Repoman Documentation

Release 1.2

David Caro

Dec 19, 2018

Contents

CHAPTER 1

Getting started

1.1 Development Environment Setup

To run the tests you need to have installed in your system a reasonable new version of 'tox' and the required project dependencies that are not available in PyPI.

See project's Dockerfile or requirements.txt to get idea which distribution packages are needed. The list is created and maintained for Fedora or CentOS with EPEL enabled.

1.2 Building and Running Tests

To build and run unit tests, run 'tox' from the top dir of the project:

```
tox
```

There is also a functional test suite that can be run by specifying additional 'tox' environment parameter:

```
tox -e functional
```

1.3 Building the Documentation

To build the docs, you can use 'docs' environment:

```
tox -e docs
```

That will leave the docs under the path `docs/_build/html/index.html`

1.4 Docker Development Image

This project requires a set of dependencies to be installed on a system. To help with that for development there is a pre-built docker image available based on the CentOS 7 official image. The development image includes everything necessary to build repoman, run unit and functional tests and generate docs.

By default, the container will cd to /mnt directory where it expects the root of repoman source code tree to be mounted, but will not run any command (there is no entry point explicitly defined). Also it does not require root privileges and is convenient to run under your UID, so you can use your working tree.

E.g. to run just 'tox' you can use:

```
sudo docker run -v /path/to/repoman/source:/mnt -u $UID -t -i marchukov/repoman-tox-
↪env-centos7 tox
```

You can pass 'tox' parameters to the container as usual, e.g. to run the functional test suite:

```
sudo docker run -v /path/to/repoman/source:/mnt -u $UID -t -i marchukov/repoman-tox-
↪env-centos7 tox -e functional
```

Since each run will create a new container, when debugging requires multiple runs, it is convenient to reuse a single container with an interactive shell:

```
sudo docker run -v /path/to/repoman/source:/mnt -u $UID -t -i marchukov/repoman-tox-
↪env-centos7 bash
```

This will give you a shell and you can then run 'tox' there or do whatever else is necessary.

1.5 Basic Tutorial

In order to get you started using repoman, here's a quick tutorial showing some of the most commonly used features, remember that for extra help on all the configuration options supported and all the sources and filters you can search the api docs or run the *docs* subcommand to get info about it:

```
repoman dummyrepo docs -h
```

1.5.1 Adding the packages of a local dir to an existing repo

This is the most simple use case, imagine that you have a directory with some artifacts, and you want to add it to an already existing repository, you can just run:

```
repoman /path/to/existing/repository add /path/to/dir/with/extra/artifacts
```

That will make sure that the artifacts in the directory */path/to/dir/with/extra/artifacts* are added to */path/to/existing/repository* and will make sure that they are added in an ordered manner, and update any metadata needed by the repos (like yum repository metadata).

1.5.2 Adding other sources of packages to an existing repo

Now let's try to use other types of sources for the artifacts, here is an example:

```
repoman /path/to/existing/repo add \
    koji:@some-tag \
    recursive:http://my.home.page/somepath \
    http://jenkins.ovirt.org/job/repoman_master_build-artifacts-el6-x86_64/ \
    http://koji.fedoraproject.org/koji/buildinfo?buildID=767073 \
    https://copr.fedorainfracloud.org/coprs/msivak/ovirt-optimizer-for-ovirt-4.0/
↪build/466823/
```

As you can see you can specify more than one source at a time, let's take a small look to the ones here:

- *koji:@some-tag*: this will go to koji, and retrieve any package tha matches the given tag
- *recursive:http://my.home.page/somepath*: this will recursively search in that web page, for link to artifacts, and retrieve all of them
- *http://jenkins.ovirt.org/job/repoman_master_build-artifacts-el6-x86_64/*: this will retrive all the artifacts from the latest successful build for that jenkins job.
- *http://koji.fedoraproject.org/koji/buildinfo?buildID=767073*: this will retrieve all the artifacts for the given koji build
- *https://copr.fedorainfracloud.org/coprs/msivak/ovirt-optimizer-for-ovirt-4.0/build/466823/*: this will retrieve all the artifacts for the given copr build

1.5.3 Filtering some of the sources

Another useful tool that repoman gives you, is the ability to filter out the artifacts that a source will expand to, that is done by appending one or more ':' separated filter strings to the source, for example:

```
repoman /path/to/existing/repo add \
    /path/to/dir/with/extra/artifacts:latest=2
```

There it will only select the artifacts with the two highest versions from the */path/to/dir/with/extra/artifacts* source, ignoring any other lower version ones, and only add those high version artifacts to the */path/to/existing/repo* repository.

You can add more than one filter to the source, and they will be applied from rightmost (outer) to leftmost (inner).

In the below example it would first filter by name (regexp) and then by version, getting only the latest one (the default for the latest filter):

```
repoman /path/to/existing/repo add \
    /path/to/dir/with/extra/artifacts:latest:name~repoman.*
```

To exclude packages, you can use the regex 'Negative Lookahead' format '^(?!<package>)' In the below example it would exclude the yum-plugin rpm:

```
repoman /path/to/existing/repo add \
    /path/to/dir/with/extra/artifacts:name~^(?!yum-plugin).*\.rpm
```

1.5.4 Getting the sources from a conf file

So, imagine that you have a bunch of different sources you want to add to a repo, having to write them down as arguments in the command line is too troublesome and has it's limitations, to overcome that issue, repoman allows you to write the sources down inside a file, and them just refernce it with the *conf:* metasource, for example, if you have a file named *mysources*:

```
## These are some useful souces for a release
# get all the packages for a koji tag
koji:@some-tag

# we will need also all the packages under this page (recursively)
recursive:http://my.home.page/somepath

# And repoman latest successful build on el6
http://jenkins.ovirt.org/job/repoman_master_build-artifacts-el6-x86_64/
```

So, if you have that file in your current directory, you can just run:

```
repoman /path/to/existing/repo add conf:mysources
```

And repoman will read that file, discard any comments or empty lines, and use any sources defined there as if they were specifyed by command line.

1.5.5 Specifying some custom repoman config options

So, imagine that you want to tweak some default options, for example, you want to force that when addin an rpm, if the distro can't be guessed from the release, for it to be added to all the known distributions of the repo. So to do that, you can use repoman –option:

```
repoman --option=store.RPMStore.on_wrong_distro=copy_to_all \
    /path/to/existing/repo add conf:mysources
```

Remember that to see all the confi options you can check with the docs subcommand like this:

```
repoman dummyrepo docs config
```

And for details on what each value means, you can go to the specific section docs, for example, for the option we added:

```
repoman dummyrepo docs stores RPMStore
```

1.5.6 Specifying a custom repoman config file

The same way as when specifying the sources, having to specify the options in the command line might be a burden, luckily we can also write the config options down in a file, and just use that file, for example, if we have a file called *custom.config*:

```
# Some config overrides
[store.RPMStore]
on_wrong_distro = copy_to_all
```

Then we can call repoman with the -cl–config option like this:

```
repoman --config=custom.config \
    /path/to/existing/repo add conf:mysources
```

CHAPTER 2

Contents

2.1 repoman package

2.1.1 Subpackages

repoman.common package

Subpackages

repoman.common.filters package

Submodules

repoman.common.filters.latest module

Usage:

```
source:latest
source:latest=N
```

Get's the latest N rpms (1 by default)

class repoman.common.filters.latest.**LatestFilter**(*config*, *stores*)

Bases: repoman.common.filters.ArtifactFilter

Usage:

```
source:latest
source:latest=N
```

Get's the latest N rpms (1 by default)

CONFIG_SECTION = 'LatestFilter'

DEFAULT_CONFIG = {}

filter (*filters_str*, *art_list*)
Filters the given art_list according to filter_str and config

Parameters

- **filter_str** – string with the filter or filters to apply
- **art_list** – list of expanded artifacts

repoman.common.filters.name module

Usage:

```
source:name~regexp
```

Filter packges by file name, for example:

```
http://myhost.com/packages/:name~vdsm.*
```

Will match all the packages in that url that have vdsm.* as name (will not match any previous path in the url)

class repoman.common.filters.name.**NameFilter** (*config*, *stores*)
Bases: repoman.common.filters.ArtifactFilter

Usage:

```
source:name~regexp
```

Filter packges by file name, for example:

```
http://myhost.com/packages/:name~vdsm.*
```

Will match all the packages in that url that have vdsm.* as name (will not match any previous path in the url)

CONFIG_SECTION = 'NameFilter'

DEFAULT_CONFIG = {}

filter (*filters_str*, *art_list*)
Filters the given art_list according to filter_str and config

Parameters

- **filter_str** – string with the filter or filters to apply
- **art_list** – list of expanded artifacts

repoman.common.filters.only_missing module

Usage:

```
source:only-missing
```

Gets only the artifacts that are not already there, getting only the ones that don't have already an artifact with the same name in the repo.

It will take only the latest from the source repo if there are multiple versions available

class repoman.common.filters.only_missing.**OnlyMissingFilter**(*config*, *stores*)
Bases: repoman.common.filters.ArtifactFilter

Usage:

```
source:only-missing
```

Gets only the artifacts that are not already there, getting only the ones that don't have already an artifact with the same name in the repo.

It will take only the latest from the source repo if there are multiple versions available

CONFIG_SECTION = 'OnlyMissingFilter'

DEFAULT_CONFIG = {}

filter(*filters_str*, *art_list*)
Filters the given art_list according to filter_str and config

Parameters

- **filter_str** – string with the filter or filters to apply
- **art_list** – list of expanded artifacts

repoman.common.sources package

Submodules

repoman.common.sources.copr module

Usage:

```
https?://${copr_host}/*
```

Handles copr build urls

class repoman.common.sources.copr.**CoprURLSource**(*config*, *stores*)
Bases: repoman.common.sources.ArtifactSource

Usage:

```
https?://${copr_host}/*
```

Handles copr build urls

CONFIG_SECTION = 'CoprURLSource'

DEFAULT_CONFIG = {'copr_host_re': 'copr\\.fedorainfracloud\\.org'}

expand(*source_str*)
Gets a source string and expands it to it's elements.

classmethod formats_list()
Returns a list of the supported string formats, used for documentation

repoman.common.sources.dir module

Usage:

```
dir_path
file_path
dir:repo_name
```

Will find all the matching artifacts under the specified dir or the given file. If relative path passed, it will be relative to the base_repos_path config value

Configuration values:

- **allowed_dir_paths** Comma separated list of paths or empty string. If set, will not allow using any path/dir/repo from outside of those paths.

class repoman.common.sources.dir.**DirSource**(*config*, *stores*)
Bases: repoman.common.sources.ArtifactSource

Usage:

```
dir_path
file_path
dir:repo_name
```

Will find all the matching artifacts under the specified dir or the given file. If relative path passed, it will be relative to the base_repos_path config value

Configuration values:

- **allowed_dir_paths** Comma separated list of paths or empty string. If set, will not allow using any path/dir/repo from outside of those paths.

CONFIG_SECTION = 'DirSource'

DEFAULT_CONFIG = {'allowed_dir_paths': ''}

check_if_allowed(*path*)

expand(*source_str*)
Gets a source string and expands it to it's elements.

classmethod formats_list()
Returns a list of the supported string formats, used for documentation

static is_allowed(*path*, *allowed_paths*)

resolve_path(*path*)

repoman.common.sources.jenkins module

Allows you to define a jenkins build job url as a source:

- If it's a build -> the artifacts archived on that build
- If it's a job -> the artifacts from the last successful build
- If it's a multiconfig build -> the artifacts from all the configs

For example::

repoman myrepo add http://jenkins.ovirt.org/jobs/lago_master_build-artifacts-el7-x86_64

will get the latest successful build artifacts for that job.

Keep in mind that if the url does not match the regexp in the config, you can still force repoman to use this source prepending the url with 'jenkins:', like this:

```
repoman myrepo add \
    jenkins:http://some.strange.url/to/my_job
```

class repoman.common.sources.jenkins.**JenkinsSource**(*config*, *stores*)

Bases: repoman.common.sources.ArtifactSource

Allows you to define a jenkins build job url as a source:

- If it's a build -> the artifacts archived on that build
- If it's a job -> the artifacts from the last successful build
- If it's a multiconfig build -> the artifacts from all the configs

For example::

repoman myrepo add http://jenkins.ovirt.org/jobs/lago_master_build-artifacts-el7-x86_64

will get the latest successful build artifacts for that job.

Keep in mind that if the url does not match the regexp in the config, you can still force repoman to use this source prepending the url with 'jenkins:', like this:

```
repoman myrepo add \
    jenkins:http://some.strange.url/to/my_job
```

CONFIG_SECTION = 'JenkinsSource'

DEFAULT_CONFIG = {'jenkins_host_re': 'jenkins\\.ovirt\\.org'}

expand(*source_str*)

Gets a source string and expands it to it's elements.

classmethod formats_list()

Returns a list of the supported string formats, used for documentation

repoman.common.sources.kojibuild module

Usage:

```
koji:name@tag
koji:@tag[@inherit]
koji:name-version-release
```

Handles koji builds

class repoman.common.sources.kojibuild.**KojiBuildSource**(*config*, *stores*)

Bases: repoman.common.sources.ArtifactSource

Usage:

```
koji:name@tag
koji:@tag[@inherit]
koji:name-version-release
```

Handles koji builds

CONFIG_SECTION = 'KojiBuildSource'

DEFAULT_CONFIG = {'koji_extra_opts': 'krbservice=host', 'koji_server': 'https://koji

expand(*source_str*)
Gets a source string and expands it to it's elements.

classmethod formats_list()
Returns a list of the supported string formats, used for documentation

repoman.common.sources.kojiurl module

Usage:

```
https?://${koji_host}/*
```

Handles koji build urls

TODO: Improve the second level detections (now filtering buildArch and buildSRPM links only)

class repoman.common.sources.kojiurl.**KojiURLSource**(*config*, *stores*)
Bases: repoman.common.sources.ArtifactSource

Usage:

```
https?://${koji_host}/*
```

Handles koji build urls

TODO: Improve the second level detections (now filtering buildArch and buildSRPM links only)

CONFIG_SECTION = 'KojiURLSource'

DEFAULT_CONFIG = {'koji_host_re': 'koji\\.fedoraproject\\.org'}

expand(*source_str*)
Gets a source string and expands it to it's elements.

classmethod formats_list()
Returns a list of the supported string formats, used for documentation

repoman.common.sources.url module

Parse a url, recursively or not.

Usage:

```
URL -> Will parse the url and get all the packages in that page
rec:URL -> Will parse the urls recursively
```

class repoman.common.sources.url.**URLSource**(*config*, *stores*)
Bases: repoman.common.sources.ArtifactSource

Parse a url, recursively or not.

Usage:

```
URL -> Will parse the url and get all the packages in that page
rec:URL -> Will parse the urls recursively
```

CONFIG_SECTION = 'URLSource'

DEFAULT_CONFIG = {}

expand(*source_str*)
Gets a source string and expands it to it's elements.

expand_page(*page_url*)

expand_recursive(*page_url*, *level=0*)

classmethod formats_list()
Returns a list of the supported string formats, used for documentation

static get_link(*page_url*, *link_url*, *internal=False*)

static strip_qs(*url*)

repoman.common.stores package

Subpackages

repoman.common.stores.RPM package

This module holds the class and methods to manage an rpm store and it's sources.

In our case an rpm store is not just a yum repository but a set of them and src files, in the following structure:

```
repository_dir
├── rpm
│   ├── $dist1  <- this is a yum repository
│   │   ├── repodata
│   │   │   └── ...
│   │   ├── SRPMS
│   │   │   ├── $srcrpm1
│   │   │   ├── $srcrpm2
│   │   │   └── ...
│   │   ├── $arch1
│   │   │   ├── $rpm1
│   │   │   ├── $rpm2
│   │   │   └── ...
│   │   ├── $arch2
│   │   └── ...
│   ├── $dist2  <- another yum reposiory
│   │   └── ...
│   └── ...
└── src
    ├── $project1
    │   │   ├── $source1
    │   │   ├── $source1.sig
    │   │   ├── $source2
    │   │   └── ...
    │   └── ...
    └── ...
```

exception repoman.common.stores.RPM.**CreaterepoError**
Bases: exceptions.Exception

exception repoman.common.stores.RPM.**CreatereposError**
Bases: exceptions.Exception

class repoman.common.stores.RPM.**RPMStore**(*config*, *repo_path=None*)
Bases: repoman.common.stores.ArtifactStore

Represents the repository sctructure, it does not require that the repo has the structure specified in the module doc when loading it, but when adding new rpms or generating the sources it will create the new files in that directory structure.

You can pass rpm properties (like version, distro, arch or major_version) as python's format variables and they will be expanded at runtime for each rpm using the expansion as store path, for example '/myrepo/{major_version}' will create all the repository structure under that path for each rpm (if you have multiple independent rpms that does not make much senes though, but you get the idea)

Configuration options:

- **distro_reg** Regular expression to extract the distribution from the release string
- **extra_symlinks** Comma separated list of orig:symlink pairs to create links, the paths
- **on_wrong_distro** Action to execute when a package has an incorrect distro (it's release string does not match the distro_reg regular expression). Possible values are 'fail', 'copy_to_all' or anything else. The default is 'fail', if 'copy_to_all' specified it will copy the rpm to all the distros (it needs to have any other distros in the dst repo, or other rpms with a defined distro). If anything else specified, it will warn and skip that rpm.
- **path_prefix** Prefixes of this store inside the globl artifact repository, separated by commas
- **rpm_dir** name of the directory that will contain the rpms (rpm by default), if empty, it will not create a subdirectory for the rpms and will be put on the root of the repo (root/$dist/$arch/*rpm)
- **signing_key** Path to the gpg keey to sign the rpms with, will not sign them if not set
- **signing_passphrase** Passphrase for the above key
- **temp_dir** Temporary dir to store any transient downloads (like rpms from urls). The caller should make sure it exists and clean it up if needed.
- **with_sources** If true, will extract the sources form the scrrpms
- **with_srcrpms** If false, will ignore the srcrpms will be relative to the store root path.

CONFIG_SECTION = 'RPMStore'

DEFAULT_CONFIG = {'distro_reg': '\\.(fc|el)\\d+(?=\\w*)', 'extra_symlinks': '', 'on

add_artifact(*pkg*, ***args*)
This method adds an artifact to the store

Parameters **artifact** – full path or url to the artifact

add_rpm(*pkg*, *onlyifnewer=False*, *to_copy=True*, *hidelog=False*)
Generic functon to add an rpm package to the repo.

Parameters

- **pkg** – path or url to the rpm file to add
- **onlyifnewer** – If set to True, will only add the package if it's not there already or the version is newer than the on already there.
- **to_copy** – If set to True, will add that package to the list of packages to copy into the repo when saving, usually used when adding new packages to the repo.

- **hidelog** – If set to True will not show the extra information (used when loading a repository to avoid verbose output)

change_path(*new_path*)
Changes the store path to the given one, copying any artifacts if needed

Parameters **new_path** (*str*) – New path to set

Returns None

create_symlinks()
Creates all the symlinks to the dirs passed on the config

static createrepo(*dst_dir*)

createrepos()
Generate the yum repositories metadata

delete_old(*keep=1*, *noop=False*)
Delete the oldest versions for each package from the repo

Parameters

- **keep** – Maximium number of versions to keep of each package
- **noop** – If set, will only log what will be done, not actually doing anything.

generate_sources(*with_patches=False*, *key=None*, *passphrase=None*)
Generate the sources directory from all the srcrpms

Parameters

- **with_patches** – If set, will also extract the .patch files from the srcrpm
- **key** – If set to the path of a gpg key, will use that key to create the detached signatures of the extracted sources
- **passphrase** – Passphrase to unlock the key

get_artifacts(*regmatch=None*, *fmatch=None*)
Returns the list of artifacts matching the params

Parameters

- **regmatch** – Regular expression to filter the rpms path with
- **fmatch** – Filter function, must return True for packages to be included, or False to be excluded. The package object will be passed as parameter

get_latest(*regmatch=None*, *fmatch=None*, *num=1*)
Return the num latest versions for each rpm in the repo

Parameters **num** – number of latest versions to return

Return type *repoman.common.artifact.Artifact*

get_rpms(*regmatch=None*, *fmatch=None*, *latest=0*)
Get the list of rpms, filtered or not.

Parameters

- **regmatch** – Regular expression that will be applied to the path of each package to filter it
- **fmatch** – Filter function that must return True for a package to be selected, will be passed the RPM object as only parameter

- **latest** – If set to N>0, it will return only the N latest versions for each package

get_store_path(*pkg*)

handles_artifact(*artifact*)
This method must return True if the given artifact (as a path or url) can be handled by the implemented store

Parameters **artifact_str** – full path or url to the artifact

is_latest_version(*pkg*)
Check if the given package is the latest version in the repo :pram pkg: RPM instance of the package to compare

path_prefix

save(***args*)
Realizes the changes made to the store, usually writing the artifacts to disk or any other operation required to persist the store state

sign_rpms()
Sign all the unsigned rpms in the repo.

Submodules

repoman.common.stores.RPM.RPM module

This module holds the helper classes to represent a repository, that in our case (oVirt) is a set of repositories, in the form:

```
Base_dir
├── rpm
│   └── $dist
│       ├── repodata
│       ├── SRPMS
│       └── $arch
└── src
    └── $name
        ├── $name-$version-src.tar.gz
        └── $name-$version-src.tar.gz.sig
```

This module has the classess that manage a set of rpms, ina hierarchical fashion, in the order:

```
name 1-* version 1-* inode 1-* rpm-instance
```

So that translated to classes, with the first being the placeholder for the whole data structure, is:

```
RPMList 1-* RPMName 1-* RPMVersion 1-* RPMInode 1-* RPM
```

All except the RPM class are implemented as subclasses of the python dict, so as key-value stores.

For clarification, here's a dictionary like diagram:

```
RPMList{
    name1: RPMName{
        version1: RPMVersion{
            inode1: RPMInode[RPM, RPM, ...]
            inode2: RPMInode[...]
```

(continues on next page)

(continued from previous page)

```
        },
        version2: RPMVersion{...}
    },
    name2: RPMName{...}
}
```

class repoman.common.stores.RPM.RPM.**RPM**(*path*, *temp_dir='/tmp'*, *distro_reg='\.(fc|el)\d+'*, *to_all_distros=()*, *verify_ssl=True*)

Bases: *repoman.common.artifact.Artifact*

__str__()

This string uniquely identifies a rpm file, if two rpms have the same string representation, the must point to the same file or a copy of it, if not, you wrongly generated two rpms with the same version/release and different content, or you signed them with different keys

extension

full_name

Unique RPM Name.

This property should uniquely identify a rpm entity, in the sense that if you have two rpms with the same full_name they must package the same content or one of them is wrongly generated (the version was not bumped or something).

generate_path(*base_dir='rpm'*)

Returns the theoretical path that the rpm should be, instead of the current path it is. As explained at the module docs.

If the package has to go to all distros, a placeholder for it will be set in the string

static get_distro(*release*, *distro_reg*)

name

sign(*key_path*, *passwd*)

Defines how to sign this artifact, by default with detached signature

type

version

class repoman.common.stores.RPM.RPM.**RPMList**(*name_class=<class 'repoman.common.stores.RPM.RPM.RPMName'>*)

Bases: *repoman.common.artifact.ArtifactList*

List of rpms, separated by name

class repoman.common.stores.RPM.RPM.**RPMName**(*name*, *version_class=<class 'repoman.common.artifact.ArtifactVersion'>*)

Bases: *repoman.common.artifact.ArtifactName*

List of available versions for a package name

add_pkg(*pkg*, *onlyifnewer*)

get_latest(*num=1*)

Returns the list of available inodes for the latest version if any

exception repoman.common.stores.RPM.RPM.**WrongDistroException**

Bases: exceptions.Exception

Submodules

repoman.common.stores.iso module

This module holds the class and methods to manage an iso store:

```
repository_dir
└── iso
    ├── $project1
    │   ├── $version
    │   |   ├── $iso1
    |   |   ├── $iso1.md5sum
    |   |   └── $iso1.md5sum.sig
    |   |   └── ...
    |   └── ...
    └── ...
```

class repoman.common.stores.iso.**Iso**(*path*, *temp_dir*, *verify_ssl=True*)
Bases: *repoman.common.artifact.Artifact*

extension

full_name
Unique ISO Name.

This property should uniquely identify an ISO entity, in the sense that if you have two isos with the same full_name they must package the same content or one of them is wrongly generated (the version was not bumped or something).

name

sign(*key*, *passwd*)
Defines how to sign this artifact, by default with detached signature

type

version

class repoman.common.stores.iso.**IsoStore**(*config*, *repo_path=None*)
Bases: repoman.common.stores.ArtifactStore

Represents the repository sctructure, it does not require that the repo has the structure specified in the module doc when loading it, but when adding new isos it will create the new files in that directory structure.

Configuration options:

- **temp_dir** Temporary dir to store any transient downloads (like isos from urls). The caller should make sure it exists and clean it up if needed.
- **path_prefix** Prefixes of this store inside the globl artifact repository, separated by commas
- **signing_key** Path to the gpg keey to sign the isos with, will not sign them if not set
- **signing_passphrase** Passphrase for the above key

CONFIG_SECTION = 'IsoStore'

DEFAULT_CONFIG = {'path_prefix': 'iso', 'signing_key': '', 'signing_passphrase': '

add_artifact(*iso*, ***args*)
This method adds an artifact to the store

Parameters **artifact** – full path or url to the artifact

add_iso(*iso*, *onlyifnewer=False*, *to_copy=True*, *hidelog=False*)
Generic functon to add an iso package to the repo.

Parameters

- **iso** – path or url to the iso file to add
- **onlyifnewer** – If set to True, will only add the package if it's not there already or the version is newer than the on already there.
- **to_copy** – If set to True, will add that package to the list of packages to copy into the repo when saving, usually used when adding new packages to the repo.
- **hidelog** – If set to True will not show the extra information (used when loading a repository to avoid verbose output)

change_path(*new_path*)
Changes the store path to the given one, copying any artifacts if needed

Parameters **new_path** (*str*) – New path to set

Returns None

delete_old(*keep=1*, *noop=False*)
Delete the oldest versions for each package from the repo

Parameters

- **keep** – Maximium number of versions to keep of each package
- **noop** – If set, will only log what will be done, not actually doing anything.

get_artifacts(*regmatch=None*, *fmatch=None*)
Get the list of isos, filtered or not.

Parameters

- **regmatch** – Regular expression that will be applied to the path of each package to filter it
- **fmatch** – Filter function that must return True for a package to be selected, will be passed the iso object as only parameter

get_latest(*regmatch=None*, *fmatch=None*, *num=1*)
Return the num latest versions for each iso in the repo

Parameters **num** – number of latest versions to return

handles_artifact(*artifact_str*)
This method must return True if the given artifact (as a path or url) can be handled by the implemented store

Parameters **artifact_str** – full path or url to the artifact

is_latest_version(*iso*)
Check if the given iso is the latest version in the repo

Parameters **iso** – ISO instance of the package to compare

path_prefix

save(***args*)
Realizes the changes made to the store, usually writing the artifacts to disk or any other operation required to persist the store state

sign_isos()
Sign all the isos in the repo.

exception repoman.common.stores.iso.**WrongIsoError**
Bases: exceptions.Exception

Any iso failure

Submodules

repoman.common.artifact module

This module holds the helper classes to represent an artifact list:

```
Base_dir
└── $name
    └── $version
        ├── $name-$version.$extension
        └── $name-$version.$extension.sig
```

This module has the classess that manage a set of artifacts, in a hierarchical fashion, in the order:

```
name 1-* version 1-* inode 1-* artifact-instance
```

So that translated to classes, with the first being the placeholder for the whole data structure, is:

```
ArtifactList 1-* ArtifactName 1-* ArtifactVersion \
1-* ArtifactInode 1-* Artifact
```

All except the Artifact class are implemented as subclasses of the python dict, so as key-value stores.

For clarification, here's a dictionary like diagram:

```
ArtifactList{
    name1: ArtifactName{
        version1: ArtifactVersion{
            inode1: ArtifactInode[Artifact, Artifact, ...]
            inode2: ArtifactInode[...]
        },
        version2: ArtifactVersion{...}
    },
    name2: ArtifactName{...}
}
```

NOTE:You have to implement at least the Artifact class

class repoman.common.artifact.**Artifact**(*path*, *temp_dir='/tmp'*, *verify_ssl=True*)
Bases: object

__str__()
This string uniquely identifies a artifact file, if two have the same string representation, the must point to the same file or a copy of it, if not, you wrongly generated two artifact with the same version/name and different content

extension

full_name
Unique Artifact Name.

This property should uniquely identify an artifact entity, in the sense that if you have two artifacts with the same full_name they must package the same content or one of them is wrongly generated (the version was not bumped or something).

generate_path()
Returns the theoretical path that the artifact should be, instead of the current path it is.

md5
Lazy md5 calculation.

name

sign(*key_path*, *passwd*)
Defines how to sign this artifact, by default with detached signature

type

version

class repoman.common.artifact.**ArtifactInode**(*inode*)
Bases: list, object

Simple list, abstracts a set of rpm instances

delete(*noop=False*)

get_artifacts(*regmatch=None*, *fmatch=None*)

class repoman.common.artifact.**ArtifactList**(*name*, *name_class=<class 'repoman.common.artifact.ArtifactName'>*)
Bases: dict, object

Dict of artifacts, by name

add_pkg(*artifact*, *onlyifnewer=False*)

delete()
Deletes all the artifacts in this list

delete_version(*art_name*, *art_version*)
Removes the given artifact's version if it's in the list

Parameters

- **art_name** (*str*) – Name of the artifact to remove it's version
- **art_version** (*str*) – Version to remove

Returns None

get_artifacts(*regmatch=None*, *fmatch=None*, *latest=0*)
Gets the list of artifacts, filtered or not.

Parameters

- **regmatch** – Regular expression to filter the rpms path with
- **fmatch** – Filter function, must return True for packages to be included, or False to be excluded. The package object will be passed as parameter
- **latest** – number of latest versions to return (0 for all,)

class repoman.common.artifact.**ArtifactName**(*name*, *version_class=<class 'repoman.common.artifact.ArtifactVersion'>*)
Bases: dict, object

Dict of available versions for an artifact name

add_artifact(*artifact*, *onlyifnewer*)

delete(*noop=False*)

delete_version(*version*, *noop=False*)

get_artifacts(*regmatch=None*, *fmatch=None*, *latest=0*)

get_latest(*num=1*)
Returns the list of available inodes for the latest version if any

class repoman.common.artifact.**ArtifactVersion**(*version*, *inode_class=<class 'repoman.common.artifact.ArtifactInode'>*)
Bases: dict, object

Abstracts a set of artifacts inodes for a version

add_artifact(*artifact*)

delete(*noop=False*)

delete_inode(*inode*, *noop=False*)

get_artifacts(*regmatch=None*, *fmatch=None*)

repoman.common.config module

exception repoman.common.config.**BadConfigError**
Bases: exceptions.Exception

class repoman.common.config.**Config**(*path=None*, *section='main'*)
Bases: object

Configuration object to wrap some config values. It keeps the configuration objects, one with the default values for all the sections and one with all the custom ones (from config files or set after).

The resolution order is: custom_config(current_section -> main_section) -> default_config(current_section -> main_section)

add_to_section(*section*, *option*, *value*)

get(*entry*, *default=None*)

get_section(*section*)

getarray(*entry*, *default=None*)

getboolean(*entry*, *default=None*)

getdict(*entry*, *default=None*)

getfloat(*entry*, *default=None*)

getint(*entry*, *default=None*)

load(*path*)

load_plugins()

set(*entry*, *value*)

repoman.common.config.**update_conf_from_plugin**(*config*, *plugins*, *prefix*)

repoman.common.parser module

When specifying a source for an artifact, you have to do it in this format:

```
source_type:value[:filter[:filter[...]]]
```

For each source, it will be expanded, and filtered. An example:

```
repo:master-nightly:name~ovirt-engine.*:latest=2
```

class repoman.common.parser.**Parser**(*config*, *stores*)
Bases: object

parse(*full_source_str*)
Parses the given source sting and returns a list of resolved artifact paths

Parameters **full_source_str** (*Sting*) – Source sting to parse

Return type list of strings

repoman.common.repo module

This module holds the class and methods to manage a repository.

In our case a repository is not just a yum repository but a set of them and another files, in the following structure:

```
repository_dir
├── store1_dir
│   └── ...
└── store2_dir
    └── ...
```

class repoman.common.repo.**Repo**(*path*, *config*)
Bases: object

Represents the repository sctructure, it does not require that the repo has the structure specified in the module doc when loading it, but when adding new rpms or generating the sources it will create the new files in that directory structure.

Configuration options:

- **allowed_repo_paths** Comma separated list of paths where repositories can be found/created

add_path_extra_dir(*dirname*)
Adds an extra subdir to the curret path

Parameters **dirname** (*str*) – Name of the extra dir to add

Returns None

add_path_suffix(*suffix*)
Adds a suffix to the repo's path

Parameters **suffix** (*str*) – Suffix to postpend to the repo's path

Returns None

add_source(*artifact_source*)
Generic function to add an artifact to the repo.

Some base (meta-)sources are supported, like:

- ***conf:path/to/file*: This will include all the sources defined in** the file *path/to/file*, it supports shell comments in the file, and empty lines
- ***stdin*: This will read any sources passd through stdin, with the** same format as conf: files (*cat sources | repoman myrepo add stdin* is the same as *repoman myrepo add conf:sources*)
- ***repo-suffix:suffix_string*: This allows you to define a suffix** string for the destination repo, it's helpful to allow generating custom repos from a base one, when the repoman command is hard-coded (with the combination of stdin source)
- ***repo-extra-dir:dirname*: This allows you to define an extra subdir** string for the destination repo, it's helpful to allow generating custom repos from a base one, when the repoman command is hardcoded (with the combination of stdin source).

Parameters **artifact_source** – source string of the artifact to add

delete_old(**args*, ***kwargs*)
Remove any old versions but the latest *num_to_keep*

Parameters

- **num_to_keep** – Number of versions to keef for each artifact
- **noop** – if True will not actually remove anything

load()
Actually load all the stores and load the contents of the repo

parse_source_stream(*source_stream*)
Given a iterable of sources, add all that apply, skipping comments and empty lines

Parameters **source_stream** – iterable with the sources, can be an open file object as returned by *open*

rebase(*new_path*)
Changes the root path of the repo

Parameters **new_path** (*str*) – New path to root the repo to

Returns None

save(**args*, ***kwargs*)
Realize all the changes made so far

repoman.common.repo.**cleanup**(*temp_dir*)

repoman.common.repo.**loaded**(*func*)

repoman.common.utils module

exception repoman.common.utils.**NotSamePackage**
Bases: exceptions.Exception

Thrown when trying to compare different packages

repoman.common.utils.**cmpfullver**(*fullver1*, *fullver2*)

Compares version strings in the form: x.y.z-a.b.c

repoman.common.utils.**cmpver**(*ver1*, *ver2*)
Compares two version in a natural sort ordering fashion (what you usually expect when comparing versions yourself). Thought for version strings in the form:

x.y.z

repoman.common.utils.**copy**(*what*, *where*)
Try to link, try to copy if cross-device

repoman.common.utils.**create_symlink**(*basepath*, *dest*, *link*)
Creates a symlink

Parameters

- **basepath** (*str*) – Path to creat the simlink on
- **dest** (*str*) – Path not relative to basepath of the destination for the link
- **link** (*str*) – Path relative to basepath of the link itself

Returns None

repoman.common.utils.**download**(*path*, *dest_path*, *tries=3*, *verify=True*)
Download a package from a url.

repoman.common.utils.**extract_sources**(*rpm_path*, *dst_dir*, *with_patches=False*)
Extract the source files fro a srcrpm, uses rpm2cpio

Parameters

- **rpm_path** – Path to the srcrpm
- **dst_dir** – Destination directory to hold the sources, will create it if it does not exist
- **with_patches** – if set to True, extract also the .patch files if any

repoman.common.utils.**find_recursive**(*base_path*, *fmatch*)
Walks a directory recursively and returns the list of files for which fmatch(filename) returns True

repoman.common.utils.**get_gpg**(*homedir='/home/docs/.gnupg'*, *use_agent=False*)

repoman.common.utils.**get_last**(*what*, *num*)

repoman.common.utils.**get_plugins**(*plugin_dir=None*)
Given a path, returns the importable files and directories in it

repoman.common.utils.**gpg_get_keyhex**(*key_path*, *gpg=None*)

repoman.common.utils.**gpg_get_keyuid**(*key_path*, *gpg=None*)

repoman.common.utils.**gpg_load_key**(*key_path*, *gpg=None*)

repoman.common.utils.**gpg_unlock**(*key_path*, *use_agent=True*, *passphrase=None*, *gpg=None*)

repoman.common.utils.**list_files**(*path*, *extension*)
Find all the files with the given extension under the given dir

repoman.common.utils.**print_busy**(*prev_pos=0*)
Shows a spinning bar when called like this: > i=0 > while True: > i = print_busy(i)

repoman.common.utils.**response2str**(*response*)

repoman.common.utils.**rsplit**(*what*, *separator*, *num_results=None*)

repoman.common.utils.**sanitize_file_name**(*file_name*, *replacement='_'*)
Replaces any unwanted characters from the given file or dir name with the given replacement string

Parameters

- **file_name** (*str*) – file or directory name to sanitize
- **replacement** (*str*) – what to put in place of the bad chars

Returns sanitized name with all the bad chars replaced

Return type str

Example

```
>>> sanitize_file_name("I'm an /ugly%#@!")
'I_m an _ugly____'
>>> sanitize_file_name("I'm an /ugly%#@!", replacement='-')
'I-m an -ugly----'
```

repoman.common.utils.**save_file**(*src_path*, *dst_path*)
Save a file to a specific new path if not there already. Will create the path tree if it does not exist already.

Parameters

- **src_path** – Source path for the package
- **dst_path** – New path to save the package to

repoman.common.utils.**sign_detached**(*src_dir*, *key*, *passphrase=None*)
Create the detached signatures for the files in the specified dir.

Parameters

- **src_dir** – File to sign or directory with files to sign (recursively)
- **key** – Key to sign the sources with
- **passphrase** – Passphrase for the given key

repoman.common.utils.**sign_file**(*gpg*, *fname*, *keyid*, *passphrase*, *detach=True*)

repoman.common.utils.**split**(*what*, *separator*, *num_results=None*)

repoman.common.utils.**to_human_size**(*fsize*)
Pass a number from bytes, to human readable form, using 1024 multiples.

repoman.common.utils.**tryint**(*mayint*)
Tries to cast to int, and returns the same object if failed.

2.1.2 Submodules

2.1.3 repoman.cmd module

This program is a helper to repo management

Started as specific for rpm files, but was modified to be able to support different types of artifacts

repoman.cmd.**add_add_artifact_parser**(*parent_parser*)

repoman.cmd.**add_createrepo_parser**(*parent_parser*)

repoman.cmd.**add_docs_parser**(*parent_parser*)

repoman.cmd.**add_generate_src_parser**(*parent_parser*)

repoman.cmd.**add_remove_old_parser**(*parent_parser*)

repoman.cmd.**add_sign_artifacts_parser**(*parent_parser*)

repoman.cmd.**do_add**(*args*, *config*, *repo*)

repoman.cmd.**do_createrepo**(*repo*)

repoman.cmd.**do_generate_src**(*config*, *repo*)

repoman.cmd.**do_remove_old**(*args*, *config*, *repo*)

repoman.cmd.**do_show_docs**(*args*)

repoman.cmd.**do_sign_artifacts**(*repo*)

repoman.cmd.**format_conf_options**(*conf_dict*)

repoman.cmd.**get_config**(*args*)

repoman.cmd.**get_latest_repo**(*args*, *config*, *base_repo*)

repoman.cmd.**get_repo**(*args*, *config*)

repoman.cmd.**handle_custom_options**(*args*, *config*)

repoman.cmd.**has_to_handle_signing_key**(*args*, *config*)

repoman.cmd.**main**()

repoman.cmd.**parse_args**()

repoman.cmd.**set_signing_key**(*config*)

repoman.cmd.**setup_logging**(*verbose=False*)

repoman.cmd.**setup_regular_logging**()

repoman.cmd.**setup_verbose_logging**()

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

B

C

D

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W