packman Documentation

Release 0.5.0

nir0s

Sep 27, 2017

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/packman.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.yaml

CHAPTER 2

Installation

Pre-Requirements

`packman` uses the following 3rd party components:

- ruby - *required for fpm*
- fpm -*main packaging framework*
- pip >1.5 -*to download python modules (and install packman)*
- virtualenv (OPTIONAL) -*to create python virtual environments.*
- rubygems (OPTIONAL) -*to download ruby gems*
- rpmbuild (OPTIONAL) -*to create rpms*
- tar (OPTIONAL) -*to create tars*
- gzip (OPTIONAL) -*to create tar.gz's*

Note: the rest of the requirements are python modules which will be installed with *packman*

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

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

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 packages in a packages file (pkm pack)
- pack a single package (pkm pack -c package_NAME)
- pack a list of packages (pkm pack -c package1,package2,package3...)
- pack packages from an alternative packages file (pkm pack -f /my_packages_file.yaml)
- pack packages with an exclusion list (pkm pack -x packageS1,package2,...)
- 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 packages in a packages file and specifying a list of packages for packman to iterate over to getting and packing a package (or all packages) in a single command.

running:

```
pkm -h
```

yeilds the following:

```
Script to run packman via command line

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

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

Options:
    -h --help                   Show this screen.
    -c --packages=<list>      Comma Separated list of package names
    -x --exclude=<list>         Comma Separated list of excluded packages
    -f --packages-file=<path> packages file path
    -v --verbose                a LOT of output
    --version                   Display current version of sandman and exit
```

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

CHAPTER 4

Packages File Configuration

Configuration of all packages is done via a YAML file containing a single dict with multiple (per package) sub-dicts. We will call it the `packages file`. An example packages file can get you started...

A package's Structure

A package is comprised of a set of key:value pairs. Each package has a set of mandatory parameters like name and version and of optional parameters like source_urls.

A very simple example of a package's configuration:

```
packages:
mock_package:
    name: test_package
    version: 3.1
    sources_path: sources
    depends:
        - make
        - g++
    prereqs:
        - curl
    source_ppas:
        - ppa:chris-lea/node.js
    source_repos:
        - deb http://nginx.org/packages/mainline/ubuntu/ precise nginx
        - deb-src http://nginx.org/packages/mainline/ubuntu/ precise nginx
    source_keys:
        - http://nginx.org/keys/nginx_signing.key
    source_urls:
        - https://github.com/jaraco/path.py/archive/master.zip
    requires:
        - make
    virtualenv:
        path: venv
        modules:
            - pyyaml
    python_modules:
        - nonexistentmodule
        - cloudify
        - pyyaml
    ruby_gems:
        - gosu
    package_path: tests
    source_package_type: dir
    destination_package_types:
        - tar.gz
        - deb
        - rpm
    keep_sources: true
    bootstrap_script: packman/tests/templates/mock_template.j2
    bootstrap_template:
    test_template_parameter: test_template_output
    config_templates:
        template_file:
            template: packman/tests/templates/mock_template.j2
            output_file: mock_template.output
            config_dir: config
        template_dir:
            templates: packman/tests/templates
            config_dir: config
        config_dir:
            files: packman/tests/templates
            config_dir: config
        params: param
```

Breakdown:

- ***name*** is the package'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 package's package name (in the future, it might dictate the package's version to download.)
- ***sources_path*** is the path where the package's parts (files, configs, etc..) will be stored before the package's package is created.
- ***depends*** is a list of dependencies for the package (obviously only applicable to specific package types like debs and rpms.)
- ***prereqs*** is a list of distribution specific requirements to install before attemping to retrieve the package's resources.
- ***source_ppas*** is a list of ppa repos to add.
- ***source_repos*** is a list of repositories to add to the local repos file (distro specific).
- ***source_keys*** is a list of keys to add.
- ***source_urls*** is a list of package sources to download.
- ***requires*** is a list of distro specific requirements to download (from apt, yum, etc..)
- ***package_path*** is the path where the package's package will be stored after the packaging process is complete for that same package.

... meh. - ***destination_package_types*** is... well.. you know.

Additional Configuration Parameters

By default, a package 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.**get_package_config**(*package_name*, *packages_dict=None*, *packages_file=None*)

returns a package's configuration

if *packages_dict* is not supplied, a packages.yaml file in the cwd will be assumed unless *packages_file* is explicitly given. after a *packages_dict* is defined, a *package_config* will be returned for the specified package_name.

Parameters

- **package** (*string*) – package name to retrieve config for.
- **packages_dict** (*dict*) – dict containing packages configuration
- **packages_file** (*string*) – packages file to search in

Return type *dict* representing package configuration

packman.packman.**packman_runner**(*action*, *packages_file=None*, *packages=None*, *excluded=None*, *verbose=False*)

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

if no *packages_file* is supplied, we will assume a local packages.yaml as *packages_file*.

if *packages* 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_package 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 package 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)
- **packages_file** (*string*) – path to file containing package config

- **packages** (*string*) – comma delimited list of packages to perform *action* on.
- **excluded** (*string*) – comma delimited list of packages to exclude
- **verbose** (*bool*) – determines output verbosity level

Return type *None*

`packman.packman.`**`get`**(*package*)

retrieves resources for packaging

Note: package params are defined in packages.yaml

Note: param names in packages.yaml 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.yaml will be appended to the filename and to the package depending on its type

Return type *None*

`packman.packman.`**`pack`**(*package*)

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

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

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

Parameters **package** (*string|dict*) – string or dict representing package name or params (coorespondingly) as configured in packages.yaml

Return type *None*

class `packman.packman.`**`Validate`**(*package*)

`validate_package_properties`()

`destination_package_types`(*package_types*)

CHAPTER 8

Packman File Structure

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.

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

Python Module Index

Index