packman Documentation

Release 0.1.1

nir0s

June 10, 2014

Contents

packman creates packages.

packman retrieves sources, maybe adds some bootstrap scripts and configuration files to them, and packs them up nice and tight in a single package.

packman's real strength is in providing an simple configuration based API to the most basic tasks in creating packages like:

- retrieving sources from apt, yum, ppa and urls.
- retrieving python modules and ruby gems WITH dependencies.
- generating different files from templates using jinja2.
- packaging using fpm (API NOT IMPLEMENTED YET - only exists in default implementation).
- handling different file operations like creating directories and removing them, taring, untaring, etc..

additionally, you can create your own python based tasks to replace the default ones and call them very simply using pkm (packman's cli).

Contents:

CHAPTER 1

Quick Start

- install Vagrant.
- install VirtualBox.
- clone the github repo:

```
git clone git@github.com:cloudify-cosmo/cloudify-packager.git
```

- go to the vagrant directory
- run:

```
vagrant up packman
vagrant ssh packman
cd ~/examples
sudo su
pkm make
```

- review the retrieved resources in /sources
- review the created deb files in /packages
- start playing around with ~/examples/packages.py

CHAPTER 2

Installation

2.1 Pre-Requirements

`packman` uses the following 3rd party components:

- python-dev -*for pycrypto (ARG!)*
- python-setuptools -*also to install packages on the ‘‘packman‘‘ instance*
- fpm -*main packaging framework*
- fabric -*to run it all*
- pip >1.5 -*to download python modules*
- jinja2 -*to create scripts and configuration files from templates*
- virtualenv (OPTIONAL) -*to create python virtual environments.*
- rubygems (OPTIONAL) -*to download ruby gems*

Note: a script is provided to install the above requirements.

2.2 Installing Packman

You can install `packman` by running `pip install packman`. Of course, you must have the prereqs installed to fully utilize `packman`‘s potential...

Note: The vagrantfile provided in the github repo can supply you with a fully working `packman` machine.

CHAPTER 3

pkm - packman's CLI

3.1 CLI Functionality

`packman`'s provides a cli interface to packman's basic features. you can:

Note: the below commands also apply to `get` (retrieving sources).

- pack all components in a `components file` (pkm pack)
- pack a single component (pkm pack -c COMPONENT_NAME)
- pack a list of components (pkm pack -c COMPONENT1,COMPONENT2,COMPONENT3...)
- pack components from an alternative `components file` (pkm pack -f /my_components_file.py)
- pack components with an exclusion list (pkm pack -x COMPONENTS1,COMPONENT2,...)
- perform all of the above on `get` and `pack` using the same command (pkm make)
- using the basic implementation of the get and pack methods for all components in a components file and specifying a list of components for packman to iterate over to getting and packing a component (or all components) in a single command.

running:

```
pkm -h
```

yeilds the following:

```
Script to run packman via command line

Usage:
    pkm get [--components=<list> --components-file=<path> --exclude=<list> -v]
    pkm pack [--components=<list> --components-file=<path> --exclude=<list> -v]
    pkm make [--components=<list> --components-file=<path> --exclude=<list> -v]
    pkm --version

Arguments:
    pack     Packs component configured in components file
    get      Gets component configured in components file
    make     Gets AND (yeah!) Packs.. don't ya kno!

Options:
    -h --help                   Show this screen.
    -c --components=<list>      Comma Separated list of component names
```

```
-x --exclude=<list>          Comma Separated list of excluded components
-f --components-file=<path> Components file path
-v --verbose                 a LOT of output
--version                    Display current version of sandman and exit
```

Note: when not specifying copmonents explicitly using the –components flag, the task will run on all components in the dict.

CHAPTER 4

Components File Configuration

Configuration of all components is done via a python file containing a single dict with multiple (per component) sub-dicts. We will call it the `components file`. An example components file can get your started...

4.1 A Component's Structure

A component is comprised of a set of key:value pairs. Each component has a set of mandatory parameters like name and version and of optional parameters like source_urls. Obviously, after processing, a component becomes a package...

A very simple example of a component's configuration for riemann:

```
COMPONENTS_PACKAGES_PATH = '/TEST'
PACKAGES_PATH = '/TEST2'
PACKAGES = {
    "riemann": {
        "name": "riemann",
        "version": "0.2.2",
        "source_urls": [
            "http://aphyr.com/riemann/riemann_0.2.2_all.deb",
        ],
        "depends": [
            'openjdk-7-jdk'
        ],
        "package_path": "{0}/riemann/".format(COMPONENT_PACKAGES_PATH),
        "sources_path": "{0}/riemann".format(PACKAGES_PATH),
        "dst_package_type": "deb"
    },
}
```

Breakdown:

- ***name*** is the component's name (DUH!). it's used to create named directories and package file names mostly.
- ***version***, when applicable, is used to apply a version to the component's package name (in the future, it might dictate the component's version to download.)
- ***source_urls*** is a list of package sources to download.
- ***depends*** is a list of dependencies for the package (obviously only applicable to specific package types like debs and rpms.)
- ***package_path*** is the path where the component's package will be stored after the packaging process is complete for that same component.

- ***sources_path*** is the path where the component's parts (files, configs, etc..) will be stored before the component's package is created.
- ***dst_package_type*** is... well.. you know.

4.2 Additional Configuration Parameters

By default, a component can be comprised of a set of parameters, all of which (names) are configurable in the definitions.py file (This is currently only available by editing the module directly). The file is not currently directly available to the user (as most of the parameters names are self-explanatory) but at a future version, a user will be able to override the parameter names by supplying an overriding definitions.py file (to override all or some of the parameter names).

For the complete list of params, see the defintions file.

CHAPTER 5

Template Handling

Component templates:

- packman uses python's jinja2 module to create files from templates.
- template files can be used to generate bootstrap scripts or configuration files by default, but can also be used using external pack/get functions (see component handling) to generate other files if relevant.

Bootstrap script tepmlates:

- Components which should be packaged along with a bootstrap script should have a .template file stationed in package-templates/
- During the packaging process, if a template file exists and its path is passed to the "pack" function (possibly from the config), the bootstrap script will be created and attached to the package (whether by copying it into the package (in case of a tar for instance), or by attaching it (deb, rpm...).)
- The bootstrap script will run automatically upon dpkg-ing when applicable.

Here's an example of a template bootstrap script (for virtualenv, since riemann doesn't require one):

```
PKG_NAME="{{ name }}"
PKG_DIR="{{ sources_path }}"

echo "extracting ${PKG_NAME}..."
sudo tar -C ${PKG_DIR} -xvf ${PKG_DIR}/*.tar.gz
echo "removing tar..."
sudo rm ${PKG_DIR}/*.tar.gz
cd ${PKG_DIR}/virtualenv*
echo "installing ${PKG_NAME}..."
sudo python setup.py install
```

The double curly braces are where the variables are eventually assigned. The name of the variable must match a component's config variable in its dict (e.g name, package_dir, etc...).

Config Templates:

- it is possible to generate configuration file/s from templates or just copy existing configuration files into the package which can later be used by the bootstrap script to deploy the package along with its config.
- **the component's "config_templates" sub-dict can be used for that purpose. 4 types of config template keys exist in the sub-**
  - __template_dir - a directory from which template files are generated (iterated over...)
  - __template_file - an explicit name from which a template file is generated.
  - __config_dir - a directory from which config files are copied.

- __config_file - an explicit name of a config file to be copied.

CHAPTER 6

Using alternative implementations of get or pack methods

packman provides a way to override the basic implementations for the get and pack methods for each component.

let's look at the example:

- we have a components file in our cwd with a `riemann` component.
- we want to run a different `get` method than the default one.
- we create a get.py file in our cwd with a function called `get_riemann`.
- this will override the get method when running `pkm get -c riemann`
- same goes for the `pack` method.
- of course, a user can create a specific get function only to extend the base get method by importing the ***get*** method from packman and adding to it.

for an example, see an example get file.

..note:: when looking for the overriding methods' names, all hyphens will be replaced by underscores and all dots will be removed. so, for instnce, you could provide a component named "java-1.7.0-openjdk", but when specifying the method's name, you should call it "get_java_170_openjdk"

CHAPTER 7

Packman's API

packman provides an API that can be used to easily create packages from external application (for instance, you could call packman from your build machine to generate packages after all tests passed).

The API is also usable when alternative implementations of *get* and *pack* for different components, as described here

Contents:

packman.packman.**init_logger**(*base_level=20*, *verbose_level=10*, *logging_config=None*)
initializes a base logger

you can use this to init a logger in any of your files. this will use config.py's LOGGER param and logging.dictConfig to configure the logger for you.

Parameters

- **base_level** (*int|logging.LEVEL*) – desired base logging level
- **verbose_level** (*int|logging.LEVEL*) – desired verbose logging level
- **logging_dict** (*dict*) – dictConfig based configuration. used to override the default configuration from config.py

Return type *python logger*

packman.packman.**set_global_verbosity_level**(*is_verbose_output=False*)
sets the global verbosity level for console and the lgr logger.

Parameters **is_verbose_output** (*bool*) – should be output be verbose

packman.packman.**get_distro**()
returns the machine's distro

packman.packman.**check_distro**(*supported=('Ubuntu', 'debian', 'centos')*, *verbose=False*)
checks that the machine's distro is supported

Parameters

- **supported** (*tuple*) – tuple of supported distros
- **verbose** (*bool*) – verbosity level

packman.packman.**get_component_config**(*component_name*, *components_dict=None*, *components_file=None*)
returns a component's configuration

if *components_dict* is not supplied, a packages.py file in the cwd will be assumed unless *components_file* is explicitly given. after a *components_dict* is defined, a *component_config* will be returned for the specified component_name.

Parameters

- **component** (*string*) – component name to retrieve config for.
- **components_dict** (*dict*) – dict containing components configuration
- **components_file** (*string*) – components file to search in

Return type *dict* representing component configuration

packman.packman.**packman_runner**(*action='pack'*, *components_file=None*, *components=None*, *excluded=None*, *verbose=False*)

logic for running packman. mainly called from the cli (pkm.py)

if no *components_file* is supplied, we will assume a local packages.py as *components_file*.

if *components* are supplied, they will be iterated over. if *excluded* are supplied, they will be ignored.

if a pack.py or get.py files are present, and an action_component function exists in the files, those functions will be used. else, the base get and pack methods supplied with packman will be used. so for instance, if you have a component named *x*, and you want to write your own *get* function for it. Just write a get_x() function in get.py.

Parameters

- **action** (*string*) – action to perform (get, pack)
- **components_file** (*string*) – path to file containing component config
- **components** (*string*) – comma delimited list of components to perform *action* on.
- **excluded** (*string*) – comma delimited list of components to exclude
- **verbose** (*bool*) – determines output verbosity level

Return type *None*

packman.packman.**get**(*component*)

retrieves resources for packaging

Note: component params are defined in packages.py

Note: param names in packages.py can be overriden by editing definitions.py which also has an explanation on each param.

Parameters

- **package** (*dict*) – dict representing package config as configured in packages.py
- **name** (*string*) – package's name will be appended to the filename and to the package depending on its type
- **version** (*string*) – version to append to package
- **source_url** (*string*) – source url to download
- **source_repo** (*string*) – source repo to add for package retrieval
- **source_ppa** (*string*) – source ppa to add for package retrieval
- **source_key** (*string*) – source key to download
- **key_file** (*string*) – key file path
- **reqs** (*list*) – list of apt requirements
- **dst_path** (*string*) – path where downloaded source are placed

- **package_path** (*string*) – path where final package is placed
- **modules** (*list*) – list of python modules to download
- **gems** (*list*) – list of ruby gems to download
- **overwrite** (*bool*) – indicated whether the sources directory be erased before creating a new package

Return type *None*

packman.packman.**pack**(*component*)

creates a package according to the provided package configuration in packages.py uses fpm (https://github.com/jordansissel/fpm/wiki) to create packages.

Note: component params are defined in packages.py but can be passed directly to the pack function as a dict.

Note: param names in packages.py can be overriden by editing definitions.py which also has an explanation on each param.

Parameters

- **component** (*string|dict*) – string or dict representing component name or params (coorespondingly) as configured in packages.py
- **name** (*string*) – package's name will be appended to the filename and to the package depending on its type
- **version** (*string*) – version to append to package
- **src_pkg_type** (*string*) – package source type (as supported by fpm)
- **dst_pkg_types** (*list*) – package destination types (as supported by fpm)
- **src_path** (*string*) – path containing sources from which package will be created
- **tmp_pkg_path** (*string*) – path where temp package is placed
- **package_path** (*string*) – path where final package is placed
- **bootstrap_script** (*string*) – path to place generated script
- **bootstrap_script_in_pkg** (*string*) –
- **config_templates** (*dict*) – configuration dict for the package's config files
- **overwrite** (*bool*) – indicates whether the destination directory be erased before creating a new package
- **mock** (*bool*) – indicates whether a mock pack will be created (for testing purposes. does not use fpm)

Return type *None*

packman.packman.**do**(*command*, *attempts=2*, *sleep_time=3*, *accepted_err_codes=None*, *capture=False*, *combine_stderr=False*, *sudo=False*)

executes a command locally with retries on failure.

if a *command* execution is successful, it will return a fabric object with the output (x.stdout, x.stderr, x.succeeded, etc..)

else, it will retry an *attempts* number of attempts and if all fails it will return the fabric output object. obviously, *attempts* must be larger than 0...

Parameters

- **command** (*string*) – shell command to be executed
- **attempts** (*int*) – number of attempts to perform on failure
- **sleep_time** (*int*) – sleeptime between attempts
- **capture** (*bool*) – should the output be captured for parsing?
- **combine_stderr** (*bool*) – combine stdout and stderr (NOT YET IMPL)
- **sudo** (*bool*) – run as sudo

Return type *responseObject* (for fabric operation)

class `packman.packman.`**`CommonHandler`**

common class to handle files and directories

`find_in_dir`(*dir*, *pattern*, *sudo=True*)

finds file/s in a dir

Parameters

- **dir** (*string*) – directory to look in
- **patten** (*string*) – what to look for

Return type `stdout` *string* if found, else *None*

`is_dir`(*dir*)

checks if a directory exists

Parameters **dir** (*string*) – directory to check

Return type *bool*

`is_file`(*file*)

checks if a file exists

Parameters **file** (*string*) – file to check

Return type *bool*

`touch`(*file*, *sudo=True*)

creates a file

Parameters **file** (*string*) – file to touch

`mkdir`(*dir*, *sudo=True*)

creates (recursively) a directory

Parameters **dir** (*string*) – directory to create

`rmdir`(*dir*, *sudo=True*)

deletes a directory

Parameters **dir** (*string*) – directory to remove

`rm`(*file*, *sudo=True*)

deletes a file or a set of files

Parameters **file(s)** (*string*) – file(s) to remove

`cp`(*src*, *dst*, *recurse=True*, *sudo=True*)

copies (recuresively or not) files or directories

Parameters

- **src** (*string*) – source to copy
- **dst** (*string*) – destination to copy to
- **recurse** (*bool*) – should the copying process be recursive?

mv(*src*, *dst*, *sudo=True*)
moves files or directories

Parameters

- **src** (*string*) – source to copy
- **dst** (*string*) – destination to copy to

tar(*chdir*, *output_file*, *input_path*, *opts='zvf'*, *sudo=True*)
tars an input file or directory

Parameters

- **chdir** (*string*) – change to this dir before archiving
- **output_file** (*string*) – tar output file path
- **input** (*string*) – input path to create tar from
- **opts** (*string*) – tar opts

untar(*chdir*, *input_file*, *opts='zvf'*, *strip=0*, *sudo=True*)
untars a file

Parameters

- **chdir** (*string*) – change to this dir before extracting
- **input_file** (*string*) – file to untar
- **opts** (*string*) – tar opts

class packman.packman.**fpmHandler**(*name*, *input_type*, *output_type*, *source*, *sudo*)
Bases: packman.packman.CommonHandler

packaging handler

fpm(***kwargs*)

class packman.packman.**PythonHandler**
Bases: packman.packman.CommonHandler

python operations handler

pips(*modules*, *venv=False*, *attempts=5*)
pip installs a list of modules

Parameters

- **modules** (*list*) – python modules to `pip install`
- **venv** (*string*) – (optional) if ommited, will use system python else, will use *venv* (for virtualenvs and such)

pip(*module*, *venv=False*, *attempts=5*, *sudo=True*, *timeout='45'*)
pip installs a module

Parameters

- **module** (*string*) – python module to `pip install`

- **venv** (*string*) – (optional) if ommited, will use system python else, will use *venv* (for virtualenvs and such)

get_python_modules(*modules*, *dir=False*, *venv=False*)
downloads python modules

Parameters

- **modules** (*list*) – python modules to download
- **dir** (*string*) – dir to download modules to
- **venv** (*string*) – (optional) if ommited, will use system python else, will use *dir* (for virtualenvs and such)

get_python_module(*module*, *dir=False*, *venv=False*)
downloads a python module

Parameters

- **module** (*string*) – python module to download
- **dir** (*string*) – dir to download module to
- **venv** (*string*) – (optional) if ommited, will use system python else, will use *dir* (for virtualenvs and such)

check_module_installed(*name*, *dir=False*)
checks to see that a module is installed

Parameters

- **name** (*string*) – module to check for
- **dir** (*string*) – (optional) if ommited, will use system python else, will use *dir* (for virtualenvs and such)

venv(*venv_dir*, *sudo=True*)
creates a virtualenv

Parameters **venv_dir** (*string*) – venv path to create

class packman.packman.**RubyHandler**
Bases: packman.packman.CommonHandler

get_ruby_gems(*gems*, *dir=False*)
downloads a list of ruby gems

Parameters

- **gems** (*list*) – gems to download
- **dir** (*string*) – directory to download gems to

get_ruby_gem(*gem*, *rbenv=False*, *dir=False*)
downloads a ruby gem

Parameters

- **gem** (*string*) – gem to download
- **dir** (*string*) – directory to download gem to

class packman.packman.**YumHandler**
Bases: packman.packman.CommonHandler

static update()
runs yum update

check_if_package_is_installed(*package*)
checks if a package is installed

Parameters **package** (*string*) – package name to check

Return type *bool* representing whether package is installed or not

downloads(*reqs*, *sources_path*)
downloads component requirements

Parameters

- **reqs** (*list*) – list of requirements to download
- **sources_path** – path to download requirements to

download(*package*, *dir*, *enable_repo=False*)
uses yum to download package debs from ubuntu's repo

Parameters

- **package** (*string*) – package to download
- **dir** (*string*) – dir to download to

installs(*packages*)
yum installs a list of packages

Parameters **package** (*list*) – packages to install

install(*package*)
yum installs a package

Parameters **package** (*string*) – package to install

add_src_repos(*source_repos*)
adds a list of source repos to the apt repo

Parameters **source_repos** (*list*) – repos to add to sources list

add_src_repo(*source_repo*)
adds a source repo to the apt repo

Parameters **source_repo** (*string*) – repo to add to sources list

add_keys(*key_files*)
adds a list of keys to the local repo

Parameters **key_files** (*string*) – key files paths

add_key(*key_file*)
adds a key to the local repo

Parameters **key_file** (*string*) – key file path

class packman.packman.**AptHandler**
Bases: packman.packman.CommonHandler

dpkg_name(*dir*)
renames deb files to conventional names

Parameters **dir** (*string*) – dir to review

check_if_package_is_installed(*package*)
checks if a package is installed

Parameters **package** (*string*) – package name to check

Return type *bool* representing whether package is installed or not

downloads(*reqs*, *sources_path*)
downloads component requirements

Parameters

- **reqs** (*list*) – list of requirements to download
- **sources_path** – path to download requirements to

download(*package*, *dir*)
uses apt to download package debs from ubuntu's repo

Parameters

- **package** (*string*) – package to download
- **dir** (*string*) – dir to download to

autoremove(*pkg*)
autoremoves package dependencies

Parameters **pkg** (*string*) – package to remove

add_src_repos(*source_repos*)
adds a list of source repos to the apt repo

Parameters **source_repos** (*list*) – repos to add to sources list

add_src_repo(*source_repo*)
adds a source repo to the apt repo

Parameters **source_repo** (*string*) – repo to add to sources list

add_ppa_repos(*source_ppas*)
adds a list of ppa repos to the apt repo

Parameters **source_ppas** (*list*) – ppa urls to add

add_ppa_repo(*source_ppa*)
adds a ppa repo to the apt repo

Parameters **source_ppa** (*string*) – ppa url to add

add_keys(*key_files*)
adds a list of keys to the local repo

Parameters **key_files** (*string*) – key files paths

add_key(*key_file*)
adds a key to the local repo

Parameters **key_file** (*string*) – key file path

static update()
runs apt-get update

installs(*packages*)
apt-get installs a list of packages

Parameters **packages** (*list*) – packages to install

install (*package*)
apt-get installs a package

Parameters **package** (*string*) – package to install

purges (*packages*)
completely purges a list of packages from the local repo

Parameters **packages** (*list*) – packages name to purge

purge (*package*)
completely purges a package from the local repo

Parameters **package** (*string*) – package name to purge

class packman.packman.**WgetHandler**
Bases: packman.packman.CommonHandler

downloads (*urls*, *dir=False*, *sudo=True*)
wgets a list of urls to a destination directory

Parameters

- **urls** (*list*) – a list of urls to download
- **dir** (*string*) – download to dir...

download (*url*, *dir=False*, *file=False*, *sudo=True*)
wgets a url to a destination directory or file

Parameters

- **url** (*string*) – url to wget?
- **dir** (*string*) – download to dir....
- **file** (*string*) – download to file...

class packman.packman.**TemplateHandler**
Bases: packman.packman.CommonHandler

generate_configs (*component*, *sudo=True*)
generates configuration files from templates

for every key in the configuration templates sub-dict, if a key corresponds with a templates/configs key (as defined in definitions.py) the relevant method for creating configuration files will be applied.

Parameters **component** (*dict*) – contains the params to use in the template

generate_from_template (*component_config*, *output_file*, *template_file*, *templates='package-templates/'*)
generates configuration files from templates using jinja2 http://jinja.pocoo.org/docs/

Parameters

- **component_config** (*dict*) – contains the params to use in the template
- **output_file** (*string*) – output file path
- **template_file** (*string*) – template file name
- **templates** (*string*) – template files directory

CHAPTER 8

Packman File Structure

8.1 Module

- packman.py contains the base functions and classes for handling component actions (pack, get, wget, mkdir, apt-download, etc..).
- packman_config.py contains the packman logger configuration.
- event_handler.py providers an interface to rabbitmq (EXPERIMENTAL and currently in development)
- definitons.py contains the base parameter definitions for the components file.
- packages.py in the current working directory contains a PACKAGES dict param with the component's configuration.

8.2 User

- components file other than packages.py (optional) can be stationed anywhere as long as they're addressed thru the cli.
- get.py in the current working directory (optional) contains the logic for downloading and arranging a component's contents.
- pack.py in the current working directory (optional) contains the logic for packaging a component.
- if bootstrap scripts exist, a "package-templates" directory must exist in the current working directory (will be changed in the future...)
- of course, any other directories and files can co-exist in the current working directory. for instance, a package-configuration directory can be created and then referenced in the components file to hold package configuration file templates.

CHAPTER 9

Indices and tables

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

p