
Pungi Documentation

Release 4.0.17

Daniel Mach

June 14, 2016

CONTENTS

1	About Pungi	3
1.1	Tool overview	3
1.2	Links	3
2	Contributing to Pungi	5
2.1	Set up development environment	5
2.2	Developing	6
2.3	Testing	7
2.4	Documenting	8
3	Testing Pungi	9
3.1	Test Data	9
3.2	Unit Tests	9
3.3	Functional Tests	9
4	Configuration	11
4.1	Minimal Config Example	11
4.2	Release	11
4.3	Base Product	12
4.4	General Settings	13
4.5	Image Naming	14
4.6	Signing	15
4.7	Git URLs	16
4.8	Createrepo Settings	16
4.9	Package Set Settings	17
4.10	Buildinstall Settings	17
4.11	Gather Settings	18
4.12	Koji Settings	20
4.13	Extra Files Settings	20
4.14	Productimg Settings	21
4.15	CreateISO Settings	21
4.16	Live Images Settings	22
4.17	Live Media Settings	22
4.18	Image Build Settings	23
4.19	OSTree Settings	25
4.20	Ostree Installer Settings	26
4.21	Media Checksums Settings	27
4.22	Translate Paths Settings	27
5	Progress notification	29
5.1	Setting it up	29

Contents:

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.

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

CONTRIBUTING TO PUNGI

2.1 Set up development environment

In order to work on *Pungi*, you should install *Fedora 23*. These packages will have to installed:

- createrepo
- createrepo_c
- cvs
- genisoimage
- gettext
- git
- isomd5sum
- jigdo
- kobo
- kobo-rpmlib
- koji
- libselinux-python
- lorax
- python-kickstart
- python-lockfile
- python-lxml
- python-productmd
- repoview
- syslinux
- yum
- yum-utils

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

- python-mock
- python-nose
- python-nose-cov

Technically, it is possible to work on *Fedora 22*, but setting it up is a major headache. Packaged version of *productmd* is too old, and *Pungi* does not easily work with *virtualenv*, mostly because many of its Python dependencies are not available on PyPI. To get it working, you have to install *kobo* and *productmd* directly from Git and provide symlinks to other packages and files in `$(virtualenvwrapper_get_site_packages_dir)`. You will still need to install all of the non-Python packages above as they are used by calling an executable.

```
$ ln -vs "$(deactivate && python -c 'import os, koji; print os.path.dirname(koji.__file__)')" "$(virtu
$ ln -vs "$(deactivate && python -c 'import os, rpm; print os.path.dirname(rpm.__file__)')" "$(virtu
$ ln -vs "$(deactivate && python -c 'import os, rpmUtils; print os.path.dirname(rpmUtils.__file__)'
$ ln -vs "$(deactivate && python -c 'import os, yum; print os.path.dirname(yum.__file__)'" "$(virtu
$ ln -vs "$(deactivate && python -c 'import os, urlgrabber; print os.path.dirname(urlgrabber.__file_
$ ln -vs "$(deactivate && python -c 'import os, krbV; print krbV.__file__')" "$(virtualenvwrapper_get
$ ln -vs "$(deactivate && python -c 'import os, sqlitecachec; print sqlitecachec.__file__')" "$(virtu
$ ln -vs "$(deactivate && python -c 'import os, _sqlitecache; print _sqlitecache.__file__')" "$(virtu
$ PYCURL_SSL_LIBRARY=nss pip install pycurl
$ pip install https://git.fedorahosted.org/cgit/kobo.git/snapshot/kobo-0.4.3.tar.gz#egg=kobo
$ pip install https://github.com/release-engineering/productmd/archive/master.tar.gz#egg=productmd
$ pip install lxml pyopenssl mock sphinx setuptools nose nose-cov
```

2.2 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
}
```

2.3 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 *nose* tests. 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-commpose
```

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.

2.4 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.

TESTING PUNGI

3.1 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

3.2 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>]]
```

3.3 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
```


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"

# KOJI
koji_profile = "koji"
runroot = False

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

# CREATEREPO
createrepo_checksum = "sha256"

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

# BUILDINSTALL
bootable = True
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, “ga” or “updates”

release_is_layered = **False** (*bool*) – typically False for an operating system, True otherwise

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"

release_is_layered = True

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

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

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

failable_deliverables [optional] (*list*) – list which deliverables on which variant and architecture can fail and not abort the whole compose

Currently handled deliverables are:

- buildinstall
- iso
- live
- image-build
- live-media
- ostree
- ostree-installer

Note: Image building is not run per-architecture. If you want to mark it as failable, specify it in a block with arch set as `*`.

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

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

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

keep_original_comps [optional] (*list*) – List of variants for which the original comps file will be copied without any modifications. Overwrites *comps_filter_environments*.

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

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 are 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*) – Python’s format string to serve as template for image names

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_valid_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_valid_formats`.

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

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_valid_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_valid_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 usually 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 interactively right after pungi starts.

4.6.1 Example

```
signing_command = '~/git/releng/scripts/sigulsign_unsigned.py -vv --password=%(signing_key_password)s'
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 kickstart file from. This URL is passed on to *Koji*. It is possible to 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 [mandatory] (*str*) – specify checksum type for createrepo; expected values: sha256, sha

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

createrepo_deltas = **False** (*bool*) – generate delta RPMs against an older compose. This needs to be used together with `-old-composes` command line argument.

4.8.2 Example

```
createrepo_checksum = "sha256"
```

4.9 Package Set Settings

4.9.1 Options

sigkeys (*[str or None]*) – priority list of sigkeys, *None* means unsigned

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

pkgset_koji_tag **[mandatory]** (*str*) – tag to read package set from

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

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:

bootable (*bool*) – whether to run the buildinstall phase

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

buildinstall_upgrade_image **[deprecated]** (*bool*) – use `noupgrade` with `lorax_options` instead

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`)

4.10.2 Example

```
bootable = True
buildinstall_method = "lorax"

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

```
    }
    "*" : {
        "noupgrade": False
    }
})
]
```

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_source [mandatory] (*str*) – from where to read initial package list; expected values: “comps”, “none”

gather_method [mandatory] (*str*) – “deps”, “nodeps”

greedy_method (*str*) – see `gather`, recommended value: “build”

multilib_methods [deprecated] (*[str]*) – use `multilib` instead to configure this per-variant

multilib_arches [deprecated] (*[str]* or `None`) – use `multilib` to implicitly configure this: if a variant on any arch has non-empty multilib methods, it is automatically eligible

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

Available methods are:

- none
- all
- runtime
- file
- kernel
- yaboot

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

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

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`.

multilib_blacklist (*dict*) – multilib blacklist; format: `{arch|*: [package_globs]}`. The patterns are tested with `fnmatch`, so shell globbing is used (not regular expression).

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

4.11.2 Example

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

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-*",
    ],
}

# 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

runroot [**mandatory**] (*bool*) – run some tasks such as `buildinstall` or `createiso` in koji build root (True) or locally (False)

runroot_channel (*str*) – name of koji channel

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

4.12.2 Example

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

4.13 Extra Files Settings

4.13.1 Options

extra_files (*list*) – references to external files to be placed in `os/` directory and media; format: `[(variant_uid_regex, {arch!*: [scm_dicts]})]`

4.13.2 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 Productimg Settings

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

4.14.1 Options

productimg = False (*bool*) – create product images; requires `bootable=True`

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

4.14.2 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",
}

```

4.15 CreateISO Settings

4.15.1 Options

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

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

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

4.15.2 Example

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

4.16 Live Images Settings

live_target (*str*) – Koji build target for which to build the images. This gets passed to `koji spin-livectd`.

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*)
- **additional_repos** (*list*) – external repos specified by URL
- **repo_from** (*list*) – repos from other variants
- **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-livectd` 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.

4.17 Live Media Settings

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

Required options:

- **name** (*str*)
- **version** (*str*)
- **target** (*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)`
- `release (str)` – a string with the release, or explicit `None` for using compose date and respin.
- `skip_tag (bool)`
- `repo ([str])` – external repo
- `repo_from ([str])` – list of variants to take extra repos from
- `title (str)`
- `install_tree_from (str)` – variant to take install tree from

If many of your media use the same value for one of `ksurl`, `release`, `target` or `version`, consider using these options to set the value in one place and have all media inherit it.

live_media_ksurl (*str*) – Provides a fallback for media that do not specify `ksurl` in the `live_media` block.

live_media_release (*str*) – Provides a fallback for media that do not specify `release` in the `live_media` block. Please note that if you set this, there is no way to unset it for a particular media. This is important if you want the release generated by Koji.

live_media_target (*str*) – Provides a fallback for media that do not specify `target` in the `live_media` block.

live_media_version (*str*) – Provides a fallback for media that do not specify `version` in the `live_media` block.

4.18 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 `None`, it will be replaced with `DATE.RESPIN` of the current compose.

You can also add extra variants to get repos from with key `repo_from`. The value should be a list of variant names.

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.

The format attr is `[('image_type', 'image_suffix'), ...]`. See `productmd` documentation for list of supported types and suffixes.

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.

4.18.1 Example

```
image_build = {
  '^Server$': [
    {
      'image-build': {
        'format': [('docker', 'tar.gz'), ('qcow2', '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',
          'Vendor': 'Fedora Project',
          'Version': '23',
          'Architecture': 'x86_64'
        }",
      },
    },
    {
      'image-build': {
        'format': [('docker', 'tar.gz'), ('qcow2', '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', 'qcow2')]
        'name': 'fedora-qcow-base',
        'target': 'koji-target-name',
        'ksversion': 'F23',      # value from pykickstart
```

```

        'version': '23',
        '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_from': ['Everything'],

        # Set release based on date and respin.
        'release': None,
    }
}
]
}

```

4.19 OSTree Settings

The *ostree* phase of *Pungi* can create ostree repositories in a Koji runroot environment.

ostree (*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 these keys:

- `treefile` – (*str*) Filename of configuration for rpm-ostree.
- `config_url` – (*str*) URL for Git repository with the `treefile`.
- `source_repo_from` – (*str*) Name of variant serving as source repository.
- `ostree_repo` – (*str*) Where to put the ostree repository

These keys are optional:

- `config_branch` – (*str*) Git branch of the repo to use. Defaults to `master`.

4.19.1 Example config

```

ostree = [
    ("^Atomic$", {
        "x86_64": {
            "treefile": "fedora-atomic-docker-host.json",
            "config_url": "https://git.fedorahosted.org/git/fedora-atomic.git",
            "source_repo_from": "Everything",
            "ostree_repo": "/mnt/koji/compose/atomic/Rawhide/"
        }
    })
]

```

4.20 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:

- `source_repo_from` – (*str*) Name of variant serving as source repository or a URL pointing the the repo.

These keys are optional:

- `release` – (*str*) Release value to set for the installer image. Set to `None` to use the `date.respin` format.

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*)
- `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.

4.20.1 Example config

```
ostree_installer = [
    ("^Atomic$", {
        "x86_64": {
            "source_repo_from": "Everything",
            "release": None,
            "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',
    })
]
```

4.21 Media Checksums Settings

media_checksums (*list*) – list of checksum types to compute, allowed values are md5, sha1 and sha256

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
- version

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

4.22 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()`.

4.22.1 Example config

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

4.22.2 Example usage

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


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.

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

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

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

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

5.1 Setting it up

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

```
--notification_script=pungi-fedmsg-notification
```


INDICES AND TABLES

- *genindex*
- *modindex*
- *search*