters custom_args (*dict*) – Contains grub config arguments

```
{'grub_directory_name': 'grub|grub2'}
```

setup_disk_boot_images (*boot_uuid, lookup_path=None*)
Create/Provide grub2 boot images and metadata

In order to boot from the disk grub2 modules, images and theme data needs to be created and provided at the correct place in the filesystem

Parameters

- **boot_uuid** (*string*) – boot device UUID
- **lookup_path** (*string*) – custom module lookup path

setup_disk_image_config (*boot_uuid=None, root_uuid=None, hypervisor=None, kernel=None, initrd=None, boot_options={}*)
Create grub2 config file to boot from disk using grub2-mkconfig

Parameters

- **boot_uuid** (*string*) – unused
- **root_uuid** (*string*) – unused
- **hypervisor** (*string*) – unused
- **kernel** (*string*) – unused
- **initrd** (*string*) – unused
- **boot_options** (*dict*) –

options dictionary that has to contain the root and boot device and optional volume configuration. KIWI has to mount the system prior to run grub2-mkconfig.

```
{
    'root_device': string,
    'boot_device': string,
    'efi_device': string,
    'system_volumes': volume_manager_instance.get_volumes()
}
```

setup_install_boot_images(*mbrid*, *lookup_path=None*)
Create/Provide grub2 boot images and metadata

In order to boot from the ISO grub2 modules, images and theme data needs to be created and provided at the correct place on the iso filesystem

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **lookup_path** (*string*) – custom module lookup path

setup_install_image_config(*mbrid*, *hypervisor='xen.gz'*, *kernel='linux'*, *initrd='initrd'*)
Create grub2 config file to boot from an ISO install image

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **hypervisor** (*string*) – hypervisor name
- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name

setup_live_boot_images(*mbrid*, *lookup_path=None*)
Create/Provide grub2 boot images and metadata

Calls setup_install_boot_images because no different action required

setup_live_image_config(*mbrid*, *hypervisor='xen.gz'*, *kernel='linux'*, *initrd='initrd'*)
Create grub2 config file to boot a live media ISO image

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **hypervisor** (*string*) – hypervisor name
- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name

write()
Write bootloader configuration

- writes grub.cfg template by KIWI if template system is used
- creates an embedded fat efi image for EFI ISO boot

write_meta_data(*root_uuid=None*, *boot_options=''*)
Write bootloader setup meta data files

- cmdline arguments initialization
- etc/default/grub setup file
- etc/default/zipl2grub.conf.in (s390 only)
- etc/sysconfig/bootloader

Parameters

- **root_uuid** (*string*) – root device UUID
- **boot_options** (*string*) – kernel options as string
- **iso_boot** (*bool*) – indicate target is an ISO

13.4.4 kiwi.bootloader.config.isolinux Module

class kiwi.bootloader.config.isolinux.**BootLoaderConfigIsoLinux**(*xml_state*, *root_dir*, *boot_dir=None*, *cus-tom_args=Non*

Bases: *kiwi.bootloader.config.base.BootLoaderConfigBase*

isolinux bootloader configuration.

post_init(*custom_args*)
isolinux post initialization method

Parameters **custom_args** (*dict*) – custom isolinux config arguments

setup_install_boot_images(*mbrid*, *lookup_path=None*)
Provide isolinux boot metadata

No extra boot images must be created for isolinux

Parameters

- **mbrid** (*string*) – unused
- **lookup_path** (*string*) – unused

setup_install_image_config(*mbrid*, *hypervisor='xen.gz'*, *kernel='linux'*, *initrd='initrd'*)
Create isolinux.cfg in memory from a template suitable to boot from an ISO image in BIOS boot mode

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **hypervisor** (*string*) – hypervisor name

- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name

setup_live_boot_images (*mbrid*, *lookup_path=None*)

Provide isolinux boot metadata

No extra boot images must be created for isolinux

Parameters

- **mbrid** (*string*) – unused
- **lookup_path** (*string*) – unused

setup_live_image_config (*mbrid*, *hypervisor='xen.gz'*, *kernel='linux'*, *initrd='initrd'*)

Create isolinux.cfg in memory from a template suitable to boot a live system from an ISO image in BIOS boot mode

Parameters

- **mbrid** (*string*) – mbrid file name on boot device
- **hypervisor** (*string*) – hypervisor name
- **kernel** (*string*) – kernel name
- **initrd** (*string*) – initrd name

write ()

Write isolinux.cfg and isolinux.msg file

13.4.5 Module Contents

class kiwi.bootloader.config.**BootLoaderConfig**

Bases: object

BootLoaderConfig factory

Parameters

- **name** (*string*) – bootloader name
- **xml_state** (*object*) – instance of XMLState
- **root_dir** (*string*) – root directory path name
- **custom_args** (*dict*) – custom bootloader config arguments dictionary

static new (*name: str*, *xml_state: object*, *root_dir: str*, *boot_dir: str = None*, *custom_args: Dict = None*)

13.5 kiwi.bootloader.install Package

13.5.1 Submodules

13.5.2 `kiwi.bootloader.install.base` Module

class `kiwi.bootloader.install.base.`**`BootLoaderInstallBase`**(*root_dir*, *device_provider*, *custom_args=None*)

Bases: `object`

Base class for bootloader installation on device

Parameters

- **`root_dir`** (*`string`*) – root directory path name
- **`device_provider`** (*`object`*) – instance of `DeviceProvider`
- **`custom_args`** (*`dict`*) – custom arguments dictionary

`install`()

Install bootloader on self.device

Implementation in specialized bootloader install class required

`install_required`()

Check if bootloader needs to be installed

Implementation in specialized bootloader install class required

`post_init`(*custom_args*)

Post initialization method

Store custom arguments by default

Parameters **`custom_args`** (*`dict`*) – custom bootloader arguments

`secure_boot_install`()

Run shim-install in self.device for secure boots

Implementation in specialized bootloader install class required

13.5.3 `kiwi.bootloader.install.grub2` Module

class kiwi.bootloader.install.grub2.**BootLoaderInstallGrub2**(*root_dir, device_provider, custom_args=None*)

Bases: *kiwi.bootloader.install.base.BootLoaderInstallBase*

grub2 bootloader installation

install()

Install bootloader on disk device

install_required()

Check if grub2 has to be installed

Take architecture and firmware setup into account to check if bootloader code in a boot record is required

Returns True or False

Return type bool

post_init(*custom_args*)

grub2 post initialization method

Parameters **custom_args** (*dict*) – Contains custom grub2 bootloader arguments

```
{
    'target_removable': bool,
    'system_volumes': list_of_volumes,
    'firmware': FirmWare_instance,
    'efi_device': string,
    'boot_device': string,
    'root_device': string
}
```

secure_boot_install()

Run shim-install in self.device for secure boots

Implementation in specialized bootloader install class required

13.5.4 Module Contents

class kiwi.bootloader.install.**BootLoaderInstall**

Bases: object

BootLoaderInstall Factory

Parameters

- **name** (*string*) – bootloader name

- **root_dir** (*string*) – root directory path name
- **device_provider** (*object*) – instance of `DeviceProvider`
- **custom_args** (*dict*) – custom arguments dictionary

static new(*name: str, root_dir: str, device_provider: object, custom_args: Dict = None*)

13.6 kiwi.bootloader Package

13.6.1 Module Contents

13.7 kiwi.bootloader.template Package

13.7.1 Submodules

13.7.2 `kiwi.bootloader.template.grub2` Module

class `kiwi.bootloader.template.grub2.`**BootLoaderTemplateGrub2**

Bases: `object`

grub2 configuraton file templates

get_install_template(*failsafe=True, hybrid=True, terminal='gfxterm', with_timeout=True*)

Bootloader configuration template for install media

Parameters

- **failsafe** (*bool*) – with failsafe true|false
- **hybrid** (*bool*) – with hybrid true|false
- **terminal** (*string*) – output terminal name

Returns instance of `Template`

Return type Template

get_iso_template(*failsafe=True, hybrid=True, terminal='gfxterm', checkiso=False*)

Bootloader configuration template for live ISO media

Parameters

- **failsafe** (*bool*) – with failsafe true|false
- **hybrid** (*bool*) – with hybrid true|false
- **terminal** (*string*) – output terminal name

Returns instance of `Template`

Return type Template

`get_multiboot_install_template`(*failsafe=True*, *terminal='gfxterm'*, *with_timeout=True*)

Bootloader configuration template for install media with hypervisor, e.g Xen dom0

Parameters

- **`failsafe`** (*`bool`*) – with failsafe true|false
- **`terminal`** (*`string`*) – output terminal name

Returns instance of `Template`

Return type Template

`get_multiboot_iso_template`(*failsafe=True*, *terminal='gfxterm'*, *checkiso=False*)

Bootloader configuration template for live ISO media with hypervisor, e.g Xen dom0

Parameters

- **`failsafe`** (*`bool`*) – with failsafe true|false
- **`terminal`** (*`string`*) – output terminal name

Returns instance of `Template`

Return type Template

13.7.3 `kiwi.bootloader.template.isolinux` Module

class `kiwi.bootloader.template.isolinux.`**`BootLoaderTemplateIsoLinux`**

Bases: `object`

isolinux configuraton file templates

`get_install_message_template`()

Bootloader template for text message file in install mode. isolinux displays this as menu if no graphics mode can be initialized

Returns instance of `Template`

Return type Template

`get_install_template`(*failsafe=True*, *with_theme=True*, *terminal=None*, *with_timeout=True*)

Bootloader configuration template for install media

Parameters

- **`failsafe`** (*`bool`*) – with failsafe true|false
- **`with_theme`** (*`bool`*) – with graphics theme true|false

Returns instance of `Template`

Return type Template

get_message_template()

Bootloader template for text message file. isolinux displays this as menu if no graphics mode can be initialized

Returns instance of `Template`

Return type Template

get_multiboot_install_template(*failsafe=True*, *with_theme=True*, *terminal=None*, *with_timeout=True*)

Bootloader configuration template for install media with hypervisor, e.g Xen dom0

Parameters

- **failsafe** (*bool*) – with failsafe true|false
- **with_theme** (*bool*) – with graphics theme true|false

Returns instance of `Template`

Return type Template

get_multiboot_template(*failsafe=True*, *with_theme=True*, *terminal=None*, *checkiso=False*)

Bootloader configuration template for live media with hypervisor, e.g Xen dom0

Parameters

- **failsafe** (*bool*) – with failsafe true|false
- **with_theme** (*bool*) – with graphics theme true|false

Returns instance of `Template`

Return type Template

get_template(*failsafe=True*, *with_theme=True*, *terminal=None*, *checkiso=False*)

Bootloader configuration template for live media

Parameters

- **failsafe** (*bool*) – with failsafe true|false
- **with_theme** (*bool*) – with graphics theme true|false

Returns instance of `Template`

Return type Template

13.7.4 Module Contents

13.8 kiwi.boot Package

13.8.1 Module Contents

13.9 kiwi.builder Package

13.9.1 Submodules

13.9.2 `kiwi.builder.archive` Module

class kiwi.builder.archive.**ArchiveBuilder**(*xml_state*, *target_dir*, *root_dir*, *custom_args=None*)

Bases: `object`

Root archive image builder

Parameters

- **xml_state** (*obsject*) – Instance of `XMLState`
- **target_dir** (*str*) – target directory path name
- **root_dir** (*str*) – root directory path name
- **custom_args** (*dict*) – Custom processing arguments defined as hash keys: * xz_options: string of XZ compression parameters

create()

Create a root archive tarball

Build a simple XZ compressed root tarball from the image root tree

Image types which triggers this builder are:

- image="tbz"

Returns result

Return type instance of `Result`

13.9.3 `kiwi.builder.container` Module

class `kiwi.builder.container.`**`ContainerBuilder`**(*xml_state*, *target_dir*, *root_dir*, *custom_args=None*)

Bases: `object`

Container image builder

Parameters

- **`xml_state`** (`object`) – Instance of `XMLState`
- **`target_dir`** (`str`) – target directory path name
- **`root_dir`** (`str`) – root directory path name
- **`custom_args`** (`dict`) – Custom processing arguments defined as hash keys: * xz_options: string of XZ compression parameters

`create`()

Builds a container image which is usually a data archive including container specific metadata.

Image types which triggers this builder are:

- image="docker"
- image="oci"
- image="appx"

Returns result

Return type instance of `Result`

13.9.4 `kiwi.builder.disk` Module

class `kiwi.builder.disk.`**`DiskBuilder`**(*xml_state*, *target_dir*, *root_dir*, *custom_args=None*)

Bases: `object`

Disk image builder

Parameters

- **`xml_state`** (`object`) – Instance of `XMLState`
- **`target_dir`** (`str`) – Target directory path name
- **`root_dir`** (`str`) – Root directory path name
- **`custom_args`** (`dict`) – Custom processing arguments defined as hash keys: * signing_keys: list of package signing keys * xz_options: string of XZ compression parameters

append_unpartitioned_space()
Extends the raw disk if an unpartitioned area is specified

create()
Build a bootable disk image and optional installation image The installation image is a bootable hybrid ISO image which embeds the disk image and an image installer

Image types which triggers this builder are:

- image="oem"
- image="vmx"

Returns result

Return type instance of `Result`

create_disk()
Build a bootable raw disk image

Raises

- ***KiwiInstallMediaError*** – if install media is required and image type is not oem
- ***KiwiVolumeManagerSetupError*** – root overlay at the same time volumes are defined is not supported

Returns result

Return type instance of `Result`

create_disk_format(*result_instance*)
Create a bootable disk format from a previously created raw disk image

Parameters **result_instance** (*object*) – instance of `Result`

Returns updated result_instance

Return type instance of `Result`

create_install_media(*result_instance*)
Build an installation image. The installation image is a bootable hybrid ISO image which embeds the raw disk image and an image installer

Parameters **result_instance** (*object*) – instance of `Result`

Returns updated result_instance with installation media

Return type instance of `Result`

13.9.5 `kiwi.builder.filesystem` Module

class kiwi.builder.filesystem.**FileSystemBuilder**(*xml_state*, *target_dir*, *root_dir*, *custom_args=None*)

Bases: `object`

Filesystem image builder

Parameters

- **xml_state** (*obsject*) – Instance of `XMLState`
- **target_dir** (*str*) – target directory path name
- **root_dir** (*str*) – root directory path name
- **custom_args** (*dict*) – Custom processing arguments defined as hash keys: * None

create()

Build a mountable filesystem image

Image types which triggers this builder are:

- image="ext2"
- image="ext3"
- image="ext4"
- image="btrfs"
- image="xfs"

Returns result

Return type instance of `Result`

13.9.6 `kiwi.builder.install` Module

class kiwi.builder.install.**InstallImageBuilder**(*xml_state*, *root_dir*, *target_dir*, *boot_image_task*, *custom_args=None*)

Bases: `object`

Installation image builder

Parameters

- **xml_state** (*object*) – instance of `XMLState`

- **root_dir** (*str*) – system image root directory
- **target_dir** (*str*) – target directory path name
- **boot_image_task** (*object*) – instance of `BootImage`
- **custom_args** (*dict*) – Custom processing arguments defined as hash keys: * xz_options: string of XZ compression parameters

create_install_iso()

Create an install ISO from the disk_image as hybrid ISO bootable via legacy BIOS, EFI and as disk from Stick

Image types which triggers this builder are:

- installiso=”true|false”
- installstick=”true|false”

create_install_pxe_archive()

Create an oem install tar archive suitable for installing a disk image via the network using the PXE boot protocol. The archive contains:

- The raw system image xz compressed
- The raw system image checksum metadata file
- The append file template for the boot server
- The system image initrd for kexec
- The install initrd
- The kernel

Image types which triggers this builder are:

- installpxe=”true|false”

13.9.7 `kiwi.builder.live` Module

class kiwi.builder.live.**LiveImageBuilder**(*xml_state*, *target_dir*, *root_dir*, *custom_args=None*)

Bases: `object`

Live image builder

Parameters

- **xml_state** (*object*) – instance of `XMLState`
- **target_dir** (*str*) – target directory path name
- **root_dir** (*str*) – root directory path name
- **custom_args** (*dict*) – Custom processing arguments

create()

Build a bootable hybrid live ISO image

Image types which triggers this builder are:

- image="iso"

Raises *KiwiLiveBootImageError* – if no kernel or hipervisor is found in boot image tree

Returns result

Return type instance of `Result`

13.9.8 `kiwi.builder.kis` Module

class `kiwi.builder.kis.`**KisBuilder**(*xml_state*, *target_dir*, *root_dir*, *custom_args=None*)

Bases: `object`

Filesystem based image builder.

Parameters

- **xml_state** (*object*) – instance of `XMLState`
- **target_dir** (*str*) – target directory path name
- **root_dir** (*str*) – system image root directory
- **custom_args** (*dict*) – Custom processing arguments defined as hash keys: * signing_keys: list of package signing keys * xz_options: string of XZ compression parameters

create()

Build a component image consisting out of a boot image(initrd) plus its appropriate kernel files and the root filesystem image with a checksum.

Image types which triggers this builder are:

- image="kis"
- image="pxe"

Raises *KiwiKisBootImageError* – if no kernel or hipervisor is found in boot image tree

Returns result

Return type instance of `Result`

13.9.9 Module Contents

class kiwi.builder.**ImageBuilder**

Bases: object

Image builder factory

static new(*xml_state: object*, *target_dir: str*, *root_dir: str*, *custom_args: Dict = None*)

13.10 kiwi.container Package

13.10.1 Submodules

13.10.2 `kiwi.container.oci` Module

class kiwi.container.oci.**ContainerImageOCI**(*root_dir*, *transport*, *custom_args=None*)

Bases: object

Create oci container from a root directory

Parameters

- **root_dir** (*string*) – root directory path name
- **custom_args** (*dict*) –

Custom processing arguments defined as hash keys:

Example

```
{
    'container_name': 'name',
    'container_tag': '1.0',
    'additional_tags': ['current', 'foobar'],
    'entry_command': ['/bin/bash', '-x'],
    'entry_subcommand': ['ls', '-l'],
    'maintainer': 'tux',
    'user': 'root',
    'workingdir': '/root',
    'expose_ports': ['80', '42'],
    'volumes': ['/var/log', '/tmp'],
    'environment': {'PATH': '/bin'},
    'labels': {'name': 'value'},
    'history': {
        'created_by': 'some explanation here',
        'comment': 'some comment here',
        'author': 'tux'
    }
}
```

create (*filename*, *base_image*)
Create compressed oci system container tar archive

Parameters

- **filename** (*string*) – archive file name
- **base_image** (*string*) – archive used as a base image

13.10.3 Module Contents

class kiwi.container.**ContainerImage**
Bases: object

Container Image factory

Parameters

- **name** (*string*) – container system name
- **root_dir** (*string*) – root directory path name
- **custom_args** (*dict*) – custom arguments

static new (*name: str*, *root_dir: str*, *custom_args: Dict = None*)

13.11 kiwi.container.setup Package

13.11.1 Submodules

13.11.2 `kiwi.container.setup.base` Module

class kiwi.container.setup.base.**ContainerSetupBase** (*root_dir*, *cus-tom_args=None*)
Bases: object

Base class for setting up the root system to create a container image from for e.g docker. The methods here are generic to linux systems following the FHS standard and modern enough e.g based on systemd

Attributes

- **root_dir** root directory path name
- **custom_args** dict of custom arguments

deactivate_bootloader_setup ()
Container bootloader setup

Tell the system there is no bootloader configuration it needs to care for. A container does not boot

deactivate_root_filesystem_check()

Container filesystem check setup

The root filesystem of a container could be an overlay or a mapped device. In any case it should not be checked for consistency as this is should be done by the container infrastructure

deactivate_systemd_service(*name*)

Container system services setup

Init systems among others also controls services which starts at boot time. A container does not really boot. Thus some services needs to be deactivated

Parameters **name** (*string*) – systemd service name

get_container_name()

Container name

Returns name

Return type str

post_init(*custom_args*)

Post initialization method

Implementation in specialized container setup class

Parameters **custom_args** (*list*) – unused

setup()

Setup container metadata

Implementation in specialized bootloader class required

setup_root_console()

Container console setup

/dev/console should be allowed to login by root

setup_static_device_nodes()

Container device node setup

Without subsystems like udev running in a container it is required to provide a set of device nodes to let the system in the container function correctly. This is done by syncing the host system nodes to the container. That this will also create device nodes which are not necessarily present in the container later is a know limitation of this method and considered harmless

13.11.3 `kiwi.container.setup.docker` Module

class `kiwi.container.setup.docker.`**`ContainerSetupDocker`**(*root_dir*, *custom_args=None*)

Bases: `kiwi.container.setup.oci.ContainerSetupOCI`

Docker container setup

13.11.4 Module Contents

class `kiwi.container.setup.`**`ContainerSetup`**

Bases: `object`

container setup factory

static new(*name: str*, *root_dir: str*, *custom_args: Dict = None*)

13.12 kiwi.filesystem Package

13.12.1 Submodules

13.12.2 `kiwi.filesystem.base` Module

class `kiwi.filesystem.base.`**`FileSystemBase`**(*device_provider: kiwi.storage.device_provider.DeviceProvider*, *root_dir: str = None*, *custom_args: Dict = None*)

Bases: `object`

Implements base class for filesystem interface

Parameters

- **device_provider** (*object*) – Instance of a class based on DeviceProvider required for filesystems which needs a block device for creation. In most cases the DeviceProvider is a LoopDevice
- **root_dir** (*string*) – root directory path name
- **custom_args** (*dict*) – custom filesystem arguments

create_on_device(*label: str = None*)

Create filesystem on block device

Implement in specialized filesystem class for filesystems which requires a block device for creation, e.g ext4.

Parameters **label** (*string*) – label name

create_on_file (*filename: str*, *label: str = None*, *exclude: List[str] = None*)
Create filesystem from root data tree

Implement in specialized filesystem class for filesystems which requires a data tree for creation, e.g squashfs.

Parameters

- **filename** (*string*) – result file path name
- **label** (*string*) – label name
- **exclude** (*list*) – list of exclude dirs/files

get_mountpoint () → Optional[str]
Provides mount point directory

Effective use of the directory is guaranteed only after sync_data

Returns directory path name

Return type string

post_init (*custom_args: Dict*)
Post initialization method

Store dictionary of custom arguments if not empty. This overrides the default custom argument hash

Parameters **custom_args** (*dict*) – custom arguments

```
{
    'create_options': ['option'],
    'mount_options': ['option'],
    'meta_data': {
        'key': 'value'
    }
}
```

sync_data (*exclude: List[str] = None*)
Copy root data tree into filesystem

Parameters **exclude** (*list*) – list of exclude dirs/files

umount ()
Umounts the filesystem in case it is mounted, does nothing otherwise

13.12.3 `kiwi.filesystem.btrfs` Module

class kiwi.filesystem.btrfs.**FileSystemBtrfs**(*device_provider: kiwi.storage.device_provider.DeviceProvid*
root_dir: str = None, custom_args: Dict = None)

Bases: `kiwi.filesystem.base.FileSystemBase`

Implements creation of btrfs filesystem

create_on_device(*label: str = None*)

Create btrfs filesystem on block device

Parameters **label** (`string`) – label name

13.12.4 `kiwi.filesystem.clicfs` Module

class kiwi.filesystem.clicfs.**FileSystemClicFs**(*device_provider: kiwi.storage.device_provider.DeviceProv*
root_dir: str = None, custom_args: Dict = None)

Bases: `kiwi.filesystem.base.FileSystemBase`

Implements creation of clicfs filesystem

create_on_file(*filename: str, label: str = None, exclude: List[str] = None*)

Create clicfs filesystem from data tree

There is no label which could be set for clicfs thus this parameter is not used

There is no option to exclude data from clicfs thus this parameter is not used

Parameters

- **filename** (`string`) – result file path name
- **label** (`string`) – unused
- **exclude** (`list`) – unused

post_init(*custom_args: Dict = None*)

Post initialization method

Initialize temporary container_dir directory to store clicfs embeded filesystem

Parameters **custom_args** (`dict`) – unused

13.12.5 `kiwi.filesystem.ext2` Module

class `kiwi.filesystem.ext2.`**`FileSystemExt2`**(*device_provider: kiwi.storage.device_provider.DeviceProvider, root_dir: str = None, custom_args: Dict = None*)

Bases: `kiwi.filesystem.base.FileSystemBase`

Implements creation of ext2 filesystem

`create_on_device`(*label: str = None*)

Create ext2 filesystem on block device

Parameters **`label`** (`string`) – label name

13.12.6 `kiwi.filesystem.ext3` Module

class `kiwi.filesystem.ext3.`**`FileSystemExt3`**(*device_provider: kiwi.storage.device_provider.DeviceProvider, root_dir: str = None, custom_args: Dict = None*)

Bases: `kiwi.filesystem.base.FileSystemBase`

Implements creation of ext3 filesystem

`create_on_device`(*label: str = None*)

Create ext3 filesystem on block device

Parameters **`label`** (`string`) – label name

13.12.7 `kiwi.filesystem.ext4` Module

class `kiwi.filesystem.ext4.`**`FileSystemExt4`**(*device_provider: kiwi.storage.device_provider.DeviceProvider, root_dir: str = None, custom_args: Dict = None*)

Bases: `kiwi.filesystem.base.FileSystemBase`

Implements creation of ext4 filesystem

`create_on_device`(*label: str = None*)

Create ext4 filesystem on block device

Parameters **`label`** (`string`) – label name

13.12.8 `kiwi.filesystem.fat16` Module

class kiwi.filesystem.fat16.**FileSystemFat16**(*device_provider: kiwi.storage.device_provider.DeviceProvid*
root_dir: str = None, custom_args: Dict = None)

Bases: *kiwi.filesystem.base.FileSystemBase*

Implements creation of fat16 filesystem

create_on_device(*label: str = None*)
Create fat16 filesystem on block device

Parameters **label** (*string*) – label name

13.12.9 `kiwi.filesystem.fat32` Module

class kiwi.filesystem.fat32.**FileSystemFat32**(*device_provider: kiwi.storage.device_provider.DeviceProvid*
root_dir: str = None, custom_args: Dict = None)

Bases: *kiwi.filesystem.base.FileSystemBase*

Implements creation of fat16 filesystem

create_on_device(*label: str = None*)
Create fat32 filesystem on block device

Parameters **label** (*string*) – label name

13.12.10 `kiwi.filesystem.isofs` Module

class kiwi.filesystem.isofs.**FileSystemIsoFs**(*device_provider: kiwi.storage.device_provider.DeviceProvid*
root_dir: str = None, custom_args: Dict = None)

Bases: *kiwi.filesystem.base.FileSystemBase*

Implements creation of iso filesystem

create_on_file(*filename: str, label: str = None, exclude: List[str] = None*)
Create iso filesystem from data tree

There is no label which could be set for iso filesystem thus this parameter is not used

Parameters

- **filename** (*string*) – result file path name

- **label** (*string*) – unused
- **exclude** (*list*) – unused

13.12.11 `kiwi.filesystem.setup` Module

class kiwi.filesystem.setup.**FileSystemSetup**(*xml_state: kiwi.xml_state.XMLState, root_dir: str*)

Bases: object

Implement filesystem setup methods

Methods from this class provides information from the root directory required before building a filesystem image

Parameters

- **xml_state** (*object*) – Instance of XMLState
- **root_dir** (*string*) – root directory path

get_size_mbytes(*filesystem: str = None*) → int

Precalculate the requires size in mbytes to store all data from the root directory in the requested filesystem. Return the configured value if present, if not return the calculated result

Parameters **filesystem** (*string*) – name

Returns mbytes

Return type int

13.12.12 `kiwi.filesystem.squashfs` Module

class kiwi.filesystem.squashfs.**FileSystemSquashFs**(*device_provider: kiwi.storage.device_provider.Devi root_dir: str = None, custom_args: Dict = None*)

Bases: *kiwi.filesystem.base.FileSystemBase*

Implements creation of squashfs filesystem

create_on_file(*filename, label: str = None, exclude: List[str] = None*)

Create squashfs filesystem from data tree

There is no label which could be set for squashfs thus this parameter is not used

Parameters

- **filename** (*string*) – result file path name

- **label** (*string*) – unused
- **exclude** (*list*) – list of exclude dirs/files

13.12.13 `kiwi.filesystem.xfs` Module

class `kiwi.filesystem.xfs.`**FileSystemXfs**(*device_provider: kiwi.storage.device_provider.DeviceProvider, root_dir: str = None, custom_args: Dict = None*)

Bases: `kiwi.filesystem.base.FileSystemBase`

Implements creation of xfs filesystem

create_on_device(*label: str = None*)

Create xfs filesystem on block device

Parameters **label** (*string*) – label name

13.12.14 Module Contents

class `kiwi.filesystem.`**FileSystem**

Bases: `object`

FileSystem factory

Parameters

- **name** (*string*) – filesystem name
- **device_provider** (*object*) – Instance of DeviceProvider
- **root_dir** (*string*) – root directory path name
- **custom_args** (*dict*) – dict of custom filesystem arguments

static new(*name: str, device_provider: object, root_dir: str = None, custom_args: Dict = None*)

13.13 kiwi.iso_tools Package

13.13.1 Submodules

13.13.2 `kiwi.iso_tools.base` Module

class `kiwi.iso_tools.base.`**IsoToolsBase**(*source_dir*)

Bases: `object`

Base Class for Parameter API for iso creation tools

Parameters

- **source_dir** (*string*) – data source dir, usually root_dir
- **boot_path** (*str*) – architecture specific boot path on the ISO
- **iso_parameters** (*str*) – list of ISO creation parameters
- **iso_loaders** (*str*) – list of ISO loaders to embed

add_efi_loader_parameters()
Add ISO creation parameters to embed the EFI loader

Implementation in specialized tool class

create_iso(*filename*, *hidden_files=None*)
Create iso file

Implementation in specialized tool class

Parameters

- **filename** (*str*) – unused
- **hidden_files** (*list*) – unused

get_tool_name()
Return caller name for iso creation tool

Implementation in specialized tool class

Returns tool name

Return type str

has_iso_hybrid_capability()
Indicate if the iso tool has the capability to embed a partition table into the iso such that it can be used as both; an iso and a disk

Implementation in specialized tool class

init_iso_creation_parameters(*custom_args=None*)
Create a set of standard parameters for the main isolinux loader

Implementation in specialized tool class

Parameters **custom_args** (*list*) – unused

list_iso(*isofile*)
List contents of an ISO image

Parameters **isofile** (*str*) – unused

static setup_media_loader_directory(*lookup_path*, *media_path*, *boot_theme*)

13.13.3 `kiwi.iso_tools.cdrtools` Module

class `kiwi.iso_tools.cdrtools.`**`IsoToolsCdrTools`**(*source_dir*)

Bases: *`kiwi.iso_tools.base.IsoToolsBase`*

cdrkit/cdrtools wrapper class

Implementation of Parameter API for iso creation tools using the cdrkit/cdrtools projects. Addressed here are the option compatible tools mkisofs and genisoimage

`add_efi_loader_parameters`()

Add ISO creation parameters to embed the EFI loader

In order to boot the ISO from EFI, the EFI binary is added as alternative loader to the ISO creation parameter list. The EFI binary must be included into a fat filesystem in order to become recognized by the firmware. For details about this file refer to _create_embedded_fat_efi_image() from bootloader/config/grub2.py

`create_iso`(*filename*, *hidden_files=None*)

Creates the iso file with the given filename using cdrtools

Parameters

- **`filename`** (*`str`*) – output filename
- **`hidden_files`** (*`list`*) – list of hidden files

`get_tool_name`()

There are tools by J.Schilling and tools from the community Depending on what is installed a decision needs to be made. mkisofs is preferred over genisoimage

Raises *`KiwiIsoToolError`* – if no iso creation tool is found

Returns tool name

Return type str

`has_iso_hybrid_capability`()

Indicate if the iso tool has the capability to embed a partition table into the iso such that it can be used as both; an iso and a disk

Returns True or False

Return type bool

`init_iso_creation_parameters`(*custom_args=None*)

Create a set of standard parameters

Parameters **`custom_args`** (*`list`*) – custom ISO creation args

`list_iso`(*isofile*)

List contents of an ISO image

Parameters **`isofile`** (*`str`*) – path to the ISO file

Returns formatted isoinfo result

Return type dict

13.13.4 `kiwi.iso_tools.iso` Module

class `kiwi.iso_tools.iso.`**`Iso`**(*source_dir*)

Bases: `object`

Implements helper methods around the creation of ISO filesystems

Parameters

- **`header_id`** (*`str`*) – static identifier string for self written headers
- **`header_end_name`** (*`str`*) – file name to store the header_id to
- **`header_end_file`** (*`str`*) – full file path for the header_end_name file
- **`boot_path`** (*`str`*) – architecture specific boot path on the ISO

`create_header_end_block`(*isofile*)

Find offset address of file containing the header_id and replace it by a list of 2k blocks in range 0 - offset + 1 This is the required preparation to support hybrid ISO images, meaning to let isohybrid work correctly

Parameters **`isofile`** (*`string`*) – path to the ISO file

Raises ***`KiwiIsoLoaderError`*** – if the header_id file is not found

Returns 512 byte blocks offset address

Return type int

`create_header_end_marker`()

Prepare iso file to become a hybrid iso image.

To do this the offest address of the end of the first iso block is required. To lookup this address a reference(marker) file named 'header_end' is created and will show up as last file in the block.

static **`create_hybrid`**(*offset*, *mbrid*, *isofile*, *efi_mode=False*)

Create hybrid ISO

A hybrid ISO embeds both, an isolinux signature as well as a disk signature. kiwi always adds an msdos and a GPT table for the disk signatures

Parameters

- **`offset`** (*`str`*) – hex offset
- **`mbrid`** (*`str`*) – boot record id
- **`isofile`** (*`str`*) – path to the ISO file
- **`efi_mode`** (*`bool`*) – sets the iso to support efi firmware or not

static **`fix_boot_catalog`**(*isofile*)

Fixup inconsistencies in boot catalog

Make sure all catalog entries are in correct order and provide complete metadata information e.g catalog name

Parameters **isofile** (*str*) – path to the ISO file

static relocate_boot_catalog (*isofile*)

Move ISO boot catalog to the standardized place

Check location of the boot catalog and move it to the place where all BIOS and firwmare implementations expects it

Parameters **isofile** (*str*) – path to the ISO file

static set_media_tag (*isofile*)

Include checksum tag in the ISO so it can be verified with the mediacheck program.

Parameters **isofile** (*str*) – path to the ISO file

setup_isolinux_boot_path ()

Write the base boot path into the isolinux loader binary

Raises ***KiwiIsoLoaderError*** – if loader/isolinux.bin is not found

13.13.5 Module Contents

class kiwi.iso_tools.**IsoTools**

Bases: object

IsoTools factory

static new (*source_dir*)

13.14 kiwi.package_manager Package

13.14.1 Submodules

13.14.2 kiwi.package_manager.base Module

class kiwi.package_manager.base.**PackageManagerBase** (*repository*, *custom_args=None*)

Bases: object

Implements base class for installation/deletion of packages and collections using a package manager

Parameters

- **repository** (*object*) – instance of Repository
- **root_dir** (*str*) – root directory path name
- **package_requests** (*list*) – list of packages to install or delete
- **collection_requests** (*list*) – list of collections to install

- **product_requests** (*list*) – list of products to install

clean_leftovers()
Cleans package manager related data not needed in the resulting image such as custom macros

Implementation in specialized package manager class

cleanup_requests()
Cleanup request queues

database_consistent()
OBSOLETE: Will be removed 2019-06-05

dump_reload_package_database(*version=45*)
OBSOLETE: Will be removed 2019-06-05

has_failed(*returncode*)
Evaluate given result return code

Any returncode != 0 is considered an error unless overwritten in specialized package manager class

Parameters **returncode** (*int*) – return code number

Returns True|False

Return type boolean

match_package_deleted(*package_list*, *log_line*)
Match expression to indicate a package has been deleted

Implementation in specialized package manager class

Parameters

- **package_list** (*list*) – unused
- **log_line** (*str*) – unused

match_package_installed(*package_list*, *log_line*)
Match expression to indicate a package has been installed

Implementation in specialized package manager class

Parameters

- **package_list** (*list*) – unused
- **log_line** (*str*) – unused

post_init(*custom_args=None*)
Post initialization method

Implementation in specialized package manager class

Parameters **custom_args** (*list*) – unused

post_process_install_requests_bootstrap(*root_bind=None*)
Process extra code required after bootstrapping

Implementation in specialized package manager class

process_delete_requests(*force=False*)
Process package delete requests (chroot)

Implementation in specialized package manager class

Parameters **force** (*bool*) – unused

process_install_requests()
Process package install requests for image phase (chroot)

Implementation in specialized package manager class

process_install_requests_bootstrap(*root_bind=None*)
Process package install requests for bootstrap phase (no chroot)

Implementation in specialized package manager class

process_only_required()
Setup package processing only for required packages

Implementation in specialized package manager class

process_plus_recommended()
Setup package processing to also include recommended dependencies

Implementation in specialized package manager class

request_collection(*name*)
Queue a package collection

Implementation in specialized package manager class

Parameters **name** (*str*) – unused

request_package(*name*)
Queue a package request

Implementation in specialized package manager class

Parameters **name** (*str*) – unused

request_package_exclusion(*name*)
Queue a package exclusion(skip) request

Implementation in specialized package manager class

Parameters **name** (*str*) – unused

request_package_lock(*name*)
Queue a package exclusion(skip) request

OBSOLETE: Will be removed 2019-06-05

Kept for API compatbility Method calls: request_package_exclusion

request_product(*name*)
Queue a product request

Implementation in specialized package manager class

Parameters **name** (*str*) – unused

update()

Process package update requests (chroot)

Implementation in specialized package manager class

13.14.3 `kiwi.package_manager.dnf` Module

class kiwi.package_manager.dnf.**PackageManagerDnf**(*repository*, *custom_args=None*)

Bases: *kiwi.package_manager.base.PackageManagerBase*

Implements base class for installation/deletion of packages and collections using dnf

Parameters

- **dnf_args** (*doct*) – dnf arguments from repository runtime configuration
- **command_env** (*dict*) – dnf command environment from repository runtime configuration

clean_leftovers()

Cleans package manager related data not needed in the resulting image such as custom macros

match_package_deleted(*package_name*, *dnf_output*)

Match expression to indicate a package has been deleted

Parameters

- **package_list** (*list*) – list of all packages
- **log_line** (*str*) – dnf status line

Returns match or None if there isn't any match

Return type match object, None

match_package_installed(*package_name*, *dnf_output*)

Match expression to indicate a package has been installed

This match for the package to be installed in the output of the dnf command is not 100% accurate. There might be false positives due to sub package names starting with the same base package name

Parameters

- **package_list** (*list*) – list of all packages
- **log_line** (*str*) – dnf status line

Returns match or None if there isn't any match

Return type match object, None

`post_init`(*custom_args=None*)
Post initialization method

Parameters **custom_args** (*list*) – custom dnf arguments

`post_process_install_requests_bootstrap`(*root_bind=None*)
Move the rpm database to the place as it is expected by the rpm package installed during bootstrap phase

Parameters **root_bind** (*object*) – unused

`process_delete_requests`(*force=False*)
Process package delete requests (chroot)

Parameters **force** (*bool*) – force deletion: true|false

Raises *KiwiRequestError* – if none of the packages to delete is installed.

Returns process results in command type

Return type namedtuple

`process_install_requests`()
Process package install requests for image phase (chroot)

Returns process results in command type

Return type namedtuple

`process_install_requests_bootstrap`(*root_bind=None*)
Process package install requests for bootstrap phase (no chroot)

Parameters **root_bind** (*object*) – unused

Returns process results in command type

Return type namedtuple

`process_only_required`()
Setup package processing only for required packages

`process_plus_recommended`()
Setup package processing to also include recommended dependencies.

`request_collection`(*name*)
Queue a collection request

Parameters **name** (*str*) – dnf group name

`request_package`(*name*)
Queue a package request

Parameters **name** (*str*) – package name

request_package_exclusion(*name*)
Queue a package exclusion(skip) request

Parameters **name** (*str*) – package name

request_product(*name*)
Queue a product request

There is no product definition in the fedora repo data

Parameters **name** (*str*) – unused

update()
Process package update requests (chroot)

Returns process results in command type

Return type namedtuple

13.14.4 kiwi.package_manager.zypper Module

class kiwi.package_manager.zypper.**PackageManagerZypper**(*repository, custom_args=None*)
Bases: *kiwi.package_manager.base.PackageManagerBase*

Implements base class for installation/deletion of packages and collections using zypper

Parameters

- **zypper_args** (*list*) – zypper arguments from repository runtime configuration
- **command_env** (*dict*) – zypper command environment from repository runtime configuration

clean_leftovers()
Cleans package manager related data not needed in the resulting image such as custom macros

has_failed(*returncode*)
Evaluate given result return code

In zypper any return code == 0 or >= 100 is considered success. Any return code different from 0 and < 100 is treated as an error we care for. Return codes >= 100 indicates an issue like ‘new kernel needs reboot of the system’ or similar which we don’t care in the scope of image building

Parameters **returncode** (*int*) – return code number

Returns True|False

Return type boolean

match_package_deleted(*package_name*, *zypper_output*)
Match expression to indicate a package has been deleted

Parameters

- **package_list** (*list*) – list of all packages
- **log_line** (*str*) – zypper status line

Returns match or None if there isn't any match

Return type match object, None

match_package_installed(*package_name*, *zypper_output*)
Match expression to indicate a package has been installed

This match for the package to be installed in the output of the zypper command is not 100% accurate. There might be false positives due to sub package names starting with the same base package name

Parameters

- **package_list** (*list*) – list of all packages
- **log_line** (*str*) – zypper status line

Returns match or None if there isn't any match

Return type match object, None

post_init(*custom_args=None*)
Post initialization method

Store custom zypper arguments

Parameters **custom_args** (*list*) – custom zypper arguments

post_process_install_requests_bootstrap(*root_bind=None*)
Move the rpm database to the place as it is expected by the rpm package installed during bootstrap phase

Parameters **root_bind** (*object*) – unused

process_delete_requests(*force=False*)
Process package delete requests (chroot)

Parameters **force** (*bool*) – force deletion: true|false

Raises ***KiwiRequestError*** – if none of the packages to delete is installed

Returns process results in command type

Return type namedtuple

process_install_requests()
Process package install requests for image phase (chroot)

Returns process results in command type

Return type namedtuple

`process_install_requests_bootstrap`(*root_bind=None*)
Process package install requests for bootstrap phase (no chroot)

Parameters **`root_bind`** (*`object`*) – unused

Returns process results in command type

Return type namedtuple

`process_only_required`()
Setup package processing only for required packages

`process_plus_recommended`()
Setup package processing to also include recommended dependencies.

`request_collection`(*name*)
Queue a collection request

Parameters **`name`** (*`str`*) – zypper pattern name

`request_package`(*name*)
Queue a package request

Parameters **`name`** (*`str`*) – package name

`request_package_exclusion`(*name*)
Queue a package exclusion(skip) request

Parameters **`name`** (*`str`*) – package name

`request_product`(*name*)
Queue a product request

Parameters **`name`** (*`str`*) – zypper product name

`update`()
Process package update requests (chroot)

Returns process results in command type

Return type namedtuple

13.14.5 Module Contents

class `kiwi.package_manager.`**`PackageManager`**
Bases: `object`

Package manager factory

Parameters

- **`repository`** (*`object`*) – instance of `Repository`
- **`package_manager`** (*`str`*) – package manager name
- **`custom_args`** (*`list`*) – custom package manager arguments list

Raises *KiwiPackageManagerSetupError* – if the requested package manager type is not supported

Returns package manager

Return type PackageManagerBase subclass

static new(*repository: object*, *package_manager_name: str*, *custom_args: List = None*)

13.15 kiwi.partitioner Package

13.15.1 Submodules

13.15.2 `kiwi.partitioner.base` Module

class kiwi.partitioner.base.**PartitionerBase**(*disk_provider*, *start_sector=None*)

Bases: `object`

Base class for partitioners

Parameters

- **disk_provider** (*object*) – Instance of DeviceProvider
- **start_sector** (*int*) – sector number

create(*name*, *mbsize*, *type_name*, *flags=None*)

Create partition

Implementation in specialized partitioner class

Parameters

- **name** (*string*) – unused
- **mbsize** (*int*) – unused
- **type_name** (*string*) – unused
- **flags** (*list*) – unused

get_id()

Current partition number

Zero indicates no partition has been created so far

Returns partition number

Return type int

post_init()

Post initialization method

Implementation in specialized partitioner class

resize_table(*entries=None*)
Resize partition table

Parameters entries (*int*) – unused

set_flag(*partition_id, flag_name*)
Set partition flag

Implementation in specialized partitioner class

Parameters

- **partition_id** (*int*) – unused
- **flag_name** (*string*) – unused

set_hybrid_mbr()
Turn partition table into hybrid table if supported

Implementation in specialized partitioner class

set_mbr()
Turn partition table into MBR (msdos table)

Implementation in specialized partitioner class

13.15.3 kiwi.partitioner.dasd Module

class kiwi.partitioner.dasd.**PartitionerDasd**(*disk_provider, start_sector=None*)
Bases: *kiwi.partitioner.base.PartitionerBase*

Implements DASD partition setup

create(*name, mbsize, type_name, flags=None*)
Create DASD partition

Parameters

- **name** (*string*) – partition name
- **mbsize** (*int*) – partition size
- **type_name** (*string*) – unused
- **flags** (*list*) – unused

post_init()
Post initialization method

Setup fdasd partition type/flag map

resize_table(*entries=None*)
Resize partition table

Nothing to be done here for DASD devices

Parameters entries (*int*) – unused

13.15.4 `kiwi.partitioner.gpt` Module

class kiwi.partitioner.gpt.**PartitionerGpt**(*disk_provider*, *start_sector=None*)

Bases: *kiwi.partitioner.base.PartitionerBase*

Implements GPT partition setup

create(*name*, *mbsize*, *type_name*, *flags=None*)

Create GPT partition

Parameters

- **name** (*string*) – partition name
- **mbsize** (*int*) – partition size
- **type_name** (*string*) – partition type
- **flags** (*list*) – additional flags

post_init()

Post initialization method

Setup gdisk partition type/flag map

resize_table(*entries=128*)

Resize partition table

Parameters **entries** (*int*) – number of default entries

set_flag(*partition_id*, *flag_name*)

Set GPT partition flag

Parameters

- **partition_id** (*int*) – partition number
- **flag_name** (*string*) – name from flag map

set_hybrid_mbr()

Turn partition table into hybrid GPT/MBR table

set_mbr()

Turn partition table into MBR (msdos table)

13.15.5 `kiwi.partitioner.msdos` Module

class kiwi.partitioner.msdos.**PartitionerMsDos**(*disk_provider*, *start_sector=None*)

Bases: *kiwi.partitioner.base.PartitionerBase*

Implement old style msdos partition setup

create(*name*, *mbsize*, *type_name*, *flags=None*)

Create msdos partition

Parameters

- **name** (*string*) – partition name
- **mbsize** (*int*) – partition size
- **type_name** (*string*) – partition type
- **flags** (*list*) – additional flags

post_init()

Post initialization method

Setup sfdisk partition type/flag map

resize_table(*entries=None*)

Resize partition table

Nothing to be done here for msdos table

Parameters **entries** (*int*) – unused

set_flag(*partition_id*, *flag_name*)

Set msdos partition flag

Parameters

- **partition_id** (*int*) – partition number
- **flag_name** (*string*) – name from flag map

13.15.6 Module Contents

class kiwi.partitioner.**Partitioner**

Bases: object

Partitioner factory

Parameters

- **table_type** (*string*) – Table type name
- **storage_provider** (*object*) – Instance of class based on DeviceProvider
- **start_sector** (*int*) – sector number

static new(*table_type: str*, *storage_provider: object*, *start_sector: int = None*)

13.16 kiwi.repository Package

13.16.1 Submodules

13.16.2 `kiwi.repository.base` Module

class `kiwi.repository.base.`**`RepositoryBase`**(*root_bind*, *custom_args=None*)

Bases: `object`

Implements base class for package manager repository handling

Attributes

Parameters

- **`root_bind`** (*`object`*) – instance of `RootBind`
- **`root_dir`** (*`str`*) – root directory path name
- **`shared_location`** (*`str`*) – shared directory between image root and build system root

`add_repo`(*name*, *uri*, *repo_type*, *prio*, *dist*, *components*, *user*, *secret*, *credentials_file*, *repo_gpgcheck*, *pkg_gpgcheck*, *sourcetype*, *use_for_bootstrap=False*)

Add repository

Implementation in specialized repository class

Parameters

- **`name`** (*`str`*) – unused
- **`uri`** (*`str`*) – unused
- **`repo_type`** – unused
- **`prio`** (*`int`*) – unused
- **`dist`** (*`str`*) – unused
- **`components`** (*`str`*) – unused
- **`user`** (*`str`*) – unused
- **`secret`** (*`str`*) – unused
- **`credentials_file`** (*`str`*) – unused
- **`repo_gpgcheck`** (*`bool`*) – unused
- **`pkg_gpgcheck`** (*`bool`*) – unused
- **`sourcetype`** (*`str`*) – unused
- **`use_for_bootstrap`** (*`bool`*) – unused

cleanup_unused_repos()
Cleanup/Delete unused repositories

Only configured repositories according to the image configuration are allowed to be active when building

Implementation in specialized repository class

delete_all_repos()
Delete all repositories

Implementation in specialized repository class

delete_repo(*name*)
Delete repository

Implementation in specialized repository class

Parameters **name** (*str*) – unused

delete_repo_cache(*name*)
Delete repository cache

Implementation in specialized repository class

Parameters **name** (*str*) – unused

import_trusted_keys(*signing_keys*)
Imports trusted keys into the image

Implementation in specialized repository class

Parameters **signing_keys** (*list*) – list of the key files to import

post_init(*custom_args*)
Post initialization method

Implementation in specialized repository class

Parameters **custom_args** (*list*) – unused

runtime_config()
Repository runtime configuration and environment

Implementation in specialized repository class

setup_package_database_configuration()
Setup package database configuration

Implementation in specialized repository class

use_default_location()
Call repository operations with default repository manager setup

Implementation in specialized repository class

13.16.3 `kiwi.repository.dnf` Module

class `kiwi.repository.dnf.`**RepositoryDnf**(*root_bind*, *custom_args=None*)

Bases: *kiwi.repository.base.RepositoryBase*

Implements repository handling for dnf package manager

Parameters

- **shared_dnf_dir** (*str*) – shared directory between image root and build system root
- **runtime_dnf_config_file** (*str*) – dnf runtime config file name
- **command_env** (*dict*) – customized os.environ for dnf
- **runtime_dnf_config** (*str*) – instance of `ConfigParser`

add_repo(*name*, *uri*, *repo_type='rpm-md'*, *prio=None*, *dist=None*, *components=None*, *user=None*, *secret=None*, *credentials_file=None*, *repo_gpgcheck=None*, *pkg_gpgcheck=None*, *sourcetype=None*, *use_for_bootstrap=False*)

Add dnf repository

Parameters

- **name** (*str*) – repository base file name
- **uri** (*str*) – repository URI
- **repo_type** – repostory type name
- **prio** (*int*) – dnf repostory priority
- **dist** (*str*) – unused
- **components** (*str*) – unused
- **user** (*str*) – unused
- **secret** (*str*) – unused
- **credentials_file** (*str*) – unused
- **repo_gpgcheck** (*bool*) – enable repository signature validation
- **pkg_gpgcheck** (*bool*) – enable package signature validation
- **sourcetype** (*str*) – source type, one of 'baseurl', 'metalink' or 'mirrorlist'
- **use_for_bootstrap** (*bool*) – unused

cleanup_unused_repos()

Delete unused dnf repositories

Repository configurations which are not used for this build must be removed otherwise they are taken into account for the package installations

delete_all_repos()
Delete all dnf repositories

delete_repo(*name*)
Delete dnf repository

Parameters **name** (*str*) – repository base file name

delete_repo_cache(*name*)
Delete dnf repository cache

The cache data for each repository is stored in a directory and additional files all starting with the repository name. The method glob deletes all files and directories matching the repository name followed by any characters to cleanup the cache information

Parameters **name** (*str*) – repository name

import_trusted_keys(*signing_keys*)
Imports trusted keys into the image

Parameters **signing_keys** (*list*) – list of the key files to import

post_init(*custom_args=None*)
Post initialization method

Store custom dnf arguments and create runtime configuration and environment

Parameters **custom_args** (*list*) – dnf arguments

runtime_config()
dnf runtime configuration and environment

Returns dnf_args:list, command_env:dict

Return type dict

setup_package_database_configuration()
Setup rpm macros for bootstrapping and image building

1. Create the rpm image macro which persists during the build
2. Create the rpm bootstrap macro to make sure for bootstrapping the rpm database location matches the host rpm database setup. This macro only persists during the bootstrap phase. If the image was already bootstrapped a compat link is created instead.

use_default_location()
Setup dnf repository operations to store all data in the default places

13.16.4 `kiwi.repository.zypper` Module

class `kiwi.repository.zypper.`**`RepositoryZypper`**(*root_bind*, *custom_args=None*)

Bases: *`kiwi.repository.base.RepositoryBase`*

Implements repo handling for zypper package manager

Parameters

- **`shared_zypper_dir`** (*`str`*) – shared directory between image root and build system root
- **`runtime_zypper_config_file`** (*`str`*) – zypper runtime config file name
- **`runtime_zypp_config_file`** (*`str`*) – libzypp runtime config file name
- **`zypper_args`** (*`list`*) – zypper caller args plus additional custom args
- **`command_env`** (*`dict`*) – customized os.environ for zypper
- **`runtime_zypper_config`** (*`object`*) – instance of `ConfigParser`

`add_repo`(*name*, *uri*, *repo_type='rpm-md'*, *prio=None*, *dist=None*, *components=None*, *user=None*, *secret=None*, *credentials_file=None*, *repo_gpgcheck=None*, *pkg_gpgcheck=None*, *sourcetype=None*, *use_for_bootstrap=False*)

Add zypper repository

Parameters

- **`name`** (*`str`*) – repository name
- **`uri`** (*`str`*) – repository URI
- **`repo_type`** – repostory type name
- **`prio`** (*`int`*) – zypper repostory priority
- **`dist`** (*`str`*) – unused
- **`components`** (*`str`*) – unused
- **`user`** (*`str`*) – credentials username
- **`secret`** (*`str`*) – credentials password
- **`credentials_file`** (*`str`*) – zypper credentials file
- **`repo_gpgcheck`** (*`bool`*) – enable repository signature validation
- **`pkg_gpgcheck`** (*`bool`*) – enable package signature validation
- **`sourcetype`** (*`str`*) – unused
- **`use_for_bootstrap`** (*`boot`*) – unused

cleanup_unused_repos()

Delete unused zypper repositories

zypper creates a system solvable which is unwanted for the purpose of building images. In addition zypper fails with an error message 'Failed to cache rpm database' if such a system solvable exists and a new root system is created

All other repository configurations which are not used for this build must be removed too, otherwise they are taken into account for the package installations

delete_all_repos()

Delete all zypper repositories

delete_repo(*name*)

Delete zypper repository

Parameters **name** (*str*) – repository name

delete_repo_cache(*name*)

Delete zypper repository cache

The cache data for each repository is stored in a list of directories of the same name as the repository name. The method deletes these directories to cleanup the cache information

Parameters **name** (*str*) – repository name

import_trusted_keys(*signing_keys*)

Imports trusted keys into the image

Parameters **signing_keys** (*list*) – list of the key files to import

post_init(*custom_args=None*)

Post initialization method

Store custom zypper arguments and create runtime configuration and environment

Parameters **custom_args** (*list*) – zypper arguments

runtime_config()

zypper runtime configuration and environment

setup_package_database_configuration()

Setup rpm macros for bootstrapping and image building

1. Create the rpm image macro which persists during the build
2. Create the rpm bootstrap macro to make sure for bootstrapping the rpm database location matches the host rpm database setup. This macro only persists during the bootstrap phase. If the image was already bootstrapped a compat link is created instead.
3. Create zypper compat link

use_default_location()

Setup zypper repository operations to store all data in the default places

13.16.5 Module Contents

class kiwi.repository.**Repository**

Bases: `object`

Repository factory

Parameters

- **root_bind** (*object*) – instance of RootBind
- **package_manager** (*str*) – package manager name
- **custom_args** (*list*) – list of custom package manager arguments to setup the repository

Raises ***KiwiRepositorySetupError*** – if package_manager is not supported

static new(*root_bind: object*, *package_manager: str*, *custom_args: List = None*)

13.17 kiwi.repository.template Package

13.17.1 Submodules

13.17.2 `kiwi.repository.template.apt` Module

class kiwi.repository.template.apt.**PackageManagerTemplateAptGet**

Bases: `object`

apt-get configuration file template

get_host_template(*exclude_docs=False*)

apt-get package manager template for apt-get called outside of the image, not chrooted

Return type Template

get_image_template(*exclude_docs=False*)

apt-get package manager template for apt-get called inside of the image, chrooted

Return type Template

13.17.3 Module Contents

13.18 kiwi.solver.repository Package

13.18.1 Submodules

13.18.2 `kiwi.solver.repository.base` Module

class `kiwi.solver.repository.base.`**`SolverRepositoryBase`**(*uri*, *user=None*, *secret=None*)

Bases: `object`

Base class interface for SAT solvable creation.

Parameters

- **`uri`** (*`object`*) – Instance of `Uri`
- **`user`** (*`string`*) – User name for uri authentication
- **`secret`** (*`string`*) – Secret token for uri authentication

`create_repository_solvable`(*target_dir='/var/tmp/kiwi/satsolver'*)

Create SAT solvable for this repository from previously created intermediate solvables by merge and store the result solvable in the specified target_dir

Parameters **`target_dir`** (*`str`*) – path name

Returns file path to solvable

Return type str

`download_from_repository`(*repo_source*, *target*)

Download given source file from the repository and store it as target file

The repo_source location is used relative to the repository location and will be part of a mime type source like: `file://repo_path/repo_source`

Parameters

- **`repo_source`** (*`str`*) – source file in the repo
- **`target`** (*`str`*) – file path

Raises ***`KiwiUriOpenError`*** – if the download fails

`get_repo_type`()

`is_uptodate`(*target_dir='/var/tmp/kiwi/satsolver'*)

Check if repository metadata is up to date

Returns True or False

Return type bool

timestamp()

Return repository timestamp

The retrieval of the repository timestamp depends on the type of the repository and is therefore supposed to be implemented in the specialized Solver Repository classes. If no such implementation exists the method returns the value 'static' to indicate there is no timestamp information available.

Return type str

class kiwi.solver.repository.rpm_md.**SolverRepositoryRpmMd**(*uri*, *user=None*, *secret=None*)

Bases: *kiwi.solver.repository.base.SolverRepositoryBase*

Class for SAT solvable creation for rpm-md type repositories.

timestamp()

Get timestamp from the first primary metadata

Returns time value as text

Return type str

class kiwi.solver.repository.rpm_dir.**SolverRepositoryRpmDir**(*uri*, *user=None*, *secret=None*)

Bases: *kiwi.solver.repository.base.SolverRepositoryBase*

Class for SAT solvable creation for rpm_dir type repositories.

class kiwi.solver.repository.suse.**SolverRepositorySUSE**(*uri*, *user=None*, *secret=None*)

Bases: *kiwi.solver.repository.base.SolverRepositoryBase*

Class for SAT solvable creation for SUSE type repositories.

13.18.3 Module Contents

class kiwi.solver.repository.**SolverRepository**

Bases: object

Repository factory for creation of SAT solvables

- **param object uri** Instance of Uri

static new(*uri: kiwi.system.uri.Uri*, *user: str = None*, *secret: str = None*)

13.19 kiwi.solver Package

13.19.1 Submodules

13.19.2 `kiwi.solver.sat` Module

class `kiwi.solver.sat.`**`Sat`**

Bases: `object`

Sat Solver class to run package solver operations

The class uses SUSE's libsolv sat plugin

`add_repository`(*solver_repository*)

Add a repository solvable to the pool. This basically add the required repository metadata which is needed to run a solver operation later.

Parameters **`solver_repository`** (*`object`*) – Instance of `SolverRepository`

`set_dist_type`(*dist*, *arch=None*)

`solve`(*job_names*, *skip_missing=False*, *ignore_recommended=True*)

Solve dependencies for the given job list. The list is allowed to contain element names of the following format:

- name describes a package name
- pattern:name describes a package collection name whose metadata type is called 'pattern' and stored as such in the repository metadata. Usually SUSE repos uses that
- group:name describes a package collection name whose metadata type is called 'group' and stored as such in the repository metadata. Usually RHEL/CentOS/Fedora repos uses that

Parameters

- **`job_names`** (*`list`*) – list of strings
- **`skip_missing`** (*`bool`*) – skip job if not found
- **`ignore_recommended`** (*`bool`*) – do not include recommended packages

Raises ***`KiwiSatSolverJobProblems`*** – if solver reports solving problems

Returns Transaction result information

Return type dict

13.19.3 Module Contents

13.20 kiwi.storage Package

13.20.1 Submodules

13.20.2 `kiwi.storage.device_provider` Module

class `kiwi.storage.device_provider.`**`DeviceProvider`**

Bases: `object`

Base class for any class providing storage devices

`get_byte_size`(*device: str*) → int

Size of device in bytes

Parameters **`device`** (*`string`*) – node name

Returns byte value from blockdev

Return type int

`get_device`()

Representation of device nodes

Could provide one ore more devices representing the storage Implementation in specialized device provider class

`get_uuid`(*device: str*) → str

UUID of device

Parameters **`device`** (*`string`*) – node name

Returns UUID from blkid

Return type str

`is_loop`() → bool

Check if device provider is loop based

By default this is always False and needs an implementation in the the specialized device provider class

Returns True or False

Return type bool

13.20.3 `kiwi.storage.disk` Module

class `kiwi.storage.disk.`**`Disk`**(*table_type: str, storage_provider: kiwi.storage.device_provider.DeviceProvider, start_sector: int = None*)

Bases: *`kiwi.storage.device_provider.DeviceProvider`*

Implements storage disk and partition table setup

Parameters

- **`table_type`** (*`string`*) – Partition table type name
- **`storage_provider`** (*`object`*) – Instance of class based on DeviceProvider
- **`start_sector`** (*`int`*) – sector number

`activate_boot_partition`()

Activate boot partition

Note: not all Partitioner instances supports this

`create_boot_partition`(*mbsize: int*)

Create boot partition

Populates kiwi_BootPart(id)

Parameters **`mbsize`** (*`int`*) – partition size

`create_efi_csm_partition`(*mbsize: int*)

Create EFI bios grub partition

Populates kiwi_BiosGrub(id)

Parameters **`mbsize`** (*`int`*) – partition size

`create_efi_partition`(*mbsize: int*)

Create EFI partition

Populates kiwi_EfiPart(id)

Parameters **`mbsize`** (*`int`*) – partition size

`create_hybrid_mbr`()

Turn partition table into a hybrid GPT/MBR table

Note: only GPT tables supports this

`create_mbr`()

Turn partition table into MBR (msdos table)

Note: only GPT tables supports this

`create_prep_partition`(*mbsize: int*)

Create prep partition

Populates kiwi_PrepPart(id)

Parameters **`mbsize`** (*`int`*) – partition size

create_root_lvm_partition(*mbsize: int*)
Create root partition for use with LVM

Populates kiwi_RootPart(id)

Parameters **mbsize** (`int`) – partition size

create_root_partition(*mbsize: int*)
Create root partition

Populates kiwi_RootPart(id) and kiwi_BootPart(id) if no extra boot partition is requested

Parameters **mbsize** (`int`) – partition size

create_root_raid_partition(*mbsize: int*)
Create root partition for use with MD Raid

Populates kiwi_RootPart(id) and kiwi_RaidPart(id) as well as the default raid device node at boot time which is configured to be kiwi_RaidDev(/dev/mdX)

Parameters **mbsize** (`int`) – partition size

create_root_readonly_partition(*mbsize: int*)
Create root readonly partition for use with overlayfs

Populates kiwi_ReadOnlyPart(id), the partition is meant to contain a squashfs readonly filesystem. The partition size should be the size of the squashfs filesystem in order to avoid wasting disk space

Parameters **mbsize** (`int`) – partition size

create_spare_partition(*mbsize: int*)
Create spare partition for custom use

Populates kiwi_SparePart(id)

Parameters **mbsize** (`int`) – partition size

create_swap_partition(*mbsize: int*)
Create swap partition

Populates kiwi_SwapPart(id)

Parameters **mbsize** (`int`) – partition size

get_device() → Dict[str, *kiwi.storage.mapped_device.MappedDevice*]
Names of partition devices

Note that the mapping requires an explicit map() call

Returns instances of MappedDevice

Return type dict

get_public_partition_id_map() → Dict[str, str]
Populated partition name to number map

is_loop() → bool

Check if storage provider is loop based

The information is taken from the storage provider. If the storage provider is loop based the disk is it too

Returns True or False

Return type bool

map_partitions()

Map/Activate partitions

In order to access the partitions through a device node it is required to map them if the storage provider is loop based

wipe()

Zap (destroy) any GPT and MBR data structures if present For DASD disks create a new VTOC table

13.20.4 `kiwi.storage.loop_device` Module

class kiwi.storage.loop_device.**LoopDevice**(*filename: str, filesize_mbytes: int = None, blocksize_bytes: int = None*)

Bases: *kiwi.storage.device_provider.DeviceProvider*

Create and manage loop device file for block operations

Parameters

- **filename** (*string*) – loop file name to create
- **filesize_mbytes** (*int*) – size of the loop file
- **blocksize_bytes** (*int*) – blocksize used in loop driver

create(*overwrite: bool = True*)

Setup a loop device of the blocksize given in the constructor The file to loop is created with the size specified in the constructor unless an existing one should not be overwritten

Parameters **overwrite** (*bool*) – overwrite existing file to loop

get_device() → Optional[str]

Device node name

Returns device node name

Return type str

is_loop() → bool

Always True

Returns True

Return type bool

13.20.5 `kiwi.storage.luks_device` Module

class `kiwi.storage.luks_device.`**`LuksDevice`**(*storage_provider: kiwi.storage.device_provider.DeviceProvider*)

Bases: *`kiwi.storage.device_provider.DeviceProvider`*

Implements luks setup on a storage device

Parameters **`storage_provider`** (*`object`*) – Instance of class based on DeviceProvider

`create_crypto_luks`(*passphrase: str, os: str = None, options: list = None, keyfile: str = None*)

Create luks device. Please note the passphrase is readable at creation time of this image. Make sure your host system is secure while this process runs

Parameters

- **`passphrase`** (*`string`*) – credentials
- **`os`** (*`string`*) – distribution name to match distribution specific options for cryptsetup
- **`options`** (*`list`*) – further cryptsetup options
- **`keyfile`** (*`string`*) – file path name file path name which contains an alternative key to unlock the luks device

`create_crypttab`(*filename: str*)

Create crypttab, setting the UUID of the storage device

Parameters **`filename`** (*`string`*) – file path name

static **`create_random_keyfile`**(*filename: str*)

Create keyfile with random data

Parameters **`filename`** (*`string`*) – file path name

`get_device`() → Optional[*kiwi.storage.mapped_device.MappedDevice*]

Instance of MappedDevice providing the luks device

Returns mapped luks device

Return type *MappedDevice*

`is_loop`() → bool

Check if storage provider is loop based

Return loop status from base storage provider

Returns True or False

Return type bool

13.20.6 `kiwi.storage.mapped_device` Module

class kiwi.storage.mapped_device.**MappedDevice**(*device: str*, *device_provider: kiwi.storage.device_provider.DeviceProv*

Bases: *kiwi.storage.device_provider.DeviceProvider*

Hold a reference on a single device

Parameters

- **device_provider** (*object*) – Instance of class based on DeviceProvider
- **device** (*string*) – Device node name

get_device() → str

Mapped device node name

Returns device node name

Return type str

is_loop() → bool

Check if storage provider is loop based

Return loop status from base storage provider

Returns True or False

Return type bool

13.20.7 `kiwi.storage.raid_device` Module

class kiwi.storage.raid_device.**RaidDevice**(*storage_provider: kiwi.storage.device_provider.DeviceProvider*)

Bases: *kiwi.storage.device_provider.DeviceProvider*

Implement raid setup on a storage device

Parameters **storage_provider** (*object*) – Instance of class based on DeviceProvider

create_degraded_raid(*raid_level*)

Create a raid array in degraded mode with one device missing. This only works in the raid levels 0(striping) and 1(mirroring)

Parameters **raid_level** (*string*) – raid level name

create_raid_config(*filename: str*) → None

Create mdadm config file from mdadm request

Parameters **filename** (*string*) – config file name

get_device() → Optional[*kiwi.storage.mapped_device.MappedDevice*]

Instance of MappedDevice providing the raid device

Returns mapped raid device

Return type *MappedDevice*

is_loop() → bool

Check if storage provider is loop based

Return loop status from base storage provider

Returns True or False

Return type bool

13.20.8 `kiwi.storage.setup` Module

class kiwi.storage.setup.**DiskSetup**(*xml_state: kiwi.xml_state.XMLState, root_dir: str*)

Bases: `object`

Implements disk setup methods

Methods from this class provides information required before building a disk image

Parameters

- **xml_state** (*object*) – Instance of XMLState
- **root_dir** (*string*) – root directory path name

boot_partition_size() → int

Size of the boot partition in mbytes

Returns boot size mbytes

Return type int

static get_boot_label() → str

Filesystem Label to use for the boot partition

Returns label name

Return type str

get_disksize_mbytes() → int

Precalculate disk size requirements in mbytes

Returns disk size mbytes

Return type int

static get_efi_label() → str

Filesystem Label to use for the EFI partition

Returns label name

Return type str

get_root_label() → str
Filesystem Label to use for the root partition

If not specified in the XML configuration the default root label is set to 'ROOT'

Returns label name

Return type str

need_boot_partition()
Decide if an extra boot partition is needed. This is done with the bootpartition attribute from the type, however if it is not set it depends on some other type configuration parameters if we need a boot partition or not

Returns True or False

Return type bool

13.20.9 Module Contents

13.21 kiwi.storage.subformat Package

13.21.1 Submodules

13.21.2 kiwi.storage.subformat.base Module

class kiwi.storage.subformat.base.**DiskFormatBase**(*xml_state:* kiwi.xml_state.XMLState, *root_dir:* *str*, *target_dir:* *str*, *custom_args:* *dict* *= None*)

Bases: object

Base class to create disk formats from a raw disk image

Parameters

- **xml_state** (*object*) – Instance of XMLState
- **root_dir** (*string*) – root directory path name
- **arch** (*string*) – Defaults.get_platform_name
- **target_dir** (*string*) – target directory path name
- **custom_args** (*dict*) – custom format options dictionary

create_image_format() → None
Create disk format

Implementation in specialized disk format class required

static get_qemu_option_list(*custom_args: dict*) → list
Create list of qemu options from custom_args dict

Parameters **custom_args** (*dict*) – arguments

Returns qemu option list

Return type list

get_target_file_path_for_format(*format_name: str*) → str
Create target file path name for specified format

Parameters **format_name** (*string*) – disk format name

Returns file path name

Return type str

has_raw_disk() → bool
Check if the base raw disk image exists

Returns True or False

Return type bool

post_init(*custom_args: dict*) → None
Post initialization method

Implementation in specialized disk format class if required

Parameters **custom_args** (*dict*) – unused

resize_raw_disk(*size_bytes*, *append=False*)
Resize raw disk image to specified size. If the request would actually shrink the disk an exception is raised. If the disk got changed the method returns True, if the new size is the same as the current size nothing gets resized and the method returns False

Parameters **size** (*int*) – size in bytes

Returns True or False

Return type bool

store_to_result(*result: kiwi.system.result.Result*)
Store result file of the format conversion into the provided result instance.

By default only the converted image file will be stored as compressed file. Subformats which creates additional metadata files or want to use other result flags needs to overwrite this method

Parameters **result** (*object*) – Instance of Result

13.21.3 `kiwi.storage.subformat.gce` Module

class `kiwi.storage.subformat.gce.`**`DiskFormatGce`**(*xml_state: kiwi.xml_state.XMLState, root_dir: str, target_dir: str, custom_args: dict = None*)

Bases: *`kiwi.storage.subformat.base.DiskFormatBase`*

Create GCE - Google Compute Engine image format

`create_image_format`() → None
Create GCE disk format and manifest

`get_target_file_path_for_format`(*format_name*)
Google requires the image name to follow their naming convetion. Therefore it's required to provide a suitable name by overriding the base class method

Parameters **`format_name`** (*`string`*) – gce

Returns file path name

Return type str

`post_init`(*custom_args: dict*) → None
GCE disk format post initialization method

Store disk tag from custom args

Parameters **`custom_args`** (*`dict`*) – custom gce argument dictionary

```
{'--tag': 'billing_code'}
```

`store_to_result`(*result: kiwi.system.result.Result*) → None
Store result file of the gce format conversion into the provided result instance. In this case compression is unwanted because the gce tarball is already created as a compressed archive

Parameters **`result`** (*`object`*) – Instance of Result

13.21.4 `kiwi.storage.subformat.ova` Module

class `kiwi.storage.subformat.ova.`**`DiskFormatOva`**(*xml_state: kiwi.xml_state.XMLState, root_dir: str, target_dir: str, custom_args: dict = None*)

Bases: *`kiwi.storage.subformat.base.DiskFormatBase`*

Create ova disk format, based on vmdk

create_image_format() → None
Create ova disk format using ovftool from https://www.vmware.com/support/developer/ovf

post_init(*custom_args: dict*) → None
vmdk disk format post initialization method

Store qemu options as list from custom args dict

Parameters **custom_args** (*dict*) – custom qemu arguments dictionary

store_to_result(*result:* kiwi.system.result.Result) → None
Store the resulting ova file into the provided result instance.

Parameters **result** (*object*) – Instance of Result

13.21.5 kiwi.storage.subformat.qcow2 Module

class kiwi.storage.subformat.qcow2.**DiskFormatQcow2**(*xml_state:* kiwi.xml_state.XMLState, *root_dir: str*, *target_dir: str*, *custom_args: dict = None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Create qcow2 disk format

create_image_format() → None
Create qcow2 disk format

post_init(*custom_args: dict*) → None
qcow2 disk format post initialization method

Store qemu options as list from custom args dict

Parameters **custom_args** (*dict*) – custom qemu arguments dictionary

store_to_result(*result:* kiwi.system.result.Result) → None
Store result file of the format conversion into the provided result instance.

In case of a qcow2 format we store the result uncompressed Since the format conversion only takes the real bytes into account such that the sparseness of the raw disk will not result in the output format and can be taken one by one

Parameters **result** (*object*) – Instance of Result

13.21.6 `kiwi.storage.subformat.vagrant_base` Module

class kiwi.storage.subformat.vagrant_base.**DiskFormatVagrantBase**(*xml_state: kiwi.xml_stat root_dir: str, target_dir: str, custom_args: dict = None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Base class for creating vagrant boxes.

The documentation of the vagrant box format can be found here: https://www.vagrantup.com/docs/boxes/format.html In a nutshell, a vagrant box is a tar, tar.gz or zip archive of the following:

1. `metadata.json`: A json file that contains the name of the provider and arbitrary additional data (that vagrant doesn't care about).
2. `Vagrantfile`: A Vagrantfile which defines the boxes' MAC address. It can be also used to define other settings of the box, e.g. the method via which the `/vagrant/` directory is shared. This file is either automatically generated by KIWI or we use a file that has been provided by the user (depends on the setting in `vagrantconfig.embebbed_vagrantfile`)
3. The actual virtual disk image: this is provider specific and vagrant simply forwards it to your virtual machine provider.

Required methods/variables that child classes must implement:

- *vagrant_post_init()*

  post initializing method that has to specify the vagrant provider name in `provider` and the box name in `image_format`. Note: new providers also needs to be specified in the schema and the box name needs to be registered to *kiwi.defaults.Defaults.get_disk_format_types()*

- *create_box_img()*

Optional methods:

- *get_additional_metadata()*
- *get_additional_vagrant_config_settings()*

create_box_img(*temp_image_dir: str*) → List[str]

Provider specific image creation step: this function creates the actual box image. It must be implemented by a child class.

create_image_format() → None

Create a vagrant box for any provider. This includes:

- creation of box metadata.json
- creation of box Vagrantfile (either from scratch or by using the user provided Vagrantfile)
- creation of result format tarball from the files created above

get_additional_metadata() → Dict

Provide *create_image_format()* with additional metadata that will be included in `metadata.json`.

The default implementation returns an empty dictionary.

Returns A dictionary that is serializable to JSON

Return type dict

get_additional_vagrant_config_settings() → str

Supply additional configuration settings for vagrant to be included in the resulting box.

This function can be used by child classes to customize the behavior for different providers: the supplied configuration settings get forwarded to `VagrantConfigTemplate.get_template()` as the parameter `custom_settings` and included in the `Vagrantfile`.

The default implementation returns nothing.

Returns additional vagrant settings

Return type str

post_init(*custom_args: Dict[str, kiwi.xml_parse.vagrantconfig]*)

vagrant disk format post initialization method

store vagrantconfig information provided via custom_args

Parameters **custom_args** (*dict*) – Contains instance of xml_parse::vagrantconfig

```
{'vagrantconfig': object}
```

store_to_result(*result:* kiwi.system.result.Result) → None

Store result file of the vagrant format conversion into the provided result instance. In this case compression is unwanted because the box is already created as a compressed tarball

Parameters **result** (*object*) – Instance of Result

vagrant_post_init() → None

Vagrant provider specific post initialization method

Setup vagrant provider and box name. This information must be set by the specialized provider class implementation to make the this base class methods effective

13.21.7 `kiwi.storage.subformat.vagrant_libvirt` Module

class `kiwi.storage.subformat.vagrant_libvirt.`**`DiskFormatVagrantLibVirt`**(*xml_* kiwi *root_* *str*, *tar-get_* *str*, *cus-tom_* *dict* = *None*

Bases: *`kiwi.storage.subformat.vagrant_base.DiskFormatVagrantBase`*

Create a vagrant box for the libvirt provider

`create_box_img`(*temp_image_dir: str*) → List[str]

Creates the qcow2 disk image box for libvirt vagrant provider

Parameters **`temp_image_dir`** (*`str`*) – Path to the temporary directory used to build the box image

Returns A list of files relevant for the libvirt box to be included in the vagrant box

Return type list

`get_additional_metadata`() → Dict

Provide box metadata needed to create the box in vagrant

Returns Returns a dictionary containing the virtual image format and the size of the image.

Return type dict

`get_additional_vagrant_config_settings`() → str

Returns settings for the libvirt provider telling vagrant to use kvm.

Returns ruby code to be evaluated as string

Return type str

`vagrant_post_init`() → None

Vagrant provider specific post initialization method

Setup vagrant provider and box name. This information must be set by the specialized provider class implementation to make the this base class methods effective

13.21.8 `kiwi.storage.subformat.vagrant_virtualbox` Module

class `kiwi.storage.subformat.vagrant_virtualbox.`**`DiskFormatVagrantVirtualB`**

Bases: *`kiwi.storage.subformat.vagrant_base.DiskFormatVagrantBase`*

Create a vagrant box for the virtualbox provider

`create_box_img`(*temp_image_dir: str*) → List[str]

Create the vmdk image for the Virtualbox vagrant provider.

This function creates the vmdk disk image and the ovf file. The latter is created via the class `VirtualboxOvfTemplate`.

Parameters **`temp_image_dir`** (*str*) – Path to the temporary directory used to build the box image

Returns A list of files relevant for the virtualbox box to be included in the vagrant box

Return type list

`get_additional_vagrant_config_settings`() → str

Configure the default shared folder to use rsync when guest additions are not present inside the box.

Returns ruby code to be evaluated as string

Return type str

`vagrant_post_init`() → None

Vagrant provider specific post initialization method

Setup vagrant provider and box name. This information must be set by the specialized provider class implementation to make the this base class methods effective

13.21.9 `kiwi.storage.subformat.vdi` Module

class kiwi.storage.subformat.vdi.**DiskFormatVdi**(*xml_state: kiwi.xml_state.XMLState, root_dir: str, target_dir: str, custom_args: dict = None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Create vdi disk format

create_image_format() → None

Create vdi disk format

post_init(*custom_args: Dict*) → None

vdi disk format post initialization method

Store qemu options as list from custom args dict

Parameters **custom_args** (*dict*) – custom qemu arguments dictionary

13.21.10 `kiwi.storage.subformat.vhd` Module

class kiwi.storage.subformat.vhd.**DiskFormatVhd**(*xml_state: kiwi.xml_state.XMLState, root_dir: str, target_dir: str, custom_args: dict = None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Create vhd disk format

create_image_format() → None

Create vhd disk format

post_init(*custom_args: Dict*) → None

vhd disk format post initialization method

Store qemu options as list from custom args dict

Parameters **custom_args** (*dict*) – custom qemu arguments dictionary

13.21.11 `kiwi.storage.subformat.vhdfixed` Module

class kiwi.storage.subformat.vhdfixed.**DiskFormatVhdFixed**(*xml_state:* kiwi.xml_state.XMLSta *root_dir:* *str*, *target_dir:* *str*, *custom_args:* *dict* = *None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Create vhd image format in fixed subformat

create_image_format() → None
Create vhd fixed disk format

post_init(*custom_args: Dict*) → None
vhd disk format post initialization method

Store qemu options as list from custom args dict Extract disk tag from custom args

Parameters **custom_args** (*dict*) – custom vhdfixed and qemu argument dictionary

```
{'--tag': 'billing_code', '--qemu-opt': 'value'}
```

store_to_result(*result:* kiwi.system.result.Result) → None
Store result file of the vhdfixed format conversion into the provided result instance. In this case compressing the result is preferred as vhdfixed is not a compressed or dynamic format.

Parameters **result** (*object*) – Instance of Result

13.21.12 `kiwi.storage.subformat.vhdx` Module

class kiwi.storage.subformat.vhdx.**DiskFormatVhdx**(*xml_state:* kiwi.xml_state.XMLState, *root_dir:* *str*, *target_dir:* *str*, *custom_args:* *dict* = *None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Create vhdx image format in dynamic subformat

create_image_format() → None
: Create vhdx dynamic disk format

post_init(*custom_args: Dict*) → None
: vhdx disk format post initialization method

  Store qemu options as list from custom args dict

  Parameters **custom_args** (*dict*) – custom qemu arguments dictionary

13.21.13 kiwi.storage.subformat.vmdk Module

class kiwi.storage.subformat.vmdk.**DiskFormatVmdk**(*xml_state: kiwi.xml_state.XMLState, root_dir: str, target_dir: str, custom_args: dict = None*)

Bases: *kiwi.storage.subformat.base.DiskFormatBase*

Create vmdk disk format

create_image_format() → None
: Create vmdk disk format and machine settings file

post_init(*custom_args: Dict*) → None
: vmdk disk format post initialization method

  Store qemu options as list from custom args dict

  Parameters **custom_args** (*dict*) – custom qemu arguments dictionary

store_to_result(*result: kiwi.system.result.Result*) → None
: Store result files of the vmdk format conversion into the provided result instance. This includes the vmdk image file and the VMware settings file

  Parameters **result** (*object*) – Instance of Result

13.21.14 Module Contents

class kiwi.storage.subformat.**DiskFormat**

Bases: object

DiskFormat factory

Parameters

- **name** (*string*) – Format name
- **xml_state** (*object*) – Instance of XMLState

- **root_dir** (*string*) – root directory path name
- **target_dir** (*string*) – target directory path name

abstract static new(*name: str*, *xml_state: kiwi.xml_state.XMLState*, *root_dir: str*, *target_dir: str*)

13.22 kiwi.storage.subformat.template Package

13.22.1 Submodules

13.22.2 `kiwi.storage.subformat.template.vmware_settings` Module

class kiwi.storage.subformat.template.vmware_settings.**VmwareSettingsTempl**

Bases: `object`

VMware machine settings template

get_template(*memory_setup=False*, *cpu_setup=False*, *network_setup=False*, *iso_setup=False*, *disk_controller='ide'*, *iso_controller='ide'*)

VMware machine configuration template

Parameters

- **memory_setup** (*bool*) – with main memory setup true|false
- **cpu_setup** (*bool*) – with number of CPU's setup true|false
- **network_setup** (*bool*) – with network emulation true|false
- **iso_setup** (*bool*) – with CD/DVD drive emulation true|false
- **disk_controller** (*string*) – add disk controller setup to template
- **iso_controller** (*string*) – add CD/DVD controller setup to template
- **network_mac** (*string*) – add static MAC address setup to template
- **network_driver** (*string*) – add network driver setup to template
- **network_connection_type** (*string*) – add connection type to template

Return type Template

13.22.3 kiwi.storage.subformat.template.vagrant_config Module

class kiwi.storage.subformat.template.vagrant_config.**VagrantConfigTemplat**

Bases: object

Generate a Vagrantfile configuration template

This class creates a simple template for the Vagrantfile that is included inside a vagrant box.

The included Vagrantfile carries additional information for vagrant: by default that is nothing, but depending on the provider additional information need to be present. These can be passed via the parameter `custom_settings` to the method *get_template()*.

Example usage:

The default without any additional settings will result in this Vagrantfile:

```
>>> vagrant_config = VagrantConfigTemplate()
>>> print(
...     vagrant_config.get_template()
... )
Vagrant.configure("2") do |config|
end
```

If your provider/box requires additional settings, provide them as follows:

```
>>> extra_settings = dedent('''
... config.vm.hostname = "no-dead-beef"
... config.vm.provider :special do |special|
...   special.secret_settings = "please_work"
... end
... ''').strip()
>>> print(
...     vagrant_config.get_template(extra_settings)
... )
Vagrant.configure("2") do |config|
  config.vm.hostname = "no-dead-beef"
  config.vm.provider :special do |special|
    special.secret_settings = "please_work"
  end
end
```

get_template(*custom_settings=None*)

Return a new template with `custom_settings` included and indented appropriately.

Parameters **custom_settings** (*str*) – String of additional settings that get pasted into the Vagrantfile template. The string is put at the correct indentation level for you, but the internal indentation has to be provided by the caller.

Returns A string with `custom_settings` inserted at the appropriate position. The template has one the variable `mac_address` that must be substituted.

Return type str

13.22.4 `kiwi.storage.subformat.template.virtualbox_ovf` Module

class `kiwi.storage.subformat.template.virtualbox_ovf.`**`VirtualboxOvfTemplat`**

Bases: `object`

Generate a OVF file template for a vagrant virtualbox box

This class provides a template for virtualbox' ovf configuration file that is embedded inside the vagrant box. The template itself was extracted from a vagrant box that was build via packer and from a script provided by Neal Gompa.

`get_template``()`

Return the actual ovf template. The following values must be substituted: - `vm_name`: the name of this VM - `disk_image_capacity`: Size of the virtual disk image in GB - `vm_description`: a description of this VM

13.22.5 Module contents

13.23 kiwi.system Package

13.23.1 Submodules

13.23.2 `kiwi.system.identifier` Module

class `kiwi.system.identifier.`**`SystemIdentifier`**

Bases: `object`

Create a random ID to identify the system

The information is used to create the mbrid file as an example

Parameters **`image_id`** (*`str`*) – hex identifier string

`calculate_id``()` → None

Calculate random hex id

Using 4 tuples of rand in range from 1..0xfe

`get_id``()` → str

Current hex identifier

Returns hex id

Return type str

write (*filename: str*) → None

Write current hex identifier to file

Parameters **filename** (*str*) – file path name

write_to_disk (*device_provider: kiwi.storage.device_provider.DeviceProvider*) → None

Write current hex identifier to MBR at offset 0x1b8 on disk

Parameters **device_provider** (*object*) – Instance based on DeviceProvider

13.23.3 kiwi.system.kernel Module

class kiwi.system.kernel.**Kernel** (*root_dir: str*)

Bases: object

Implementes kernel lookup and extraction from given root tree

Parameters

- **root_dir** (*str*) – root directory path name
- **kernel_names** (*list*) – list of kernel names to search for functions.sh::suseStripKernel() provides a normalized file so that we do not have to search for many different names in this code

copy_kernel (*target_dir: str*, *file_name: str = None*) → None

Copy kernel to specified target

If no file_name is given the target filename is set as kernel-<kernel.version>.kernel

Parameters

- **target_dir** (*str*) – target path name
- **filename** (*str*) – base filename in target

copy_xen_hypervisor (*target_dir: str*, *file_name: str = None*) → None

Copy xen hypervisor to specified target

If no file_name is given the target filename is set as hypervisor-<xen.name>

Parameters

- **target_dir** (*str*) – target path name
- **filename** (*str*) – base filename in target

get_kernel (*raise_on_not_found: bool = False*) → Optional[*kiwi.system.kernel.kernel_type*]

Lookup kernel files and provide filename and version

Parameters **raise_on_not_found** (*bool*) – sets the method to raise an exception if the kernel is not found

Raises *KiwiKernelLookupError* – if raise_on_not_found flag is active and kernel is not found

Returns tuple with filename, kernelname and version

Return type tuple|None

get_xen_hypervisor() → Optional[*kiwi.system.kernel.xen_hypervisor_type*]

Lookup xen hypervisor and provide filename and hypervisor name

Returns tuple with filename and hypervisor name

Return type tuple|None

class kiwi.system.kernel.**kernel_type**(*name*, *filename*, *version*)

Bases: `tuple`

property filename

Alias for field number 1

property name

Alias for field number 0

property version

Alias for field number 2

class kiwi.system.kernel.**xen_hypervisor_type**(*filename*, *name*)

Bases: `tuple`

property filename

Alias for field number 0

property name

Alias for field number 1

13.23.4 kiwi.system.prepare Module

class kiwi.system.prepare.**SystemPrepare**(*xml_state: kiwi.xml_state.XMLState*, *root_dir: str*, *allow_existing: bool = False*)

Bases: `object`

Implements preparation and installation of a new root system

Parameters

- **xml_state** (*object*) – instance of `XMLState`
- **root_dir** (*str*) – Path to new root directory
- **allow_existing** (*bool*) – Allow using an existing root_dir

clean_package_manager_leftovers() → None
This methods cleans some package manager artifacts created at run time such as macros

delete_packages(*manager: kiwi.package_manager.base.PackageManagerBase, packages: List, force: bool = False*) → None
Delete one or more packages using the package manager inside of the new root directory. If the removal is set with `force` flag only listed packages are deleted and any dependency break or leftover is ignored.

Parameters

- **manager** (*object*) – instance of a `PackageManager` subclass
- **packages** (*list*) – package list
- **force** (*bool*) – force deletion true|false

Raises ***KiwiSystemDeletePackagesFailed*** – if installation process fails

install_bootstrap(*manager: kiwi.package_manager.base.PackageManagerBase, plus_packages: List = None*) → None
Install system software using the package manager from the host, also known as bootstrapping

Parameters

- **manager** (*object*) – instance of a `PackageManager` subclass
- **plus_packages** (*list*) – list of additional packages

Raises ***KiwiBootStrapPhaseFailed*** – if the bootstrapping process fails either installing packages or including bootstrap archives

install_packages(*manager: kiwi.package_manager.base.PackageManagerBase, packages: List*) → None
Install one or more packages using the package manager inside of the new root directory

Parameters

- **manager** (*object*) – instance of a `PackageManager` subclass
- **packages** (*list*) – package list

Raises ***KiwiSystemInstallPackagesFailed*** – if installation process fails

install_system(*manager: kiwi.package_manager.base.PackageManagerBase*) → None
Install system software using the package manager inside of the new root directory. This is done via a chroot operation and requires the desired package manager to became installed via the bootstrap phase

Parameters manager (*object*) – instance of a `PackageManager` subclass

Raises ***KiwiInstallPhaseFailed*** – if the install process fails either installing packages or including any archive

pinch_system(*manager:* kiwi.package_manager.base.PackageManagerBase = *None*, *force:* bool = *False*) → None

Delete packages marked for deletion in the XML description. If force param is set to False uninstalls packages marked with `type="uninstall"` if any; if force is set to True deletes packages marked with `type="delete"` if any.

Parameters

- **manager** (*object*) – instance of `PackageManager` subclass
- **force** (*bool*) – Forced deletion True|False

Raises ***KiwiPackagesDeletePhaseFailed*** – if the deletion packages process fails

setup_repositories(*clear_cache:* bool = *False*, *signing_keys:* List[str] = *None*) → kiwi.package_manager.base.PackageManagerBase

Set up repositories for software installation and return a package manager for performing software installation tasks

Parameters

- **clear_cache** (*bool*) – Flag the clear cache before configure anything
- **signing_keys** (*list*) – Keys imported to the package manager

Returns instance of `PackageManager` subclass

Return type *PackageManager*

update_system(*manager:* kiwi.package_manager.base.PackageManagerBase) → None

Install package updates from the used repositories. the process uses the package manager from inside of the new root directory

Parameters **manager** (*object*) – instance of a `PackageManager` subclass

Raises ***KiwiSystemUpdateFailed*** – if packages update fails

13.23.5 `kiwi.system.profile` Module

class kiwi.system.profile.**Profile**(*xml_state:* kiwi.xml_state.XMLState)

Bases: `object`

Create bash readable .profile environment from the XML description

Parameters **xml_state** (*object*) – instance of :class`XMLState`

add(*key:* str, *value:* str) → None

Add key/value pair to profile dictionary

Parameters

- **key** (*str*) – profile key
- **value** (*str*) – profile value

create (*filename: str*) → None
Create bash quoted profile

Parameters **filename** (*str*) – file path name

delete (*key: str*) → None

get_settings () → Dict
Return all profile elements that has a value

13.23.6 kiwi.system.result Module

class kiwi.system.result.**Result** (*xml_state: kiwi.xml_state.XMLState*)
Bases: object

Collect image building results

Parameters

- **result_files** (*dict*) – dict of result files
- **class_version** (*object*) – *Result* class version
- **xml_state** (*object*) – instance of XMLState

add (*key: str, filename: str, use_for_bundle: bool = True, compress: bool = False, shasum: bool = True*) → None
Add result tuple to result_files list

Parameters

- **key** (*str*) – name
- **filename** (*str*) – file path name
- **use_for_bundle** (*bool*) – use when bundling results true|false
- **compress** (*bool*) – compress when bundling true|false
- **shasum** (*bool*) – create shasum when bundling true|false

dump (*filename: str*) → None
Picke dump this instance to a file

Parameters **filename** (*str*) – file path name

Raises ***KiwiResultError*** – if pickle fails to dump *Result* instance

get_results () → Dict[str, *kiwi.system.result.result_file_type*]
Current list of result tuples

static load (*filename: str*) → result_type
Load pickle dumped filename into a Result instance

Parameters **filename** (*str*) – file path name

Raises ***KiwiResultError*** – if filename does not exist or pickle fails to load filename

print_results() → None
Print results human readable

static verify_image_size(*size_limit: int*, *filename: str*) → None
Verifies the given image file does not exceed the size limit. Throws an exception if the limit is exceeded. If the size limit is set to None no verification is done.

Parameters

- **size_limit** (*int*) – The size limit for filename in bytes.
- **filename** (*str*) – File to verify.

Raises ***KiwiResultError*** – if filename exceeds the size limit

class kiwi.system.result.**result_file_type**(*filename*, *use_for_bundle*, *compress*, *shasum*)
Bases: tuple

property compress
Alias for field number 2

property filename
Alias for field number 0

property shasum
Alias for field number 3

property use_for_bundle
Alias for field number 1

13.23.7 kiwi.system.root_bind Module

class kiwi.system.root_bind.**RootBind**(*root_init: kiwi.system.root_init.RootInit*)
Bases: object

Implements binding/copying of host system paths into the new root directory

Parameters

- **root_dir** (*str*) – root directory path name
- **cleanup_files** (*list*) – list of files to cleanup, delete
- **mount_stack** (*list*) – list of mounted directories for cleanup
- **dir_stack** (*list*) – list of directories for cleanup
- **config_files** (*list*) – list of initial config files

- **bind_locations** (*list*) – list of kernel filesystems to bind mount
- **shared_location** (*str*) – shared directory between image root and build system root

cleanup() → None

Cleanup mounted locations, directories and intermediate config files

mount_kernel_file_systems() → None

Bind mount kernel filesystems

Raises ***KiwiMountKernelFileSystemsError*** – if some kernel filesystem fails to mount

mount_shared_directory(*host_dir: str = None*) → None

Bind mount shared location

The shared location is a directory which shares data from the image buildsystem host with the image root system. It is used for the repository setup and the package manager cache to allow chroot operations without being forced to duplicate this data

Parameters **host_dir** (*str*) – directory to share between image root and build system root

Raises ***KiwiMountSharedDirectoryError*** – if mount fails

setup_intermediate_config() → None

Create intermediate config files

Some config files e.g etc/hosts needs to be temporarly copied from the buildsystem host to the image root system in order to allow e.g DNS resolution in the way as it is configured on the buildsystem host. These config files only exists during the image build process and are not part of the final image

Raises ***KiwiSetupIntermediateConfigError*** – if the management of intermediate configuration files fails

umount_kernel_file_systems() → None

Umount kernel filesystems

Raises ***KiwiMountKernelFileSystemsError*** – if some kernel filesystem fails to mount

13.23.8 kiwi.system.root_init Module

class kiwi.system.root_init.**RootInit**(*root_dir: str, allow_existing: bool = False*)

Bases: object

Implements creation of new root directory for a linux system

Host system independent static default files and device nodes are created to initialize a new base system

Parameters **root_dir** (*str*) – root directory path name

create() → None
Create new system root directory

The method creates a temporary directory and initializes it for the purpose of building a system image from it. This includes the following setup:

- create core system paths
- create static core device nodes

On success the contents of the temporary location are synced to the specified root_dir and the temporary location will be deleted. That way we never work on an incomplete initial setup

Raises ***KiwiRootInitCreationError*** – if the init creation fails at some point

delete() → None
Force delete root directory and its contents

13.23.9 `kiwi.system.setup` Module

class kiwi.system.setup.**SystemSetup**(*xml_state: kiwi.xml_state.XMLState*, *root_dir: str*)
Bases: object

Implementation of system setup steps supported by kiwi

Kiwi is not responsible for the system configuration, however some setup steps needs to be performed in order to provide a minimal work environment inside of the image according to the desired image type.

Parameters

- **xml_state** (*object*) – instance of XMLState
- **root_dir** (*str*) – root directory path name

call_config_script() → None
Call config.sh script chrooted

call_disk_script() → None
Call disk.sh script chrooted

call_edit_boot_config_script(*filesystem: str*, *boot_part_id: int*, *working_directory: str = None*) → None
Call configured editbootconfig script _NON_ chrooted

Pass the boot filesystem name and the partition number of the boot partition as parameters to the call

Parameters

- **filesystem** (*str*) – boot filesystem name

- **boot_part_id** (*int*) – boot partition number
- **working_directory** (*str*) – directory name

call_edit_boot_install_script (*diskname: str, boot_device_node: str, working_directory: str = None*) → None
Call configured editbootinstall script _NON_ chrooted

Pass the disk file name and the device node of the boot partition as parameters to the call

Parameters

- **diskname** (*str*) – file path name
- **boot_device_node** (*str*) – boot device node name
- **working_directory** (*str*) – directory name

call_image_script () → None
Call images.sh script chrooted

cleanup () → None
Delete all traces of a kiwi description which are not required in the later image

create_fstab (*fstab: kiwi.utils.fstab.Fstab*) → None
Create etc/fstab from given Fstab object

Custom fstab modifications are possible and handled in the following order:

1. Look for an optional fstab.append file which allows to append custom fstab entries to the final fstab. Once embedded the fstab.append file will be deleted
2. Look for an optional fstab.patch file which allows to patch the current contents of the fstab file with a given patch file. Once patched the fstab.patch file will be deleted
3. Look for an optional fstab.script file which is called chrooted for the purpose of updating the fstab file as appropriate. Note: There is no validation in place that checks if the script actually handles fstab or any other file in the image rootfs. Once called the fstab.script file will be deleted

Parameters **fstab** (*object*) – instance of Fstab

create_init_link_from_linuxrc () → None
kiwi boot images provides the linuxrc script, however the kernel also expects an init executable to be present. This method creates a hard link to the linuxrc file

create_recovery_archive () → None
Create a compressed recovery archive from the root tree for use with kiwi's recvoery system. The method creates additional data into the image root filesystem which is deleted prior to the creation of a new recovery data set

export_modprobe_setup (*target_root_dir: str*) → None
Export etc/modprobe.d to given root_dir

Parameters **target_root_dir** (*str*) – path name

export_package_changes (*target_dir: str*) → Optional[str]
Export image package changelog for comparision of actual changes of the installed packages

Parameters **target_dir** (*str*) – path name

export_package_list (*target_dir: str*) → Optional[str]
Export image package list as metadata reference used by the open buildservice

Parameters **target_dir** (*str*) – path name

export_package_verification (*target_dir: str*) → Optional[str]
Export package verification result as metadata reference used by the open buildservice

Parameters **target_dir** (*str*) – path name

import_cdroot_files (*target_dir: str*) → None
Copy cdroot files from the image description to the specified target directory. Supported is a tar archive named config-cdroot.tar[.compression-postfix]

Parameters **target_dir** (*str*) – directory to unpack archive to

import_description () → None
Import XML descriptions, custom scripts, archives and script helper methods

import_image_identifier () → None
Create etc/ImageID identifier file

import_overlay_files (*follow_links: bool = False*, *preserve_owner_group: bool = False*) → None
Copy overlay files from the image description to the image root tree. Supported are a root/ directory or a root.tar.gz tarball. The root/ directory takes precedence over the tarball.

In addition the method also supports profile specific overlay files which are searched in a directory of the same name as the profile name.

The overall order for including overlay files is as follows:

1. root/ dir or root.tar.gz
2. PROFILE_NAME/ dir(s) in the order of the selected profiles

Parameters

- **follow_links** (*bool*) – follow symlinks true|false
- **preserve_owner_group** (*bool*) – preserve permissions true|false

import_repositories_marked_as_imageinclude () → None
Those <repository> sections which are marked with the imageinclude attribute should be permanently added to the image repository configuration

script_exists(*name: str*) → bool
: Check if provided script base name exists in the image description

    Parameters **name** (*str*) – script base name

set_selinux_file_contexts(*security_context_file: str*) → None
: Initialize the security context fields (extended attributes) on the files matching the security_context_file

    Parameters **security_context_file** (*str*) – path file name

setup_groups() → None
: Add groups for configured users

setup_keyboard_map() → None
: Setup console keyboard

setup_locale() → None
: Setup UTF8 system wide locale

setup_machine_id() → None
: Setup systemd machine id

    Empty out the machine id which was provided by the package installation process. This will instruct the dracut initrd code to create a new machine id. This way a golden image produces unique machine id's on first deployment and boot of the image.

    Note: Requires dracut connected image type

    This method must only be called if the image is of a type which gets booted via a dracut created initrd. Deleting the machine-id without the dracut initrd creating a new one produces an inconsistent system

setup_permissions() → None
: Check and Fix permissions using chkstat

    Call chkstat in system mode which reads /etc/sysconfig/security to determine the configured security level and applies the appropriate permission definitions from the /etc/permissions* files. It's possible to provide those files as overlay files in the image description to apply a certain permission setup when needed. Otherwise the default setup as provided on the package level applies.

    It's required that the image root system has chkstat installed. If not present KIWI skips this step and continuous with a warning.

setup_plymouth_splash() → None
: Setup the KIWI configured splash theme as default

    The method uses the plymouth-set-default-theme tool to setup the theme for the plymouth splash system. Only in case the tool could be found in the image root, it is assumed plymouth splash is in use and the tool is called in a chroot operation

setup_timezone() → None
: Setup timezone symlink

setup_users() → None

Add/Modify configured users

13.23.10 `kiwi.system.shell` Module

class kiwi.system.shell.**Shell**

Bases: `object`

Special character handling for shell evaluated code

static format_to_variable_value(*value: Optional[Any]*) → str

Format given variable value to return a string value representation that can be sourced by shell scripts. If the provided value is not representable as a string (list, dict, tuple etc) an exception is raised

Parameters **value** (*any*) – a python variable

Raises ***KiwiShellVariableValueError*** – if value is an iterable

Returns string value representation

Return type str

static quote(*message: str*) → str

Quote characters which have a special meaning for bash but should be used as normal characters. actually I had planned to use pipes.quote but it does not quote as I had expected it. e.g 'name_wit_a_$' does not quote the $ so we do it on our own for the scope of kiwi

Parameters **message** (*str*) – message text

Returns quoted text

Return type str

static quote_key_value_file(*filename: str*) → List[str]

Quote given input file which has to be of the form key=value to be able to become sourced by the shell

Parameters **filename** (*str*) – file path name

Returns list of quoted text

Return type List[str]

static run_common_function(*name: str, parameters: List[str]*) → None

Run a function implemented in config/functions.sh

Parameters

- **name** (*str*) – function name
- **parameters** (*list*) – function arguments

13.23.11 `kiwi.system.size` Module

class `kiwi.system.size.`**`SystemSize`**(*source_dir: str*)

Bases: `object`

Provide source tree size information

Parameters **`source_dir`** (`str`) – source directory path name

`accumulate_files`() → int

Calculate sum of all files in the source tree

Returns number of files

Return type int

`accumulate_mbyte_file_sizes`(*exclude: List[str] = None*) → int

Calculate data size of all data in the source tree

Parameters **`exclude`** (`list`) – list of paths to exclude

Returns mbytes

Return type int

`customize`(*size: float*, *requested_filesystem: str*) → int

Increase the sum of all file sizes by an empiric factor

Each filesystem has some overhead it needs to manage itself. Thus the plain data size is always smaller as the size of the container which embeds it. This method increases the given size by a filesystem specific empiric factor to ensure the given data size can be stored in a filesystem of the customized size

Parameters

- **`size`** (`float`) – mbsize to update
- **`requested_filesystem`** (`str`) – filesystem name

Returns mbytes

Return type int

13.23.12 `kiwi.system.uri` Module

class `kiwi.system.uri.`**`Uri`**(*uri: str*, *repo_type: str = 'rpm-md'*)

Bases: `object`

Normalize url types

Allow to translate the available KIWI repo source types into standard mime types

Parameters

- **`uri`** (`str`) – URI, remote or local repository location

- **repo_type** (*str*) – repository type name, defaults to 'rpm-md' and is only effectively used when building inside of the open build service which maps local repositories to a specific environment

alias() → str

Create hexdigest from URI as alias

If the repository definition from the XML description does not provide an alias, kiwi creates one for you. However it's better to assign a human readable alias in the XML configuration

Returns alias name as hexdigest

Return type str

credentials_file_name() → str

Filename to store repository credentials

Returns credentials file name

Return type str

get_fragment() → str

Returns the fragment part of the URI.

Returns fragment part of the URI if any, empty string otherwise

Return type str

is_public() → bool

Check if URI is considered to be publicly reachable

Returns True|False

Return type bool

is_remote() → bool

Check if URI is a remote or local location

Returns True|False

Return type bool

translate(*check_build_environment: bool = True*) → str

Translate repository location according to their URI type

Depending on the URI type the provided location needs to be adapted e.g updated by the service URL in case of an open buildservice project name

Raises ***KiwiUriStyleUnknown*** – if the uri scheme can't be detected, is unknown or it is inconsistent with the build environment

Parameters **check_build_environment** (*bool*) – specify if the uri translation should depend on the environment the build is called in. As of today this only effects the translation result if the image build happens inside of the Open Build Service

Returns translated repository location

Return type str

13.23.13 `kiwi.system.users` Module

class `kiwi.system.users.Users`(*root_dir: str*)

Bases: `object`

Operations on users and groups in a root directory

Parameters **root_dir** (*str*) – root directory path name

group_add(*group_name: str, options: List[str]*) → None

Add group with options

Parameters

- **group_name** (*str*) – group name
- **options** (*list*) – groupadd options

group_exists(*group_name: str*) → bool

Check if group exists

Parameters **group_name** (*str*) – group name

Returns True|False

Return type bool

setup_home_for_user(*user_name: str, group_name: str, home_path: str*) → None

Setup user home directory

Parameters

- **user_name** (*str*) – user name
- **group_name** (*str*) – group name
- **home_path** (*str*) – path name

user_add(*user_name: str, options: List[str]*) → None

Add user with options

Parameters

- **user_name** (*str*) – user name
- **options** (*list*) – useradd options

user_exists(*user_name: str*) → bool

Check if user exists

Parameters **user_name** (*str*) – user name

Returns True|False

Return type bool

user_modify(*user_name: str*, *options: List[str]*) → None

Modify user with options

Parameters

- **user_name** (*str*) – user name
- **options** (*list*) – usermod options

13.23.14 Module Contents

13.24 kiwi.tasks package

13.24.1 Submodules

13.24.2 kiwi.tasks.base Module

class kiwi.tasks.base.**CliTask**(*should_perform_task_setup=True*)

Bases: object

Base class for all task classes, loads the task and provides the interface to the command options and the XML description

Attributes

- **should_perform_task_setup** Indicates if the task should perform the setup steps which covers the following task configurations: * setup debug level * setup logfile * setup color output

load_xml_description(*description_directory*)

Load, upgrade, validate XML description

Attributes

- **xml_data** instance of XML data toplevel domain (image), stateless data
- **config_file** used config file path
- **xml_state** Instance of XMLState, stateful data

quadruple_token(*option*)

Helper method for commandline options of the form –option a,b,c,d

Make sure to provide a common result for option values which separates the information in a comma separated list of values

Parameters **option** (*str*) – comma separated option string

Returns common option value representation

Return type list

run_checks(*checks*)

This method runs the given runtime checks excluding the ones disabled in the runtime configuration file.

Parameters **checks** (*dict*) – A dictionary with the runtime method names as keys and their arguments list as the values.

sextuple_token(*option*)

Helper method for commandline options of the form –option a,b,c,d,e,f

Make sure to provide a common result for option values which separates the information in a comma separated list of values

Parameters **option** (*str*) – comma separated option string

Returns common option value representation

Return type list

13.24.3 kiwi.tasks.result_bundle Module

class kiwi.tasks.result_bundle.**ResultBundleTask**(*should_perform_task_setup=True*)

Bases: *kiwi.tasks.base.CliTask*

Implements result bundler

Attributes

- **manual** Instance of Help

process()

Create result bundle from the image build results in the specified target directory. Each result image will contain the specified bundle identifier as part of its filename. Uncompressed image files will also become xz compressed and a sha sum will be created from every result image

13.24.4 kiwi.tasks.result_list Module

class kiwi.tasks.result_list.**ResultListTask**(*should_perform_task_setup=True*)

Bases: *kiwi.tasks.base.CliTask*

Implements result listing

Attributes

- **manual** Instance of Help

process()

List result information from a previous system command

13.24.5 `kiwi.tasks.system_build` Module

class kiwi.tasks.system_build.**SystemBuildTask**(*should_perform_task_setup=True*)

Bases: *kiwi.tasks.base.CliTask*

Implements building of system images

Attributes

- **manual** Instance of Help

process()

Build a system image from the specified description. The build command combines the prepare and create commands

13.24.6 `kiwi.tasks.system_create` Module

class kiwi.tasks.system_create.**SystemCreateTask**(*should_perform_task_setup=True*)

Bases: *kiwi.tasks.base.CliTask*

Implements creation of system images

Attributes

- **manual** Instance of Help

process()

Create a system image from the specified root directory the root directory is the result of a system prepare command

13.24.7 `kiwi.tasks.system_prepare` Module

class kiwi.tasks.system_prepare.**SystemPrepareTask**(*should_perform_task_setup=True*

Bases: *kiwi.tasks.base.CliTask*

Implements preparation and installation of a new root system

Attributes

- **manual** Instance of Help

process()

Prepare and install a new system for chroot access

13.24.8 `kiwi.tasks.system_update` Module

class `kiwi.tasks.system_update.`**`SystemUpdateTask`**(*should_perform_task_setup=True*)

Bases: *`kiwi.tasks.base.CliTask`*

Implements update and maintenance of root systems

Attributes

- **`manual`** Instance of Help

`process`()

Update root system with latest repository updates and optionally allow to add or delete packages. the options to add or delete packages can be used multiple times

13.24.9 Module Contents

13.25 kiwi.utils Package

13.25.1 Submodules

13.25.2 `kiwi.utils.checksum` Module

class `kiwi.utils.block.`**`BlockID`**(*device*)

Bases: `object`

Get information from a block device

Parameters **`device`** (*`str`*) – block device node name name

`get_blkid`(*id_type*)

Retrieve information for specified metadata ID from block device

Parameters **`id_type`** (*`string`*) – metadata ID, see man blkid for details

Returns ID of the block device

Return type str

`get_filesystem`()

Retrieve filesystem type from block device

Returns filesystem type

Return type str

`get_label`()

Retrieve filesystem label from block device

Returns block device label

Return type str

get_uuid()

Retrieve filesystem uuid from block device

Returns uuid for the filesystem of the block device

Return type str

13.25.3 kiwi.utils.block Module

class kiwi.utils.checksum.**Checksum**(*source_filename*)

Bases: object

Manage checksum creation for files

Parameters

- **source_filename** (*str*) – source file name to build checksum for
- **checksum_filename** (*str*) – target file with checksum information

matches(*checksum*, *filename*)

Compare given checksum with reference checksum stored in the provided filename. If the checksum matches the method returns True, or False in case it does not match

Parameters

- **checksum** (*str*) – checksum string to compare
- **filename** (*str*) – filename containing checksum

Returns True or False

Return type bool

md5(*filename=None*)

Create md5 checksum

Parameters **filename** (*str*) – filename for checksum

Returns checksum

Return type str

sha256(*filename=None*)

Create sha256 checksum

Parameters **filename** (*str*) – filename for checksum

13.25.4 `kiwi.utils.compress` Module

class `kiwi.utils.compress.`**`Compress`**(*source_filename*, *keep_source_on_compress=False*)

Bases: `object`

File compression / decompression

Parameters

- **`keep_source`** (*`bool`*) – Request to keep the uncompressed source
- **`source_filename`** (*`str`*) – Source file name to compress
- **`supported_zipper`** (*`list`*) – List of supported compression tools
- **`compressed_filename`** (*`str`*) – Compressed file name path with compression suffix
- **`uncompressed_filename`** (*`str`*) – Uncompressed file name path

`get_format`()

Detect compression format

Returns compression format name or None if it couldn't be inferred

Return type Optional[str]

`gzip`()

Create gzip(max compression) compressed file

`uncompress`(*temporary=False*)

Uncompress with format autodetection

By default the original source file will be changed into the uncompressed variant. If temporary is set to True a temporary file is created instead

Parameters **`temporary`** (*`bool`*) – uncompress to a temporary file

`xz`(*options=None*)

Create XZ compressed file

Parameters **`options`** (*`list`*) – custom xz compression options

13.25.5 `kiwi.utils.sync` Module

class `kiwi.utils.sync.`**`DataSync`**(*source_dir*, *target_dir*)

Bases: `object`

Sync data from a source directory to a target directory using the rsync protocol

Parameters

- **`source_dir`** (*`str`*) – source directory path name

- **target_dir** (*str*) – target directory path name

sync_data (*options=None, exclude=None*)
Sync data from source to target using rsync

Parameters

- **options** (*list*) – rsync options
- **exclude** (*list*) – file patterns to exclude

target_supports_extended_attributes ()
Check if the target directory supports extended filesystem attributes

Returns True or False

Return type bool

13.25.6 kiwi.utils.sysconfig Module

class kiwi.utils.sysconfig.**SysConfig** (*source_file*)
Bases: object

Read and Write sysconfig style files

Parameters **source_file** (*str*) – source file path

get (*key*)

write ()
Write back source file with changed content but in same order

13.25.7 Module Contents

13.26 kiwi.volume_manager Package

13.26.1 Submodules

13.26.2 kiwi.volume_manager.base Module

class kiwi.volume_manager.base.**VolumeManagerBase** (*device_map, root_dir, volumes, custom_args=None*)
Bases: *kiwi.storage.device_provider.DeviceProvider*

Implements base class for volume management interface

Parameters

- **mountpoint** (*str*) – root mountpoint for volumes

- **device_map** (*object*) – dictionary of low level DeviceProvider intances
- **root_dir** (*str*) – root directory path name
- **volumes** (*list*) – list of volumes from `XMLState::get_volumes()`
- **volume_group** (*str*) – volume group name
- **volume_map** (*map*) – map volume name to device node
- **mount_list** (*list*) – list of volume MountManager's
- **device** (*str*) – storage device node name
- **custom_args** (*dict*) – custom volume manager arguments for all volume manager and filesystem specific tasks
- **custom_filesystem_args** (*list*) – custom filesystem creation and mount arguments, subset of the custom_args information suitable to be passed to a FileSystem instance

Raises ***KiwiVolumeManagerSetupError*** – if the given root_dir doesn't exist

apply_attributes_on_volume (*toplevel*, *volume*)

create_volume_paths_in_root_dir ()
Implements creation of volume paths in the given root directory

create_volumes (*filesystem_name*)
Implements creation of volumes

Implementation in specialized volume manager class required

Parameters **filesystem_name** (*str*) – unused

get_canonical_volume_list ()
Implements hierarchical sorting of volumes according to their paths and provides information about the volume configured as the one eating all the rest space

Returns list of canonical_volume_type tuples

Return type list

get_device ()
Return current DeviceProvider dictionary

Returns device_map

Return type dict

get_fstab (*persistency_type*, *filesystem_name*)
Implements setup of the fstab entries. The method should return a list of fstab compatible entries

Parameters

- **persistency_type** (*str*) – unused
- **filesystem_name** (*str*) – unused

get_mountpoint()

Provides mount point directory

Effective use of the directory is guaranteed only after sync_data

Returns directory path name

Return type string

get_volume_mbsize(*volume*, *all_volumes*, *filesystem_name*, *resize_on_boot=False*)

Implements size lookup for the given path and desired filesystem according to the specified size type

Parameters

- **volume** (*tuple*) – volume to check size for
- **all_volumes** (*list*) – list of all volume tuples
- **filesystem_name** (*str*) – filesystem name
- **resize_on_boot** – specify the time of the resize. If the resize happens at boot time the volume size is only the minimum size to just store the data. If the volume size is fixed and does not change at boot time the returned size is the requested size which can be greater than the minimum needed size.

Returns mbsize

Return type int

get_volumes()

Implements return of dictionary of volumes and their mount options

is_loop()

Check if storage provider is loop based

The information is taken from the storage provider. If the storage provider is loop based the volume manager is it too

Returns True of False

Return type bool

mount_volumes()

Implements mounting of all volumes below one master directory

Implementation in specialized volume manager class required

post_init(*custom_args*)

Post initialization method

Implementation in specialized volume manager class if required

Parameters **custom_args** (*dict*) – unused

set_property_readonly_root()
Implements setup of read-only root property

setup(*name=None*)
Implements setup required prior to the creation of volumes

Implementation in specialized volume manager class required

Parameters **name** (*str*) – unused

setup_mountpoint()
Implements creation of a master directory holding the mounts of all volumes

sync_data(*exclude=None*)
Implements sync of root directory to mounted volumes

Parameters **exclude** (*list*) – file patterns to exclude

umount_volumes()
Implements umounting of all volumes

Implementation in specialized volume manager class required

13.26.3 kiwi.volume_manager.btrfs Module

class kiwi.volume_manager.btrfs.**VolumeManagerBtrfs**(*device_map*, *root_dir*, *volumes*, *custom_args=None*)
Bases: *kiwi.volume_manager.base.VolumeManagerBase*

Implements btrfs sub-volume management

Parameters

- **subvol_mount_list** (*list*) – list of mounted btrfs subvolumes
- **toplevel_mount** (*object*) – MountManager for root mountpoint

create_volumes(*filesystem_name*)
Create configured btrfs subvolumes

Any btrfs subvolume is of the same btrfs filesystem. There is no way to have different filesystems per btrfs subvolume. Thus the filesystem_name has no effect for btrfs

Parameters **filesystem_name** (*string*) – unused

get_fstab(*persistency_type='by-label'*, *filesystem_name=None*)
Implements creation of the fstab entries. The method returns a list of fstab compatible entries

Parameters

- **persistency_type** (*string*) – by-label | by-uuid
- **filesystem_name** (*string*) – unused

Returns list of fstab entries

Return type list

get_mountpoint() → str
Provides btrfs root mount point directory

Effective use of the directory is guaranteed only after sync_data

Returns directory path name

Return type string

get_volumes()
Return dict of volumes

Returns volumes dictionary

Return type dict

mount_volumes()
Mount btrfs subvolumes

post_init(*custom_args*)
Post initialization method

Store custom btrfs initialization arguments

Parameters **custom_args** (*list*) – custom btrfs volume manager arguments

set_property_readonly_root()
Sets the root volume to be a readonly filesystem

setup(*name=None*)
Setup btrfs volume management

In case of btrfs a toplevel(@) subvolume is created and marked as default volume. If snapshots are activated via the custom_args the setup method also created the @/.snapshots/1/snapshot subvolumes. There is no concept of a volume manager name, thus the name argument is not used for btrfs

Parameters **name** (*string*) – unused

sync_data(*exclude=None*)
Sync data into btrfs filesystem

If snapshots are activated the root filesystem is synced into the first snapshot

Parameters **exclude** (*list*) – files to exclude from sync

umount_volumes()
Umount btrfs subvolumes

Returns True if all subvolumes are successfully unmounted

Return type bool

13.26.4 `kiwi.volume_manager.lvm` Module

class `kiwi.volume_manager.lvm.`**`VolumeManagerLVM`**(*device_map, root_dir, volumes, custom_args=None*)

Bases: *`kiwi.volume_manager.base.VolumeManagerBase`*

Implements LVM volume management

`create_volumes`(*filesystem_name*)

Create configured lvm volumes and filesystems

All volumes receive the same filesystem

Parameters **`filesystem_name`** (*`str`*) – volumes filesystem name

`get_device`()

Dictionary of MappedDevice instances per volume

Note: The mapping requires an explicit create_volumes() call

Returns root plus volume device map

Return type dict

`get_fstab`(*persistency_type, filesystem_name*)

Implements creation of the fstab entries. The method returns a list of fstab compatible entries

Parameters

- **`persistency_type`** (*`str`*) – unused
- **`filesystem_name`** (*`str`*) – volumes filesystem name

Returns fstab entries

Return type list

`get_volumes`()

Return dict of volumes

Returns volumes dictionary

Return type dict

`mount_volumes`()

Mount lvm volumes

`post_init`(*custom_args*)

Post initialization method

Store custom lvm initialization arguments

Parameters **`custom_args`** (*`list`*) – custom lvm volume manager arguments

`setup`(*volume_group_name='systemVG'*)
Setup lvm volume management

In case of LVM a new volume group is created on a PV initialized storage device

Parameters **`name`** (*`str`*) – volume group name

`umount_volumes`()
Umount lvm volumes

Returns True if all subvolumes are successfully unmounted

Return type bool

13.26.5 Module Contents

`class` `kiwi.volume_manager.`**`VolumeManager`**
Bases: `object`

VolumeManager factory

Parameters

- **`name`** (*`str`*) – volume management name
- **`device_map`** (*`dict`*) – dictionary of low level DeviceProvider intances
- **`root_dir`** (*`str`*) – root directory path name
- **`volumes`** (*`list`*) – list of volumes from `XMLState::get_volumes()`
- **`custom_args`** (*`dict`*) – dictionary of custom volume manager arguments

`static new`(*name: str, device_map: object, root_dir: str, volumes: List, custom_args: Dict = None*)

Links

- GitHub Sources
- GitHub Releases
- RPM Packages
- Build Tests TumbleWeed(x86)
- Build Tests TumbleWeed(arm)
- Build Tests TumbleWeed(s390)
- Build Tests TumbleWeed(ppc64le)

- Build Tests RawHide(x86)
- Build Tests Leap(x86)
- Build Tests Fedora(x86)
- Build Tests Fedora(arm)
- Build Tests Fedora(ppc64le)
- Build Tests CentOS(x86)
- Build Tests Ubuntu(x86)
- Build Tests Debian(x86)
- Build Tests ArchLinux(x86)

CHAPTER

FOURTEEN

THE APPLIANCE CONCEPT

An appliance is a ready to use image of an operating system including a pre-configured application for a specific use case. The appliance is provided as an image file and needs to be deployed to, or activated in the target system or service.

KIWI NG can create appliances in various forms: beside classical installation ISOs and images for virtual machines it can also build images that boot via PXE or Vagrant boxes.

In KIWI NG, the appliance is specified via a collection of human readable files in a directory, also called the `image description`. At least one XML file `config.xml` or `.kiwi` is required. In addition there may as well be other files like scripts or configuration data.

CHAPTER

FIFTEEN

USE CASES

The following list shows a selection of use cases for which an appliance is needed:

Private and Public Clouds Cloud environments are managed through an API provided by the cloud service provider. The classic way to install a machine is not possible in such an environment because there is no physical access to the machine. An appliance is needed to be registered with the cloud

Custom Linux Distribution Linux distributors provides their distribution based on a collection of packages and release them on an install media like a DVD or an USB stick. Typically a lot more software components exists for the distribution which are not part of the default installation media or the installation media comes with software and installation routines that are not matching your target audience. With an appliance made by KIWI NG you can create provide an installation media that matches custom criteria as needed by the customer and does not require extra post processing steps after the default installation method provided by the distributor.

Live Systems The ability to have a Linux OS that runs from a small storage device like a USB stick or a SD card is the swiss army knife of many system administrators. The creation of such a live system includes use of technologies that are not part of a standard installation process. An appliance builder is needed to create this sort of system

Embedded Systems Embedded Systems like the Raspberry Pi comes with limited hardware components. Their boot sequences often does not allow for classic installation methods through USB or DVD devices. Instead they boot through SD card slots or via the network. SoC (System on Chip) devices also tend to implement non standard boot methods which can only be implemented through custom OS appliances.

And More ...

CHAPTER

SIXTEEN

CONTACT

- Mailing list

  The kiwi-images group is an open group and anyone can subscribe, even if you do not have a Google account.

- Matrix

  An open network for secure, decentralized communication. Please find the #kiwi room via Riot on the web or by using the supported clients.

PYTHON MODULE INDEX

INDEX