Pungi Documentation

Release 4.1.39

Daniel Mach

Sep 23, 2019

CONTENTS

1 About Pungi **3**
- 1.1 Tool overview . . . 3
- 1.2 Links . . . 3
- 1.3 Origin of name . . . 4

2 Phases **5**
- 2.1 Init . . . 5
- 2.2 Pkgset . . . 5
- 2.3 Buildinstall . . . 5
- 2.4 Gather . . . 6
- 2.5 ExtraFiles . . . 6
- 2.6 Createrepo . . . 6
- 2.7 OSTree . . . 6
- 2.8 Productimg . . . 6
- 2.9 Createiso . . . 6
- 2.10 ExtraIsos . . . 7
- 2.11 LiveImages, LiveMedia . . . 7
- 2.12 ImageBuild . . . 7
- 2.13 OSBS . . . 7
- 2.14 OSTreeInstaller . . . 7
- 2.15 ImageChecksum . . . 7
- 2.16 Test . . . 7

3 Config file format **9**
- 3.1 Importing other files . . . 9
- 3.2 Formatting strings . . . 10

4 Configuration **11**
- 4.1 Minimal Config Example . . . 11
- 4.2 Release . . . 11
- 4.3 Base Product . . . 12
- 4.4 General Settings . . . 12
- 4.5 Image Naming . . . 14
- 4.6 Signing . . . 16
- 4.7 Git URLs . . . 16
- 4.8 Createrepo Settings . . . 17
- 4.9 Package Set Settings . . . 17
- 4.10 Buildinstall Settings . . . 18
- 4.11 Gather Settings . . . 20
- 4.12 Koji Settings . . . 23

4.13 Runroot "openssh" method settings . . . 24
4.14 Extra Files Settings . . . 25
4.15 Productimg Settings . . . 26
4.16 CreateISO Settings . . . 27
4.17 Automatic generation of version and release . . . 28
4.18 Common options for Live Images, Live Media and Image Build . . . 28
4.19 Live Images Settings . . . 29
4.20 Live Media Settings . . . 29
4.21 Image Build Settings . . . 30
4.22 OSTree Settings . . . 32
4.23 Ostree Installer Settings . . . 33
4.24 OSBS Settings . . . 34
4.25 Extra ISOs . . . 35
4.26 Media Checksums Settings . . . 37
4.27 Translate Paths Settings . . . 37
4.28 Miscellaneous Settings . . . 38

5 Exporting files from SCM **39**
5.1 `file` vs. `dir` . . . 39
5.2 Caveats . . . 39

6 Progress notification **41**
6.1 Setting it up . . . 41

7 Gathering packages **43**
7.1 Variant types . . . 43
7.2 Profiling . . . 44
7.3 Modular compose . . . 44

8 Processing comps files **45**

9 Contributing to Pungi **47**
9.1 Set up development environment . . . 47
9.2 Developing . . . 48
9.3 Testing . . . 50
9.4 Documenting . . . 50

10 Testing Pungi **51**
10.1 Test Data . . . 51
10.2 Unit Tests . . . 51
10.3 Functional Tests . . . 51

11 Managing compose from multiple parts **53**
11.1 General settings . . . 53
11.2 Partial compose settings . . . 54

Contents:

CHAPTER

ONE

ABOUT PUNGI

Pungi is a distribution compose tool.

Composes are release snapshots that contain release deliverables such as:

- installation trees
  - RPMs
  - repodata
  - comps
- (bootable) ISOs
- kickstart trees
  - anaconda images
  - images for PXE boot

1.1 Tool overview

Pungi consists of multiple separate executables backed by a common library.

The main entry-point is the `pungi-koji` script. It loads the compose configuration and kicks off the process. Composing itself is done in phases. Each phase is responsible for generating some artifacts on disk and updating the `compose` object that is threaded through all the phases.

Pungi itself does not actually do that much. Most of the actual work is delegated to separate executables. *Pungi* just makes sure that all the commands are invoked in the appropriate order and with correct arguments. It also moves the artifacts to correct locations.

The executable name `pungi-koji` comes from the fact that most of those separate executables submit tasks to Koji that does the actual work in an auditable way.

However unlike doing everything manually in Koji, Pungi will make sure you are building all images from the same package set, and will produce even deliverables that Koji can not create like YUM repos and installer ISOs.

1.2 Links

- Upstream GIT: https://pagure.io/pungi/
- Issue tracker: https://pagure.io/pungi/issues
- Questions can be asked on *#fedora-releng* IRC channel on FreeNode

1.3 Origin of name

The name *Pungi* comes from the instrument used to charm snakes. *Anaconda* being the software Pungi was manipulating, and anaconda being a snake, led to the referential naming.

The first name, which was suggested by Seth Vidal, was *FIST*, *Fedora Installation <Something> Tool*. That name was quickly discarded and replaced with Pungi.

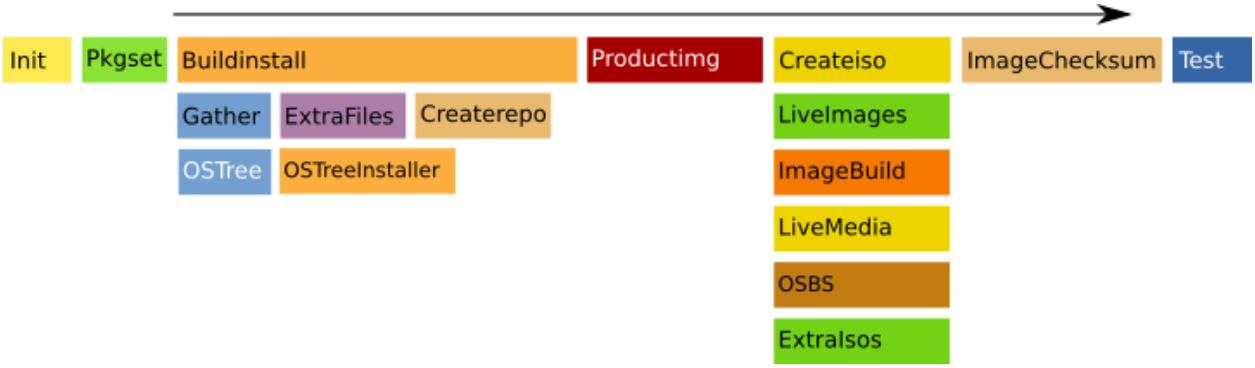
There was also a bit of an inside joke that when said aloud, it could sound like punji, which is a sharpened stick at the bottom of a trap. Kind of like software...

CHAPTER

TWO

PHASES

Each invocation of `pungi-koji` consists of a set of phases.



Most of the phases run sequentially (left-to-right in the diagram), but there are use cases where multiple phases run in parallel. This happens for phases whose main point is to wait for a Koji task to finish.

2.1 Init

The first phase to ever run. Can not be skipped. It prepares the comps files for variants (by filtering out groups and packages that should not be there). See *Processing comps files* for details about how this is done.

2.2 Pkgset

This phase loads a set of packages that should be composed. It has two separate results: it prepares repos with packages in `work/` directory (one per arch) for further processing, and it returns a data structure with mapping of packages to architectures.

2.3 Buildinstall

Spawns a bunch of threads, each of which runs either `lorax` or `buildinstall` command (the latter coming from `anaconda` package). The commands create `boot.iso` and other boot configuration files. The image is finally linked into the `compose/` directory as netinstall media.

The created images are also needed for creating live media or other images in later phases.

With `lorax` this phase runs one task per variant.arch combination. For `buildinstall` command there is only one task per architecture and `product.img` should be used to customize the results.

2.4 Gather

This phase uses data collected by `pkgset` phase and figures out what packages should be in each variant. The basic mapping can come from comps file, a JSON mapping or `additional_packages` config option. This inputs can then be enriched by adding all dependencies. See *Gathering packages* for details.

Once the mapping is finalized, the packages are linked to appropriate places and the `rpms.json` manifest is created.

2.5 ExtraFiles

This phase collects extra files from the configuration and copies them to the compose directory. The files are described by a JSON file in the compose subtree where the files are copied. This metadata is meant to be distributed with the data (on ISO images).

2.6 Createrepo

This phase creates RPM repositories for each variant.arch tree. It is actually reading the `rpms.json` manifest to figure out which packages should be included.

2.7 OSTree

Updates an ostree repository with a new commit with packages from the compose. The repository lives outside of the compose and is updated immediately. If the compose fails in a later stage, the commit will not be reverted.

Implementation wise, this phase runs `rpm-ostree` command in Koji runroot (to allow running on different arches).

2.8 Productimg

Creates `product.img` files for customizing the bootable media created in `buildinstall` phase.

2.9 Createiso

Generates ISO files and accumulates enough metadata to be able to create `image.json` manifest. The file is however not created in this phase, instead it is dumped in the `pungi-koji` script itself.

The files include a repository with all RPMs from the variant. There will be multiple images if the packages do not fit on a single image.

The image will be bootable if `buildinstall` phase is enabled and the packages fit on a single image.

There can also be images with source repositories. These are never bootable.

2.10 Extralsos

This phase is very similar to `createiso`, except it combines content from multiple variants onto a single image. Packages, repodata and extra files from each configured variant are put into a subdirectory. Additional extra files can be put into top level of the image. The image will be bootable if the main variant is bootable.

2.11 LiveImages, LiveMedia

Creates media in Koji with `koji spin-livecd`, `koji spin-appliance` or `koji spin-livemedia` command. When the media are finished, the images are copied into the `compose/` directory and metadata for images is updated.

2.12 ImageBuild

This phase wraps up `koji image-build`. It also updates the metadata ultimately responsible for `images.json` manifest.

2.13 OSBS

This phase builds docker base images in OSBS.

The finished images are available in registry provided by OSBS, but not downloaded directly into the compose. The is metadata about the created image in `compose/metadata/osbs.json`.

2.14 OSTreeInstaller

Creates bootable media that carry an ostree repository as a payload. These images are created by running `lorax` with special templates. Again it runs in Koji runroot.

2.15 ImageChecksum

Responsible for generating checksums for the images. The checksums are stored in image manifest as well as files on disk. The list of images to be processed is obtained from the image manifest. This way all images will get the same checksums irrespective of the phase that created them.

2.16 Test

This phase is supposed to run some sanity checks on the finished compose.

The first test is to run `repoclosure` on each repository. By default errors are only reported in the log, the compose will still be considered a success. The actual error has to be looked up in the compose logs directory. Configuration allows customizing this.

The other test is to check all images listed the metadata and verify that they look sane. For ISO files headers are checked to verify the format is correct, and for bootable media a check is run to verify they have properties that allow booting.

CHAPTER

THREE

CONFIG FILE FORMAT

The configuration file parser is provided by kobo

The file follows a Python-like format. It consists of a sequence of variables that have a value assigned to them.

```
variable = value
```

The variable names must follow the same convention as Python code: start with a letter and consist of letters, digits and underscores only.

The values can be either an integer, float, boolean (`True` or `False`), a string or `None`. Strings must be enclosed in either single or double quotes.

Complex types are supported as well.

A list is enclosed in square brackets and items are separated with commas. There can be a comma after the last item as well.

```
a_list = [1,
          2,
          3,
         ]
```

A tuple works like a list, but is enclosed in parenthesis.

```
a_tuple = (1, "one")
```

A dictionary is wrapped in brackets, and consists of `key: value` pairs separated by commas. The keys can only be formed from basic types (int, float, string).

```
a_dict = {
    'foo': 'bar',
    1: None
}
```

The value assigned to a variable can also be taken from another variable.

```
one = 1
another = one
```

Anything on a line after a # symbol is ignored and functions as a comment.

3.1 Importing other files

It is possible to include another configuration file. The files are looked up relative to the currently processed file.

The general structure of import is:

```
from FILENAME import WHAT
```

The `FILENAME` should be just the base name of the file without extension (which must be `.conf`). `WHAT` can either be a comma separated list of variables or `*`.

```
# Opens constants.conf and brings PI and E into current scope.
from constants import PI, E

# Opens common.conf and brings everything defined in that file into current
# file as well.
from common import *
```

Note: Pungi will copy the configuration file given on command line into the `logs/` directory. Only this single file will be copied, not any included ones. (Copying included files requires a fix in kobo library.)

The JSON-formatted dump of configuration is correct though.

3.2 Formatting strings

String interpolation is available as well. It uses a `%`-encoded format. See Python documentation for more details.

```
joined = "%s %s" % (var_a, var_b)

a_dict = {
    "fst": 1,
    "snd": 2,
}
another = "%(fst)s %(snd)s" % a_dict
```

CHAPTER

FOUR

CONFIGURATION

Please read productmd documentation for terminology and other release and compose related details.

4.1 Minimal Config Example

```
# RELEASE
release_name = "Fedora"
release_short = "Fedora"
release_version = "23"

# GENERAL SETTINGS
comps_file = "comps-f23.xml"
variants_file = "variants-f23.xml"
module_defaults_dir = "module_defaults"

# KOJI
koji_profile = "koji"
runroot = False

# PKGSET
sigkeys = [None]
pkgset_source = "koji"
pkgset_koji_tag = "f23"

# CREATEREPO
createrepo_checksum = "sha256"

# GATHER
gather_method = "deps"
greedy_method = "build"
check_deps = False

# BUILDINSTALL
buildinstall_method = "lorax"
```

4.2 Release

Following **mandatory** options describe a release.

4.2.1 Options

release_name [mandatory] (*str*) – release name

release_short [mandatory] (*str*) – release short name, without spaces and special characters

release_version [mandatory] (*str*) – release version

release_type = "ga" (*str*) – release type, for example `ga`, `updates` or `updates-testing`. See list of all valid values in productmd documentation.

release_internal = False (*bool*) – whether the compose is meant for public consumption

4.2.2 Example

```
release_name = "Fedora"
release_short = "Fedora"
release_version = "23"
# release_type = "ga"
```

4.3 Base Product

Base product options are **optional** and we need to them only if we're composing a layered product built on another (base) product.

4.3.1 Options

base_product_name (*str*) – base product name

base_product_short (*str*) – base product short name, without spaces and special characters

base_product_version (*str*) – base product **major** version

base_product_type = "ga" (*str*) – base product type, "ga", "updates" etc., for full list see documentation of *productmd*.

4.3.2 Example

```
release_name = "RPM Fusion"
release_short = "rf"
release_version = "23.0"

base_product_name = "Fedora"
base_product_short = "Fedora"
base_product_version = "23"
```

4.4 General Settings

4.4.1 Options

comps_file [mandatory] (*scm_dict*, *str* or None) – reference to comps XML file with installation groups

variants_file [mandatory] (*scm_dict* or *str*) – reference to variants XML file that defines release variants and architectures

module_defaults_dir [optional] (*scm_dict* or *str*) – reference the module defaults directory containing modulemd-defaults YAML documents

failable_deliverables [optional] (*list*) – list which deliverables on which variant and architecture can fail and not abort the whole compose. This only applies to `buildinstall` and `iso` parts. All other artifacts can be configured in their respective part of configuration.

Please note that `*` as a wildcard matches all architectures but `src`.

comps_filter_environments [optional] (*bool*) – When set to `False`, the comps files for variants will not have their environments filtered to match the variant.

tree_arches ([*str*]) – list of architectures which should be included; if undefined, all architectures from variants.xml will be included

tree_variants ([*str*]) – list of variants which should be included; if undefined, all variants from variants.xml will be included

repoclosure_strictness (*list*) – variant/arch mapping describing how repoclosure should run. Possible values are

- `off` – do not run repoclosure
- `lenient` – (default) run repoclosure and write results to logs, but detected errors are only reported in logs
- `fatal` – abort compose when any issue is detected

When multiple blocks in the mapping match a variant/arch combination, the last value will win.

repoclosure_backend (*str*) – Select which tool should be used to run repoclosure over created repositories. By default `yum` is used, but you can switch to `dnf`. Please note that when `dnf` is used, the build dependencies check is skipped. On Python 3, only `dnf` backend is available.

compose_type (*str*) – Allows to set default compose type. Type set via a command-line option overwrites this.

4.4.2 Example

```
comps_file = {
    "scm": "git",
    "repo": "https://git.fedorahosted.org/git/comps.git",
    "branch": None,
    "file": "comps-f23.xml.in",
}

variants_file = {
    "scm": "git",
    "repo": "https://pagure.io/pungi-fedora.git ",
    "branch": None,
    "file": "variants-fedora.xml",
}

failable_deliverables = [
    ('^.*$', {
        # Buildinstall can fail on any variant and any arch
        '*': ['buildinstall'],
        'src': ['buildinstall'],
        # Nothing on i386 blocks the compose
```

(continues on next page)

(continued from previous page)

```
        'i386': ['buildinstall', 'iso', 'live'],
    })
]

tree_arches = ["x86_64"]
tree_variants = ["Server"]

repoclosure_strictness = [
    # Make repoclosure failures fatal for compose on all variants ...
    ('^.*$', {'*': 'fatal'}),
    # ... except for Everything where it should not run at all.
    ('^Everything$', {'*': 'off'})
]
```

4.5 Image Naming

Both image name and volume id are generated based on the configuration. Since the volume id is limited to 32 characters, there are more settings available. The process for generating volume id is to get a list of possible formats and try them sequentially until one fits in the length limit. If substitutions are configured, each attempted volume id will be modified by it.

For layered products, the candidate formats are first `image_volid_layered_product_formats` followed by `image_volid_formats`. Otherwise, only `image_volid_formats` are tried.

If no format matches the length limit, an error will be reported and compose aborted.

4.5.1 Options

There a couple common format specifiers available for both the options:

- `compose_id`
- `release_short`
- `version`
- `date`
- `respin`
- `type`
- `type_suffix`
- `label`
- `label_major_version`
- `variant`
- `arch`
- `disc_type`

image_name_format [optional] (*str|dict*) – Python's format string to serve as template for image names. The value can also be a dict mapping variant UID regexes to the format string. The pattern should not overlap, otherwise it is undefined which one will be used.

This format will be used for all phases generating images. Currently that means `createiso`, `live_images` and `buildinstall`.

Available extra keys are:

- `disc_num`
- `suffix`

image_volid_formats [optional] (*list*) – A list of format strings for generating volume id.

The extra available keys are:

- `base_product_short`
- `base_product_version`

image_volid_layered_product_formats [optional] (*list*) – A list of format strings for generating volume id for layered products. The keys available are the same as for `image_volid_formats`.

restricted_volid = False (*bool*) – New versions of lorax replace all non-alphanumerical characters with dashes (underscores are preserved). This option will mimic similar behaviour in Pungi.

volume_id_substitutions [optional] (*dict*) – A mapping of string replacements to shorten the volume id.

disc_types [optional] (*dict*) – A mapping for customizing `disc_type` used in image names.

Available keys are:

- `boot` – for `boot.iso` images created in *buildinstall* phase
- `live` – for images created by *live_images* phase
- `dvd` – for images created by *createiso* phase
- `ostree` – for ostree installer images

Default values are the same as the keys.

4.5.2 Example

```
# Image name respecting Fedora's image naming policy
image_name_format = "%(release_short)s-%(variant)s-%(disc_type)s-%(arch)s-%(version)s
↪%(suffix)s"
# Use the same format for volume id
image_volid_formats = [
    "%(release_short)s-%(variant)s-%(disc_type)s-%(arch)s-%(version)s"
]
# No special handling for layered products, use same format as for regular images
image_volid_layered_product_formats = []
# Replace "Cloud" with "C" in volume id etc.
volume_id_substitutions = {
    'Cloud': 'C',
    'Alpha': 'A',
    'Beta': 'B',
    'TC': 'T',
}

disc_types = {
    'boot': 'netinst',
    'live': 'Live',
    'dvd': 'DVD',
}
```

4.6 Signing

If you want to sign deliverables generated during pungi run like RPM wrapped images. You must provide few configuration options:

signing_command [optional] (*str*) – Command that will be run with a koji build as a single argument. This command must not require any user interaction. If you need to pass a password for a signing key to the command, do this via command line option of the command and use string formatting syntax `%(signing_key_password)s`. (See **signing_key_password_file**).

signing_key_id [optional] (*str*) – ID of the key that will be used for the signing. This ID will be used when crafting koji paths to signed files (`kojipkgs.fedoraproject.org/packages/NAME/VER/REL/data/signed/KEYID/..`).

signing_key_password_file [optional] (*str*) – Path to a file with password that will be formatted into **signing_command** string via `%(signing_key_password)s` string format syntax (if used). Because pungi config is usualy stored in git and is part of compose logs we don't want password to be included directly in the config. Note: If – string is used instead of a filename, then you will be asked for the password interactivelly right after pungi starts.

4.6.1 Example

```
signing_command = '~/git/releng/scripts/sigulsign_unsigned.py -vv --password=
↪%(signing_key_password)s fedora-24'
signing_key_id = '81b46521'
signing_key_password_file = '~/password_for_fedora-24_key'
```

4.7 Git URLs

In multiple places the config requires URL of a Git repository to download some file from. This URL is passed on to *Koji*. It is possible to specify which commit to use using this syntax:

```
git://git.example.com/git/repo-name.git?#<rev_spec>
```

The `<rev_spec>` pattern can be replaced with actual commit SHA, a tag name, `HEAD` to indicate that tip of default branch should be used or `origin/<branch_name>` to use tip of arbitrary branch.

If the URL specifies a branch or `HEAD`, *Pungi* will replace it with the actual commit SHA. This will later show up in *Koji* tasks and help with tracing what particular inputs were used.

Note: The `origin` must be specified because of the way *Koji* works with the repository. It will clone the repository then switch to requested state with `git reset --hard REF`. Since no local branches are created, we need to use full specification including the name of the remote.

4.8 Createrepo Settings

4.8.1 Options

createrepo_checksum (*str*) – specify checksum type for createrepo; expected values: `sha512`, `sha256`, `sha`. Defaults to `sha256`.

createrepo_c = True (*bool*) – use createrepo_c (True) or legacy createrepo (False)

createrepo_deltas = False (*list*) – generate delta RPMs against an older compose. This needs to be used together with `--old-composes` command line argument. The value should be a mapping of variants and architectures that should enable creating delta RPMs. Source and debuginfo repos never have deltas.

createrepo_use_xz = False (*bool*) – whether to pass `--xz` to the createrepo command. This will cause the SQLite databases to be compressed with xz.

createrepo_num_threads (*int*) – how many concurrent `createrepo` process to run. The default is to use one thread per CPU available on the machine.

createrepo_num_workers (*int*) – how many concurrent `createrepo` workers to run. Value defaults to 3.

createrepo_database (*bool*) – whether to create SQLite database as part of the repodata. This is only useful as an optimization for clients using Yum to consume to the repo. Default value depends on gather backend. For DNF it's turned off, for Yum the default is `True`.

createrepo_extra_args (*[str]*) – a list of extra arguments passed on to `createrepo` or `createrepo_c` executable. This could be useful for enabling zchunk generation and pointing it to correct dictionaries.

product_id = None (*scm_dict*) – If specified, it should point to a directory with certificates `<variant_uid>-<arch>-*.pem`. Pungi will copy each certificate file into the relevant Yum repositories as a `productid` file in the `repodata` directories. The purpose of these `productid` files is to expose the product data to subscription-manager. subscription-manager inclues a "product-id" Yum plugin that can read these `productid` certificate files from each Yum repository.

product_id_allow_missing = False (*bool*) – When `product_id` is used and a certificate for some variant and architecture is missing, Pungi will exit with an error by default. When you set this option to `True`, Pungi will ignore the missing certificate and simply log a warning message.

4.8.2 Example

```
createrepo_checksum = "sha"
createrepo_deltas = [
    # All arches for Everything should have deltas.
    ('^Everything$', {'*': True}),
    # Also Server.x86_64 should have them (but not on other arches).
    ('^Server$', {'x86_64': True}),
]
```

4.9 Package Set Settings

4.9.1 Options

sigkeys ([*str* or None]) – priority list of sigkeys; if the list includes an empty string or *None*, unsigned packages will be allowed

pkgset_source [mandatory] (*str*) – "koji" (any koji instance) or "repos" (arbitrary yum repositories)

pkgset_koji_tag (*str|[str]*) – tag(s) to read package set from. This option can be omitted for modular composes.

pkgset_koji_builds (*str|[str]*) – extra build(s) to include in a package set defined as NVRs.

pkgset_koji_module_tag (*str|[str]*) – tags to read module from. This option works similarly to listing tags in variants XML. If tags are specified and variants XML specifies some modules via NSVC (or part of), only modules matching that list will be used (and taken from the tag). Inheritance is used automatically.

pkgset_koji_inherit = True (*bool*) – inherit builds from parent tags; we can turn it off only if we have all builds tagged in a single tag

pkgset_koji_inherit_modules = False (*bool*) – the same as above, but this only applies to modular tags. This option applies to the content tags that contain the RPMs.

pkgset_repos (*dict*) – A mapping of architectures to repositories with RPMs: `{arch: [repo]}`. Only use when `pkgset_source = "repos"`.

pkgset_exclusive_arch_considers_noarch = True (*bool*) – If a package includes `noarch` in its `ExclusiveArch` tag, it will be included in all architectures since `noarch` is compatible with everything. Set this option to `False` to ignore `noarch` in `ExclusiveArch` and always consider only binary architectures.

4.9.2 Example

```
sigkeys = [None]
pkgset_source = "koji"
pkgset_koji_tag = "f23"
```

4.10 Buildinstall Settings

Script or process that creates bootable images with Anaconda installer is historically called buildinstall.

4.10.1 Options

buildinstall_method (*str*) – "lorax" (f16+, rhel7+) or "buildinstall" (older releases)

lorax_options (*list*) – special options passed on to *lorax*.

Format: `[(variant_uid_regex, {arch|*: {option: name}})]`.

Recognized options are:

- `bugurl` – *str* (default `None`)
- `nomacboot` – *bool* (default `True`)
- `noupgrade` – *bool* (default `True`)
- `add_template` – *[str]* (default empty)
- `add_arch_template` – *[str]* (default empty)
- `add_template_var` – *[str]* (default empty)
- `add_arch_template_var` – *[str]* (default empty)
- `rootfs_size` – [*int*] (default empty)

- `version` – [*str*] (default from `release_version`) – used as `--version` and `--release` argument on the lorax command line

lorax_extra_sources (*list*) – a variant/arch mapping with urls for extra source repositories added to Lorax command line. Either one repo or a list can be specified.

buildinstall_kickstart (*scm_dict*) – If specified, this kickstart file will be copied into each file and pointed to in boot configuration.

buildinstall_topdir (*str*) – Full path to top directory where the runroot buildinstall Koji tasks output should be stored. This is useful in situation when the Pungi compose is not generated on the same storage as the Koji task is running on. In this case, Pungi can provide input repository for runroot task using HTTP and set the output directory for this task to `buildinstall_topdir`. Once the runroot task finishes, Pungi will copy the results of runroot tasks to the compose working directory.

buildinstall_skip (*list*) – mapping that defines which variants and arches to skip during buildinstall; format: `[(variant_uid_regex, {arch|*: True})]`. This is only supported for lorax.

4.10.2 Example

```
buildinstall_method = "lorax"

# Enables macboot on x86_64 for all variants and builds upgrade images
# everywhere.
lorax_options = [
    ("^.*$", {
        "x86_64": {
            "nomacboot": False
        }
        "*": {
            "noupgrade": False
        }
    })
]

# Don't run buildinstall phase for Modular variant
buildinstall_skip = [
    ('^Modular', {
        '*': True
    })
]

# Add another repository for lorax to install packages from
lorax_extra_sources = [
    ('^Simple$', {
        '*': 'https://example.com/repo/$basearch/',
    })
]
```

Note: It is advised to run buildinstall (lorax) in koji, i.e. with **runroot enabled** for clean build environments, better logging, etc.

Warning: Lorax installs RPMs into a chroot. This involves running %post scriptlets and they frequently run executables in the chroot. If we're composing for multiple architectures, we **must** use runroot for this reason.

4.11 Gather Settings

4.11.1 Options

gather_method [mandatory] (*str*|*dict*) – Options are `deps`, `nodeps` and `hybrid`. Specifies whether and how package dependencies should be pulled in. Possible configuration can be one value for all variants, or if configured per-variant it can be a simple string `hybrid` or a a dictionary mapping source type to a value of `deps` or `nodeps`. Make sure only one regex matches each variant, as there is no guarantee which value will be used if there are multiple matching ones. All used sources must have a configured method unless hybrid solving is used.

gather_fulltree = False (*bool*) – When set to `True` all RPMs built from an SRPM will always be included. Only use when `gather_method = "deps"`.

gather_selfhosting = False (*bool*) – When set to `True`, *Pungi* will build a self-hosting tree by following build dependencies. Only use when `gather_method = "deps"`.

greedy_method (*str*) – This option controls how package requirements are satisfied in case a particular `Requires` has multiple candidates.

- `none` – the best packages is selected to satisfy the dependency and only that one is pulled into the compose
- `all` – packages that provide the symbol are pulled in
- `build` – the best package is selected, and then all packages from the same build that provide the symbol are pulled in

---

Note: As an example let's work with this situation: a package in the compose has `Requires: foo`. There are three packages with `Provides: foo`: `pkg-a`, `pkg-b-provider-1` and `pkg-b-provider-2`. The `pkg-b-*` packages are build from the same source package. Best match determines `pkg-b-provider-1` as best matching package.

- With `greedy_method = "none"` only `pkg-b-provider-1` will be pulled in.
- With `greedy_method = "all"` all three packages will be pulled in.
- With `greedy_method = "build"` ``pkg-b-provider-1`` and `pkg-b-provider-2` will be pulled in.

---

gather_backend (*str*) –This changes the entire codebase doing dependency solving, so it can change the result in unpredictable ways.

On Python 2, the choice is between `yum` or `dnf` and defaults to `yum`. On Python 3 `dnf` is the only option and default.

Particularly the multilib work is performed differently by using `python-multilib` library. Please refer to `multilib` option to see the differences.

multilib (*list*) – mapping of variant regexes and arches to list of multilib methods

Available methods are:

- `none` – no package matches this method
- `all` – all packages match this method
- `runtime` – packages that install some shared object file (`*.so.*`) will match.

- `devel` – packages whose name ends with `-devel` or `--static` suffix will be matched. When `dnf` is used, this method automatically enables `runtime` method as well. With `yum` backend this method also uses a hardcoded blacklist and whitelist.
- `kernel` – packages providing `kernel` or `kernel-devel` match this method (only in `yum` backend)
- `yaboot` – only `yaboot` package on `ppc` arch matches this (only in `yum` backend)

additional_packages (*list*) – additional packages to be included in a variant and architecture; format: `[(variant_uid_regex, {arch|*: [package_globs]})]`

The packages specified here are matched against RPM names, not any other provides in the package not the name of source package. Shell globbing is used, so wildcards are possible. The package can be specified as name only or `name.arch`.

filter_packages (*list*) – packages to be excluded from a variant and architecture; format: `[(variant_uid_regex, {arch|*: [package_globs]})]`

See *additional_packages* for details about package specification.

filter_system_release_packages (*bool*) – for each variant, figure out the best system release package and filter out all others. This will not work if a variant needs more than one system release package. In such case, set this option to `False`.

gather_prepopulate = None (*scm_dict*) – If specified, you can use this to add additional packages. The format of the file pointed to by this option is a JSON mapping `{variant_uid: {arch: {build: [package]}}}`. Packages added through this option can not be removed by `filter_packages`.

multilib_blacklist (*dict*) – multilib blacklist; format: `{arch|*: [package_globs]}`.

See *additional_packages* for details about package specification.

multilib_whitelist (*dict*) – multilib blacklist; format: `{arch|*: [package_names]}`. The whitelist must contain exact package names; there are no wildcards or pattern matching.

gather_lookaside_repos = [] (*list*) – lookaside repositories used for package gathering; format: `[(variant_uid_regex, {arch|*: [repo_urls]})]`

hashed_directories = False (*bool*) – put packages into "hashed" directories, for example `Packages/k/kernel-4.0.4-301.fc22.x86_64.rpm`

check_deps = True (*bool*) – Set to `False` if you don't want the compose to abort when some package has broken dependencies.

require_all_comps_packages = False (*bool*) – Set to `True` to abort compose when package mentioned in comps file can not be found in the package set. When disabled (the default), such cases are still reported as warnings in the log.

gather_source_mapping (*str*) – JSON mapping with initial packages for the compose. The value should be a path to JSON file with following mapping: `{variant: {arch: {rpm_name: [rpm_arch|None]}}}`.

gather_profiler = False (*bool*) – When set to `True` the gather tool will produce additional performance profiling information at the end of its logs. Only takes effect when `gather_backend = "dnf"`.

variant_as_lookaside (*list*) – a variant/variant mapping that tells one or more variants in compose has other variant(s) in compose as a lookaside. Only top level variants are supported (not addons/layered products). Format: `[(variant_uid, variant_uid)]`

4.11.2 Example

```
gather_method = "deps"
greedy_method = "build"
check_deps = False
hashed_directories = True

gather_method = {
    "^Everything$": {
        "comps": "deps"      # traditional content defined by comps groups
    },
    "^Modular$": {
        "module": "nodeps"   # Modules do not need dependencies
    },
    "^Mixed$": {             # Mixed content in one variant
        "comps": "deps",
        "module": "nodeps"
    }
    "^OtherMixed$": "hybrid",    # Using hybrid depsolver
}

additional_packages = [
    # bz#123456
    ('^(Workstation|Server)$', {
        '*': [
            'grub2',
            'kernel',
        ],
    }),
]

filter_packages = [
    # bz#111222
    ('^.*$', {
        '*': [
            'kernel-doc',
        ],
    }),
]

multilib = [
    ('^Server$', {
        'x86_64': ['devel', 'runtime']
    })
]

multilib_blacklist = {
    "*": [
        "gcc",
    ],
}

multilib_whitelist = {
    "*": [
        "alsa-plugins-*",
    ],
}
```

(continues on next page)

(continued from previous page)

```
# gather_lookaside_repos = [
#     ('^.*$', {
#         'x86_64': [
#             "https://dl.fedoraproject.org/pub/fedora/linux/releases/22/Everything/
↪x86_64/os/",
#             "https://dl.fedoraproject.org/pub/fedora/linux/releases/22/Everything/
↪source/SRPMS/",
#         ]
#     }),
# ]
```

Note: It is a good practice to attach bug/ticket numbers to additional_packages, filter_packages, multilib_blacklist and multilib_whitelist to track decisions.

4.12 Koji Settings

4.12.1 Options

koji_profile (*str*) – koji profile name. This tells Pungi how to communicate with your chosen Koji instance. See Koji's documentation about profiles for more information about how to set up your Koji client profile. In the examples, the profile name is "koji", which points to Fedora's koji.fedoraproject.org.

runroot_method (*str*) – Runroot method to use. It can further specify the runroot method in case the `runroot` is set to True.

Available methods are:

- `local` – runroot tasks are run locally
- `koji` – runroot tasks are run in Koji
- `openssh` – runroot tasks are run on remote machine connected using OpenSSH. The `runroot_ssh_hostnames` for each architecture must be set and the user under which Pungi runs must be configured to login as `runroot_ssh_username` using the SSH key.

runroot_channel (*str*) – name of koji channel

runroot_tag (*str*) – name of koji **build** tag used for runroot

runroot_weights (*dict*) – customize task weights for various runroot tasks. The values in the mapping should be integers, the keys can be selected from the following list. By default no weight is assigned and Koji picks the default one according to policy.

- `buildinstall`
- `createiso`
- `ostree`
- `ostree_installer`

4.12.2 Example

```
koji_profile = "koji"
runroot_channel = "runroot"
runroot_tag = "f23-build"
```

4.13 Runroot "openssh" method settings

4.13.1 Options

runroot_ssh_username (*str*) – For `openssh` runroot method, configures the username used to login the remote machine to run the runroot task. Defaults to "root".

runroot_ssh_hostnames (*dict*) – For `openssh` runroot method, defines the hostname for each architecture on which the runroot task should be running. Format: `{"x86_64": "runroot-x86-64.localhost.tld", ...}`

runroot_ssh_init_template (*str*) [optional] – For `openssh` runroot method, defines the command to initializes the runroot task on the remote machine. This command is executed as first command for each runroot task executed.

The command can print a string which is then available as `{runroot_key}` for other SSH commands. This string might be used to keep the context across different SSH commands executed for single runroot task.

The goal of this command is setting up the environment for real runroot commands. For example preparing the unique mock environment, mounting the desired file-systems, ...

The command string can contain following variables which are replaced by the real values before executing the init command:

- `{runroot_tag}` - Tag to initialize the runroot environment from.

When not set, no init command is executed.

runroot_ssh_install_packages_template (*str*) [optional] – For `openssh` runroot method, defines the template for command to install the packages requested to run the runroot task.

The template string can contain following variables which are replaced by the real values before executing the install command:

- `{runroot_key}` - Replaced with the string returned by `runroot_ssh_init_template` if used. This can be used to keep the track of context of SSH commands beloging to single runroot task.
- `{packages}` - White-list separated list of packages to install.

Example (The `{runroot_key}` is expected to be set to mock config file using the `runroot_ssh_init_template` command.): `"mock -r {runroot_key} --install {packages}"`

When not set, no command to install packages on remote machine is executed.

runroot_ssh_run_template (*str*) [optional] – For `openssh` runroot method, defines the template for the main runroot command.

The template string can contain following variables which are replaced by the real values before executing the install command:

- `{runroot_key}` - Replaced with the string returned by `runroot_ssh_init_template` if used. This can be used to keep the track of context of SSH commands beloging to single runroot task.
- `{command}` - Command to run.

Example (The `{runroot_key}` is expected to be set to mock config file using the `runroot_ssh_init_template` command.): `"mock -r {runroot_key} chroot -- {command}"`

When not set, the runroot command is run directly.

4.14 Extra Files Settings

4.14.1 Options

extra_files (*list*) – references to external files to be placed in os/ directory and media; format: `[(variant_uid_regex, {arch|*: [scm_dict]})]`. See *Exporting files from SCM* for details. If the dict specifies a `target` key, an additional subdirectory will be used.

4.14.2 Example

```
extra_files = [
    ('^.*$', {
        '*': [
            # GPG keys
            {
                "scm": "rpm",
                "repo": "fedora-repos",
                "branch": None,
                "file": [
                    "/etc/pki/rpm-gpg/RPM-GPG-KEY-22-fedora",
                ],
                "target": "",
            },
            # GPL
            {
                "scm": "git",
                "repo": "https://pagure.io/pungi-fedora",
                "branch": None,
                "file": [
                    "GPL",
                ],
                "target": "",
            },
        ],
    }),
]
```

4.14.3 Extra Files Metadata

If extra files are specified a metadata file, `extra_files.json`, is placed in the `os/` directory and media. The checksums generated are determined by `media_checksums` option. This metadata file is in the format:

```
{
  "header": {"version": "1.0},
  "data": [
    {
```

(continues on next page)

(continued from previous page)

```
      "file": "GPL",
      "checksums": {
        "sha256": "8177f97513213526df2cf6184d8ff986c675afb514d4e68a404010521b880643"
      },
      "size": 18092
    },
    {
      "file": "release-notes/notes.html",
      "checksums": {
        "sha256": "82b1ba8db522aadf101dca6404235fba179e559b95ea24ff39ee1e5d9a53bdcb"
      },
      "size": 1120
    }
  ]
}
```

4.15 Productimg Settings

Product images are placed on installation media and provide additional branding and Anaconda changes specific to product variants.

4.15.1 Options

productimg = False (*bool*) – create product images; requires buildinstall_method option

productimg_install_class (*scm_dict*, *str*) – reference to install class **file**

productimg_po_files (*scm_dict*, *str*) – reference to a **directory** with po files for install class translations

4.15.2 Example

```
productimg = True
productimg_install_class = {
    "scm": "git",
    "repo": "http://git.example.com/productimg.git",
    "branch": None,
    "file": "fedora23/%(variant_id)s.py",
}
productimg_po_files = {
    "scm": "git",
    "repo": "http://git.example.com/productimg.git",
    "branch": None,
    "dir": "po",
}
```

4.16 CreateISO Settings

4.16.1 Options

createiso_skip = False (*list*) – mapping that defines which variants and arches to skip during createiso; format: [(variant_uid_regex, {arch|*: True})]

createiso_max_size (*list*) – mapping that defines maximum expected size for each variant and arch. If the ISO is larger than the limit, a warning will be issued.

Format: `[(variant_uid_regex, {arch|*: number})]`

createiso_max_size_is_strict (*list*) – Set the value to `True` to turn the warning from `createiso_max_size` into a hard error that will abort the compose. If there are multiple matches in the mapping, the check will be strict if at least one match says so.

Format: `[(variant_uid_regex, {arch|*: bool})]`

create_jigdo = True (*bool*) – controls the creation of jigdo from ISO

create_optional_isos = False (*bool*) – when set to `True`, ISOs will be created even for `optional` variants. By default only variants with type `variant` or `layered-product` will get ISOs.

createiso_break_hardlinks = False (*bool*) – when set to `True`, all files that should go on the ISO and have a hardlink will be first copied into a staging directory. This should work around a bug in `genisoimage` including incorrect link count in the image, but it is at the cost of having to copy a potentially significant amount of data.

The staging directory is deleted when ISO is successfully created. In that case the same task to create the ISO will not be re-runnable.

iso_size = 4700000000 (*int|str*) – size of ISO image. The value should either be an integer meaning size in bytes, or it can be a string with `k`, `M`, `G` suffix (using multiples of 1024).

split_iso_reserve = 10MiB (*int|str*) – how much free space should be left on each disk. The format is the same as for `iso_size` option.

iso_hfs_ppc64le_compatible = True (*bool*) – when set to False, the Apple/HFS compatibility is turned off for ppc64le ISOs. This option only makes sense for bootable products, and affects images produced in *createiso* and *extra_isos* phases.

Note: Source architecture needs to be listed explicitly. Excluding '*' applies only on binary arches. Jigdo causes significant increase of time to ISO creation.

4.16.2 Example

```
createiso_skip = [
    ('^Workstation$', {
        '*': True,
        'src': True
    }),
]
```

4.17 Automatic generation of version and release

Version and release values for certain artifacts can be generated automatically based on release version, compose label, date, type and respin. This can be used to shorten the config and keep it the same for multiple uses.

Compose ID	Label	Version	Date	Respin	Release
F-Rawhide-20170406.n.0	-	Rawhide	20170406	0	20170406.n.0
F-26-20170329.1	Alpha-1.6	26_Alpha	20170329	1	1.6
F-Atomic-25-20170407.0	RC-20170407.0	25	20170407	0	20170407.0
F-Atomic-25-20170407.0	-	25	20170407	0	20170407.0

All non-RC milestones from label get appended to the version. For release either label is used or date, type and respin.

4.18 Common options for Live Images, Live Media and Image Build

All images can have `ksurl`, `version`, `release` and `target` specified. Since this can create a lot of duplication, there are global options that can be used instead.

For each of the phases, if the option is not specified for a particular deliverable, an option named `<PHASE_NAME>_<OPTION>` is checked. If that is not specified either, the last fallback is `global_<OPTION>`. If even that is unset, the value is considered to not be specified.

The kickstart URL is configured by these options.

- `global_ksurl` – global fallback setting
- `live_media_ksurl`
- `image_build_ksurl`
- `live_images_ksurl`

Target is specified by these settings.

- `global_target` – global fallback setting
- `live_media_target`
- `image_build_target`
- `live_images_target`

Version is specified by these options. If no version is set, a default value will be provided according to *automatic versioning*.

- `global_version` – global fallback setting
- `live_media_version`
- `image_build_version`
- `live_images_version`

Release is specified by these options. If set to a magic value to `!RELEASE_FROM_LABEL_DATE_TYPE_RESPIN`, a value will be generated according to *automatic versioning*.

- `global_release` – global fallback setting

- `live_media_release`
- `image_build_release`
- `live_images_release`

Each configuration block can also optionally specify a `failable` key. For live images it should have a boolean value. For live media and image build it should be a list of strings containing architectures that are optional. If any deliverable fails on an optional architecture, it will not abort the whole compose. If the list contains only `"*"`, all arches will be substituted.

4.19 Live Images Settings

live_images (*list*) – Configuration for the particular image. The elements of the list should be tuples `(variant_uid_regex, {arch|*: config})`. The config should be a dict with these keys:

- `kickstart` (*str*)
- `ksurl` (*str*) [optional] – where to get the kickstart from
- `name` (*str*)
- `version` (*str*)
- `target` (*str*)
- `repo` (*str|[str]*) – repos specified by URL or variant UID
- `specfile` (*str*) – for images wrapped in RPM
- `scratch` (*bool*) – only RPM-wrapped images can use scratch builds, but by default this is turned off
- `type` (*str*) – what kind of task to start in Koji. Defaults to `live` meaning `koji spin-livecd` will be used. Alternative option is `appliance` corresponding to `koji spin-appliance`.
- `sign` (*bool*) – only RPM-wrapped images can be signed

live_images_no_rename (*bool*) – When set to `True`, filenames generated by Koji will be used. When `False`, filenames will be generated based on `image_name_format` configuration option.

4.20 Live Media Settings

live_media (*dict*) – configuration for `koji spin-livemedia`; format: `{variant_uid_regex: [{opt:value}]}`

Required options:

- `name` (*str*)
- `version` (*str*)
- `arches` (*[str]*) – what architectures to build the media for; by default uses all arches for the variant.
- `kickstart` (*str*) – name of the kickstart file

Available options:

- `ksurl` (*str*)
- `ksversion` (*str*)
- `scratch` (*bool*)

- `target` (*str*)
- `release` (*str*) – a string with the release, or `!RELEASE_FROM_LABEL_DATE_TYPE_RESPIN` to automatically generate a suitable value. See *automatic versioning* for details.
- `skip_tag` (*bool*)
- `repo` (*str|[str]*) – repos specified by URL or variant UID
- `title` (*str*)
- `install_tree_from` (*str*) – variant to take install tree from

4.21 Image Build Settings

image_build (*dict*) – config for `koji image-build`; format: {variant_uid_regex: [{opt: value}]}

By default, images will be built for each binary arch valid for the variant. The config can specify a list of arches to narrow this down.

Note: Config can contain anything what is accepted by `koji image-build --config configfile.ini`

Repo can be specified either as a string or a list of strings. It will be automatically transformed into format suitable for `koji`. A repo for the currently built variant will be added as well.

If you explicitly set `release` to `!RELEASE_FROM_LABEL_DATE_TYPE_RESPIN`, it will be replaced with a value generated as described in *automatic versioning*.

If you explicitly set `release` to `!RELEASE_FROM_DATE_RESPIN`, it will be replaced with a value generated as described in *automatic versioning*.

If you explicitly set `version` to `!VERSION_FROM_VERSION`, it will be replaced with a value generated as described in *automatic versioning*.

Please don't set `install_tree`. This gets automatically set by *pungi* based on current variant. You can use `install_tree_from` key to use install tree from another variant.

Both the install tree and repos can use one of following formats:

- URL to the location
- name of variant in the current compose
- absolute path on local filesystem (which will be translated using configured mappings or used unchanged, in which case you have to ensure the koji builders can access it)

You can set either a single format, or a list of formats. For available values see help output for `koji image-build` command.

If `ksurl` ends with `#HEAD`, Pungi will figure out the SHA1 hash of current HEAD and use that instead.

Setting `scratch` to `True` will run the koji tasks as scratch builds.

4.21.1 Example

```
image_build = {
    '^Server$': [
        {
            'image-build': {
                'format': ['docker', 'qcow2']
                'name': 'fedora-qcow-and-docker-base',
                'target': 'koji-target-name',
                'ksversion': 'F23',     # value from pykickstart
                'version': '23',
                # correct SHA1 hash will be put into the URL below automatically
                'ksurl': 'https://git.fedorahosted.org/git/spin-kickstarts.git?
↪somedirectoryifany#HEAD',
                'kickstart': "fedora-docker-base.ks",
                'repo': ["http://someextrarepos.org/repo", "ftp://rekcod.oi/repo"],
                'distro': 'Fedora-20',
                'disk_size': 3,

                # this is set automatically by pungi to os_dir for given variant
                # 'install_tree': 'http://somepath',
            },
            'factory-parameters': {
                'docker_cmd':  "[ '/bin/bash' ]",
                'docker_env': "[ 'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
↪bin:/sbin:/bin' ]",
                'docker_labels': "{'Name': 'fedora-docker-base', 'License': u'GPLv2',
↪'RUN': 'docker run -it --rm ${OPT1} --privileged -v \`pwd\`:/atomicapp -v /run:/run␣
↪-v /:/host --net=host --name ${NAME} -e NAME=${NAME} -e IMAGE=${IMAGE} ${IMAGE} -v $
↪{OPT2} run ${OPT3} /atomicapp', 'Vendor': 'Fedora Project', 'Version': '23',
↪'Architecture': 'x86_64' }",
            }
        },
        {
            'image-build': {
                'format': ['docker', 'qcow2']
                'name': 'fedora-qcow-and-docker-base',
                'target': 'koji-target-name',
                'ksversion': 'F23',     # value from pykickstart
                'version': '23',
                # correct SHA1 hash will be put into the URL below automatically
                'ksurl': 'https://git.fedorahosted.org/git/spin-kickstarts.git?
↪somedirectoryifany#HEAD',
                'kickstart': "fedora-docker-base.ks",
                'repo': ["http://someextrarepos.org/repo", "ftp://rekcod.oi/repo"],
                'distro': 'Fedora-20',
                'disk_size': 3,

                # this is set automatically by pungi to os_dir for given variant
                # 'install_tree': 'http://somepath',
            }
        },
        {
            'image-build': {
                'format': 'qcow2',
                'name': 'fedora-qcow-base',
                'target': 'koji-target-name',
                'ksversion': 'F23',     # value from pykickstart
                'version': '23',
```

(continues on next page)

(continued from previous page)

```
                'ksurl': 'https://git.fedorahosted.org/git/spin-kickstarts.git?
↪somedirectoryifany#HEAD',
                'kickstart': "fedora-docker-base.ks",
                'distro': 'Fedora-23',

                # only build this type of image on x86_64
                'arches': ['x86_64']

                # Use install tree and repo from Everything variant.
                'install_tree_from': 'Everything',
                'repo': ['Everything'],

                # Set release automatically.
                'release': '!RELEASE_FROM_LABEL_DATE_TYPE_RESPIN',
            }
        }
    ]
}
```

4.22 OSTree Settings

The `ostree` phase of *Pungi* can create and update ostree repositories. This is done by running `rpm-ostree compose` in a Koji runroot environment. The ostree repository itself is not part of the compose and should be located in another directory. Any new packages in the compose will be added to the repository with a new commit.

ostree (*dict*) – a mapping of configuration for each. The format should be `{variant_uid_regex: config_dict}`. It is possible to use a list of configuration dicts as well.

The configuration dict for each variant arch pair must have these keys:

- `treefile` – (*str*) Filename of configuration for `rpm-ostree`.
- `config_url` – (*str*) URL for Git repository with the `treefile`.
- `repo` – (*str|dict|[str|dict]*) repos specified by URL or variant UID or a dict of repo options, `baseurl` is required in the dict.
- `ostree_repo` – (*str*) Where to put the ostree repository

These keys are optional:

- `keep_original_sources` – (*bool*) Keep the existing source repos in the tree config file. If not enabled, all the original source repos will be removed from the tree config file.
- `config_branch` – (*str*) Git branch of the repo to use. Defaults to `master`.
- `arches` – (*[str]*) List of architectures for which to update ostree. There will be one task per architecture. By default all architectures in the variant are used.
- `failable` – (*[str]*) List of architectures for which this deliverable is not release blocking.
- `update_summary` – (*bool*) Update summary metadata after tree composing. Defaults to `False`.
- `force_new_commit` – (*bool*) Do not use rpm-ostree's built-in change detection. Defaults to `False`.
- `version` – (*str*) Version string to be added as versioning metadata. If this option is set to `!OSTREE_VERSION_FROM_LABEL_DATE_TYPE_RESPIN`, a value will be generated automatically as `$VERSION.$RELEASE`. If this option is set to `!VERSION_FROM_VERSION_DATE_RESPIN`, a value will be generated automatically as `$VERSION.$DATE.$RESPIN`. *See how those values are created*.

- `tag_ref` – (*bool*, default `True`) If set to `False`, a git reference will not be created.
- `ostree_ref` – (*str*) To override value `ref` from `treefile`.

4.22.1 Example config

```
ostree = {
    "^Atomic$": {
        "treefile": "fedora-atomic-docker-host.json",
        "config_url": "https://git.fedorahosted.org/git/fedora-atomic.git",
        "repo": [
            "Server",
            "http://example.com/repo/x86_64/os",
            {"baseurl": "Everything"},
            {"baseurl": "http://example.com/linux/repo", "exclude": "systemd-container
↪"},
        ],
        "keep_original_sources": True,
        "ostree_repo": "/mnt/koji/compose/atomic/Rawhide/",
        "update_summary": True,
        # Automatically generate a reasonable version
        "version": "!OSTREE_VERSION_FROM_LABEL_DATE_TYPE_RESPIN",
        # Only run this for x86_64 even if Atomic has more arches
        "arches": ["x86_64"],
    }
}
```

4.23 Ostree Installer Settings

The `ostree_installer` phase of *Pungi* can produce installer image bundling an OSTree repository. This always runs in Koji as a `runroot` task.

ostree_installer (*dict*) – a variant/arch mapping of configuration. The format should be `[(variant_uid_regex, {arch|*: config_dict})]`.

The configuration dict for each variant arch pair must have this key:

These keys are optional:

- `repo` – (*str|[str]*) repos specified by URL or variant UID
- `release` – (*str*) Release value to set for the installer image. Set to `!RELEASE_FROM_LABEL_DATE_TYPE_RESPIN` to generate the value *automatically*.
- `failable` – (*[str]*) List of architectures for which this deliverable is not release blocking.

These optional keys are passed to `lorax` to customize the build.

- `installpkgs` – (*[str]*)
- `add_template` – (*[str]*)
- `add_arch_template` – (*[str]*)
- `add_template_var` – (*[str]*)
- `add_arch_template_var` – (*[str]*)
- `rootfs_size` – (*[str]*)

- `template_repo` – (*str*) Git repository with extra templates.
- `template_branch` – (*str*) Branch to use from `template_repo`.

The templates can either be absolute paths, in which case they will be used as configured; or they can be relative paths, in which case `template_repo` needs to point to a Git repository from which to take the templates.

ostree_installer_overwrite = False (*bool*) – by default if a variant including OSTree installer also creates regular installer images in buildinstall phase, there will be conflicts (as the files are put in the same place) and Pungi will report an error and fail the compose.

With this option it is possible to opt-in for the overwriting. The traditional `boot.iso` will be in the `iso/` subdirectory.

4.23.1 Example config

```
ostree_installer = [
    ("^Atomic$", {
        "x86_64": {
            "repo": [
                "Everything",
                "https://example.com/extra-repo1.repo",
                "https://example.com/extra-repo2.repo",
            ],
            "release": "!RELEASE_FROM_LABEL_DATE_TYPE_RESPIN",
            "installpkgs": ["fedora-productimg-atomic"],
            "add_template": ["atomic-installer/lorax-configure-repo.tmpl"],
            "add_template_var": [
                "ostree_osname=fedora-atomic",
                "ostree_ref=fedora-atomic/Rawhide/x86_64/docker-host",
            ],
            "add_arch_template": ["atomic-installer/lorax-embed-repo.tmpl"],
            "add_arch_template_var": [
                "ostree_repo=https://kojipkgs.fedoraproject.org/compose/atomic/
↪Rawhide/",
                "ostree_osname=fedora-atomic",
                "ostree_ref=fedora-atomic/Rawhide/x86_64/docker-host",
            ]
            'template_repo': 'https://git.fedorahosted.org/git/spin-kickstarts.git',
            'template_branch': 'f24',
        }
    })
]
```

4.24 OSBS Settings

Pungi can build container images in OSBS. The build is initiated through Koji `container-build` plugin. The base image will be using RPMs from the current compose and a `Dockerfile` from specified Git repository.

Please note that the image is uploaded to a registry and not exported into compose directory. There will be a metadata file in `compose/metadata/osbs.json` with details about the built images (assuming they are not scratch builds).

osbs (*dict*) – a mapping from variant regexes to configuration blocks. The format should be `{variant_uid_regex: [config_dict]}`.

The configuration for each image must have at least these keys:

- `url` – (*str*) URL pointing to a Git repository with `Dockerfile`. Please see *Git URLs* section for more details.
- `target` – (*str*) A Koji target to build the image for.
- `git_branch` – (*str*) A branch in SCM for the `Dockerfile`. This is required by OSBS to avoid race conditions when multiple builds from the same repo are submitted at the same time. Please note that `url` should contain the branch or tag name as well, so that it can be resolved to a particular commit hash.

Optionally you can specify `failable`. If it has a truthy value, failure to create the image will not abort the whole compose.

Note: Once OSBS gains support for multiple architectures, the usage of this option will most likely change to list architectures that are allowed to fail.

The configuration will pass other attributes directly to the Koji task. This includes `name`, `version`, `scratch` and `priority`.

A value for `yum_repourls` will be created automatically and point at a repository in the current compose. You can add extra repositories with `repo` key having a list of urls pointing to `.repo` files or just variant uid, Pungi will create the .repo file for that variant. `gpgkey` can be specified to enable gpgcheck in repo files for variants.

osbs_registries (*dict*) – It is possible to configure extra information about where to push the image (unless it is a scratch build). For each finished build, Pungi will try to match NVR against a key in this mapping (using shell-style globbing) and take the corresponding value and collect them across all built images. The data will be saved into `logs/global/osbs-registries.json` as a mapping from Koji NVR to the registry data. The same data is also sent to the message bus on `osbs-request-push` topic once the compose finishes successfully. Handling the message and performing the actual push is outside of scope for Pungi.

4.24.1 Example config

```
osbs = {
    "^Server$": {
        # required
        "url": "git://example.com/dockerfiles.git?#HEAD",
        "target": "f24-docker-candidate",
        "git_branch": "f24-docker",

        # optional
        "name": "fedora-docker-base",
        "version": "24",
        "repo": ["Everything", "https://example.com/extra-repo.repo"],
        # This will result in three repo urls being passed to the task.
        # They will be in this order: Server, Everything, example.com/
        "gpgkey": 'file:///etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release',
    }
}
```

4.25 Extra ISOs

Create an ISO image that contains packages from multiple variants. Such ISO always belongs to one variant, and will be stored in ISO directory of that variant.

The ISO will be bootable if buildinstall phase runs for the parent variant. It will reuse boot configuration from that variant.

extra_isos (*dict*) – a mapping from variant UID regex to a list of configuration blocks.

- `include_variants` – (*list*) list of variant UIDs from which content should be added to the ISO; the variant of this image is added automatically.

Rest of configuration keys is optional.

- `filename` – (*str*) template for naming the image. In addition to the regular placeholders `filename` is available with the name generated using `image_name_format` option.
- `volid` – (*str*) template for generating volume ID. Again `volid` placeholder can be used similarly as for file name. This can also be a list of templates that will be tried sequentially until one generates a volume ID that fits into 32 character limit.
- `extra_files` – (*list*) a list of *scm_dict* objects. These files will be put in the top level directory of the image.
- `arches` – (*list*) a list of architectures for which to build this image. By default all arches from the variant will be used. This option can be used to limit them.
- `failable_arches` – (*list*) a list of architectures for which the image can fail to be generated and not fail the entire compose.
- `skip_src` – (*bool*) allows to disable creating an image with source packages.
- `inherit_extra_files` – (*bool*) by default extra files in variants are ignored. If you want to include them in the ISO, set this option to `True`.
- `max_size` – (*int*) expected maximum size in bytes. If the final image is larger, a warning will be issued.

4.25.1 Example config

```
extra_isos = {
    'Server': [{
        # Will generate foo-DP-1.0-20180510.t.43-Server-x86_64-dvd1.iso
        'filename': 'foo-{filename}',
        'volid': 'foo-{arch}',

        'extra_files': [{
            'scm': 'git',
            'repo': 'https://pagure.io/pungi.git',
            'file': 'setup.py'
        }],

        'include_variants': ['Client']
    }]
}
# This should create image with the following layout:
#  .
#  ├── Client
#  │   ├── Packages
#  │   │   ├── a
#  │   │   └── b
#  │   └── repodata
#  ├── Server
#  │   ├── Packages
```

(continues on next page)

(continued from previous page)

```
#   │   │   ├── a
#   │   │   └── b
#   │   └── repodata
#   └── setup.py
```

4.26 Media Checksums Settings

media_checksums (*list*) – list of checksum types to compute, allowed values are anything supported by Python's `hashlib` module (see documentation for details).

media_checksum_one_file (*bool*) – when `True`, only one `CHECKSUM` file will be created per directory; this option requires `media_checksums` to only specify one type

media_checksum_base_filename (*str*) – when not set, all checksums will be save to a file named either `CHECKSUM` or based on the digest type; this option allows adding any prefix to that name

It is possible to use format strings that will be replace by actual values. The allowed keys are:

- `arch`
- `compose_id`
- `date`
- `label`
- `label_major_version`
- `release_short`
- `respin`
- `type`
- `type_suffix`
- `version`
- `dirname` (only if `media_checksum_one_file` is enabled)

For example, for Fedora the prefix should be `%(release_short)s-%(variant)s-%(version)s-%(date)s%(type_%(respin)s`.

4.27 Translate Paths Settings

translate_paths (*list*) – list of paths to translate; format: `[(path, translated_path)]`

Note: This feature becomes useful when you need to transform compose location into e.g. a HTTP repo which is can be passed to `koji image-build`. The `path` part is normalized via `os.path.normpath()`.

4.27.1 Example config

```
translate_paths = [
    ("/mnt/a", "http://b/dir"),
]
```

4.27.2 Example usage

```
>>> from pungi.util import translate_paths
>>> print translate_paths(compose_object_with_mapping, "/mnt/a/c/somefile")
http://b/dir/c/somefile
```

4.28 Miscellaneous Settings

paths_module (*str*) – Name of Python module implementing the same interface as `pungi.paths`. This module can be used to override where things are placed.

link_type = `hardlink-or-copy` (*str*) – Method of putting packages into compose directory.

Available options:

- `hardlink-or-copy`
- `hardlink`
- `copy`
- `symlink`
- `abspath-symlink`

skip_phases (*list*) – List of phase names that should be skipped. The same functionality is available via a command line option.

release_discinfo_description (*str*) – Override description in `.discinfo` files. The value is a format string accepting `%(variant_name)s` and `%(arch)s` placeholders.

symlink_isos_to (*str*) – If set, the ISO files from `buildinstall`, `createiso` and `live_images` phases will be put into this destination, and a symlink pointing to this location will be created in actual compose directory.

dogpile_cache_backend (*str*) – If set, Pungi will use the configured Dogpile cache backend to cache various data between multiple Pungi calls. This can make Pungi faster in case more similar composes are running regularly in short time.

For list of available backends, please see the https://dogpilecache.readthedocs.io documentation.

Most typical configuration uses the `dogpile.cache.dbm` backend.

dogpile_cache_arguments (*dict*) – Arguments to be used when creating the Dogpile cache backend. See the particular backend's configuration for the list of possible key/value pairs.

For the `dogpile.cache.dbm` backend, the value can be for example following:

```
{
    "filename": "/tmp/pungi_cache_file.dbm"
}
```

dogpile_cache_expiration_time (*int*) – Defines the default expiration time in seconds of data stored in the Dogpile cache. Defaults to 3600 seconds.

CHAPTER

FIVE

EXPORTING FILES FROM SCM

Multiple places in Pungi can use files from external storage. The configuration is similar independently of the backend that is used, although some features may be different.

The so-called `scm_dict` is always put into configuration as a dictionary, which can contain following keys.

- `scm` – indicates which SCM system is used. This is always required. Allowed values are:
  - `file` – copies files from local filesystem
  - `git` – copies files from a Git repository
  - `cvs` – copies files from a CVS repository
  - `rpm` – copies files from a package in the compose
- `repo` – for Git and CVS backends URL to the repository, for RPM a shell glob for matching package names (or a list of such globs); for `file` backend this option should be empty (or left out)
- `branch` – branch name for Git and CVS backends, with `master` and `HEAD` as defaults. Ignored for other backends.
- `file` – a list of files that should be exported.
- `dir` – a directory that should be exported. All its contents will be exported. This option is mutually exclusive with `file`.
- `command` – defines a shell command to run after Git clone to generate the needed file (for example to run `make`). Only supported in Git backend.

5.1 `file` vs. `dir`

Exactly one of these two options has to be specified. Documentation for each configuration option should specify whether it expects a file or a directory.

For `extra_files` phase either key is valid and should be chosen depending on what the actual use case.

5.2 Caveats

The `rpm` backend can only be used in phases that would extract the files after `pkgset` phase finished. You can't get comps file from a package.

Depending on Git repository URL configuration Pungi can only export the requested content using `git archive`. When a command should run this is not possible and a clone is always needed.

CHAPTER

SIX

PROGRESS NOTIFICATION

Pungi has the ability to emit notification messages about progress and general status of the compose. These can be used to e.g. send messages to *fedmsg*. This is implemented by actually calling a separate script.

The script will be called with one argument describing action that just happened. A JSON-encoded object will be passed to standard input to provide more information about the event. At the very least, the object will contain a `compose_id` key.

The script is invoked in compose directory and can read other information there.

Currently these messages are sent:

- `status-change` – when composing starts, finishes or fails; a `status` key is provided to indicate details
- `phase-start` – on start of a phase
- `phase-stop` – when phase is finished
- `createiso-targets` – with a list of images to be created
- `createiso-imagedone` – when any single image is finished
- `createiso-imagefail` – when any single image fails to create
- `fail-to-start` – when there are incorrect CLI options or errors in configuration file; this message does not contain `compose_id` nor is it started in the compose directory (which does not exist yet)
- `ostree` – when a new commit is created, this message will announce its hash and the name of ref it is meant for.

For phase related messages `phase_name` key is provided as well.

A `pungi-fedmsg-notification` script is provided and understands this interface.

6.1 Setting it up

The script should be provided as a command line argument `--notification-script`.

```
--notification-script=pungi-fedmsg-notification
```

CHAPTER

SEVEN

GATHERING PACKAGES

A compose created by Pungi consists of one or more variants. A variant contains a subset of the content targeted at a particular use case.

There are different types of variants. The type affects how packages are gathered into the variant.

The inputs for gathering are defined by various gather sources. Packages from all sources are collected to create a big list of package names, comps groups names and a list of packages that should be filtered out.

Note: The inputs for both explicit package list and comps file are interpreted as RPM names, not any arbitrary provides nor source package name.

Next, `gather_method` defines how the list is processed. For `nodeps`, the results from source are used pretty much as is[1]. For `deps` method, a process will be launched to figure out what dependencies are needed and those will be pulled in.

7.1 Variant types

Variant is a base type that has no special behaviour.

Addon is built on top of a regular variant. Any packages that should go to both the addon and its parent will be removed from addon. Packages that are only in addon but pulled in because of `gather_fulltree` option will be moved to parent.

Integrated Layered Product works similarly to *addon*. Additionally, all packages from addons on the same parent variant are removed integrated layered products.

The main difference between an *addon* and *integrated layered product* is that *integrated layered product* has its own identity in the metadata (defined with product name and version).

Note: There's also *Layered Product* as a term, but this is not related to variants. It's used to describe a product that is not a standalone operating system and is instead meant to be used on some other base system.

Optional contains packages that complete the base variants' package set. It always has `fulltree` and `selfhosting` enabled, so it contains build dependencies and packages which were not specifically requested for base variant.

Some configuration options are overridden for particular variant types.

[1] The lists are filtered based on what packages are available in the package set, but nothing else will be pulled in.

Table 1: Depsolving configuration

Variant	Fulltree	Selfhosting
base	configurable	configurable
addon/ILP	enabled	disabled
optional	enabled	enabled

7.2 Profiling

Profiling data on the `pungi-gather` tool can be enabled by setting the `gather_profiler` configuration option to `True`.

7.3 Modular compose

A compose with `gather_source` set to `module` is called *modular*. The package list is determined by a list of modules.

The list of modules that will be put into a variant is defined in the `variants.xml` file. The file can contain either *Name:Stream* or *Name:Stream:Version* references. See Module Naming Policy for details. When *Version* is missing from the specification, Pungi will ask PDC for the latest one.

The module metadata in PDC contains a list of RPMs in the module as well as Koji tag from which the packages can be retrieved.

7.3.1 Restrictions

- A modular compose must always use Koji as a package set source.

CHAPTER

EIGHT

PROCESSING COMPS FILES

The comps file that Pungi takes as input is not really pure comps as used by tools like DNF. There are extensions used to customize how the file is processed.

The first step of Pungi processing is to retrieve the actual file. This can use anything that *Exporting files from SCM* supports.

Pungi extensions are `arch` attribute on `packageref`, `group` and `environment` tags. The value of this attribute is a comma separated list of architectures.

Second step Pungi performs is creating a file for each architecture. This is done by removing all elements with incompatible `arch` attribute. No additional clean up is performed on this file. The resulting file is only used internally for the rest of the compose process.

Third and final step is to create comps file for each Variant.Arch combination. This is the actual file that will be included in the compose. The start file is the original input file, from which all elements with incompatible architecture are removed. Then clean up is performed by removing all empty groups, removing non-existing groups from environments and categories and finally removing empty environments and categories. As a last step groups not listed in the variants file are removed.

CHAPTER

NINE

CONTRIBUTING TO PUNGI

9.1 Set up development environment

In order to work on *Pungi*, you should install recent version of *Fedora*. These packages will have to installed:

- createrepo_c
- cvs
- gcc
- genisoimage
- gettext
- git
- gobject-introspection
- isomd5sum
- jigdo
- kobo
- krb5-devel
- libcurl-devel
- libmodulemd
- libselinux-python
- lorax
- python-dogpile-cache
- python-jsonschema
- python-kickstart
- python-libcomps
- python-lockfile
- python-lxml
- python2-multilib
- python-productmd
- PyYAML
- repoview

- rpm-devel
- syslinux
- yum
- yum-utils

For running unit tests, these packages are recommended as well:

- python-mock
- python-nose
- python-nose-cov
- python-unittest2
- rpmdevtools
- python-parameterized

While being difficult, it is possible to work on *Pungi* using *virtualenv*. Install *python-virtualenvwrapper* (after installation you have to add the command to *source /usr/local/bin/virtualenvwrapper.sh* to your shell startup file, depending on where it was installed by package manager) and use following steps. It will link system libraries into the virtual environment and install all packages preferably from PyPI or from tarball. You will still need to install all of the non-Python packages above as they are used by calling an executable.

```
$ mkvirtualenv pungienv
$ for pkg in gi libcomps pykickstart rpmUtils selinux urlgrabber yum; do ln -vs "
↪$(deactivate && python -c 'import os, '$pkg'; print(os.path.dirname('$pkg'.__file__
↪))')" "$(virtualenvwrapper_get_site_packages_dir)"; done
$ for pkg in _deltarpm krbV _selinux deltarpm sqlitecachec _sqlitecache; do ln -vs "
↪$(deactivate && python -c 'import os, '$pkg'; print('$pkg'.__file__)')" "
↪$(virtualenvwrapper_get_site_packages_dir)"; done
$ pip install -U pip
$ PYCURL_SSL_LIBRARY=nss pip install pycurl --no-binary :all:
$ pip install beanbag jsonschema 'kobo>=0.6.0' lockfile lxml mock nose nose-cov␣
↪productmd pyopenssl python-multilib requests requests-kerberos setuptools sphinx␣
↪ordered_set koji PyYAML dogpile.cache parameterized
```

Now you should be able to run all existing tests.

9.2 Developing

Currently the development workflow for Pungi is on master branch:

- Make your own fork at https://pagure.io/pungi
- Clone your fork locally (replacing $USERNAME with your own):

```
git clone git@pagure.io:forks/$USERNAME/pungi.git
```

- cd into your local clone and add the remote upstream for rebasing:

```
cd pungi
git remote add upstream git@pagure.io:pungi.git
```

Note: This workflow assumes that you never `git commit` directly to the master branch of your fork. This will make more sense when we cover rebasing below.

- create a topic branch based on master:

```
git branch my_topic_branch master
git checkout my_topic_branch
```

- Make edits, changes, add new features, etc. and then make sure to pull from upstream master and rebase before submitting a pull request:

```
# lets just say you edited setup.py for sake of argument
git checkout my_topic_branch

# make changes to setup.py
git add setup.py
git commit -s -m "added awesome feature to setup.py"

# now we rebase
git checkout master
git pull --rebase upstream master
git push origin master
git push origin --tags
git checkout my_topic_branch
git rebase master

# resolve merge conflicts if any as a result of your development in
# your topic branch
git push origin my_topic_branch
```

Note: In order to for your commit to be merged, you must sign-off on it. Use `-s` option when running `git commit`.

- Create pull request in the pagure.io web UI
- For convenience, here is a bash shell function that can be placed in your ~/.bashrc and called such as `pullupstream pungi-4-devel` that will automate a large portion of the rebase steps from above:

```
pullupstream () {
  if [[ -z "$1" ]]; then
    printf "Error: must specify a branch name (e.g. - master, devel)\n"
  else
    pullup_startbranch=$(git describe --contains --all HEAD)
    git checkout $1
    git pull --rebase upstream master
    git push origin $1
    git push origin --tags
    git checkout ${pullup_startbranch}
  fi
}
```

9.3 Testing

You must write unit tests for any new code (except for trivial changes). Any code without sufficient test coverage may not be merged.

To run all existing tests, suggested method is to use *nosetests*. With additional options, it can generate code coverage. To make sure even tests from executable files are run, don't forget to use the `--exe` option.

```
$ make test
$ make test-cover

# Running single test file
$ python tests/test_arch.py [TestCase...]
```

In the `tests/` directory there is a shell script `test_compose.sh` that you can use to try and create a miniature compose on dummy data. The actual data will be created by running `make test-data` in project root.

```
$ make test-data
$ make test-compose
```

This testing compose does not actually use all phases that are available, and there is no checking that the result is correct. It only tells you whether it crashed or not.

Note: Even when it finishes successfully, it may print errors about `repoclosure` on *Server-Gluster.x86_64* in *test* phase. This is not a bug.

9.4 Documenting

You must write documentation for any new features and functional changes. Any code without sufficient documentation may not be merged.

To generate the documentation, run `make doc` in project root.

CHAPTER

TEN

TESTING PUNGI

10.1 Test Data

Tests require test data and not all of it is available in git. You must create test repositories before running the tests:

```
make test-data
```

Requirements: createrepo_c, rpmbuild

10.2 Unit Tests

Unit tests cover functionality of Pungi python modules. You can run all of them at once:

```
make test
```

which is shortcut to:

```
python2 setup.py test
python3 setup.py test
```

You can alternatively run individual tests:

```
cd tests
./<test>.py [<class>[.<test>]]
```

10.3 Functional Tests

Because compose is quite complex process and not everything is covered with unit tests yet, the easiest way how to test if your changes did not break anything badly is to start a compose on a relatively small and well defined package set:

```
cd tests
./test_compose.sh
```

CHAPTER

ELEVEN

MANAGING COMPOSE FROM MULTIPLE PARTS

There may be cases where it makes sense to split a big compose into separate parts, but create a compose output that links all output into one familiar structure.

The *pungi-orchestrate* tools allows that.

It works with an INI-style configuration file. The `[general]` section contains information about identity of the main compose. Other sections define individual parts.

The parts are scheduled to run in parallel, with the minimal amount of serialization. The final compose directory will contain hard-links to the files.

11.1 General settings

target Path to directory where the final compose should be created.

compose_type Type of compose to make.

release_name Name of the product for the final compose.

release_short Short name of the product for the final compose.

release_version Version of the product for the final compose.

release_type Type of the product for the final compose.

extra_args Additional arguments that wil be passed to the child Pungi processes.

koji_profile If specified, a current event will be retrieved from the Koji instance and used for all parts.

kerberos If set to yes, a kerberos ticket will be automatically created at the start. Set keytab and principal as well.

kerberos_keytab Path to keytab file used to create the kerberos ticket.

kerberos_principal Kerberos principal for the ticket

pre_compose_script Commands to execute before first part is started. Can contain multiple commands on separate lines.

post_compose_script Commands to execute after the last part finishes and final status is updated. Can contain multiple commands on separate lines.

```
post_compose_script =
    compose-latest-symlink $COMPOSE_PATH
    custom-post-compose-script.sh
```

Multiple environment variables are defined for the scripts:

- `COMPOSE_PATH`

- COMPOSE_ID
- COMPOSE_DATE
- COMPOSE_TYPE
- COMPOSE_RESPIN
- COMPOSE_LABEL
- RELEASE_ID
- RELEASE_NAME
- RELEASE_SHORT
- RELEASE_VERSION
- RELEASE_TYPE
- RELEASE_IS_LAYERED – YES for layered products, empty otherwise
- BASE_PRODUCT_NAME – only set for layered products
- BASE_PRODUCT_SHORT – only set for layered products
- BASE_PRODUCT_VERSION – only set for layered products
- BASE_PRODUCT_TYPE – only set for layered products

notification_script Executable name (or path to a script) that will be used to send a message once the compose is finished. In order for a valid URL to be included in the message, at least one part must configure path translation that would apply to location of main compose.

Only two messages will be sent, one for start and one for finish (either successful or not).

11.2 Partial compose settings

Each part should have a separate section in the config file.

It can specify these options:

config Path to configuration file that describes this part. If relative, it is resolved relative to the file with parts configuration.

just_phase, skip_phase Customize which phases should run for this part.

depends_on A comma separated list of other parts that must be finished before this part starts.

failable A boolean toggle to mark a part as failable. A failure in such part will mark the final compose as incomplete, but still successful.