distgen Documentation

Release 0.16

Pavel Raiskup, Slavek Kabrda

Sep 18, 2017

Contents:

1 Introduction to distgen **1**
- 1.1 Download/Installation . . . 1
- 1.2 Simple Example with Dockerfile . . . 1

2 Configs in distgen **5**
- 2.1 Builtin Configs . . . 5
- 2.2 Using Config Values in Templates . . . 6
- 2.3 Creating Your Own Config . . . 6
- 2.4 Dynamic Values in Templates . . . 6

3 Specs and Multispecs **9**
- 3.1 Specs . . . 9
- 3.2 Multispecs . . . 9
- 3.3 Combining Specs and Multispecs . . . 12

4 Builtins **13**
- 4.1 Commands . . . 13

5 Macros **15**

6 Recursive Rendering **17**
- 6.1 Example . . . 17
- 6.2 Usage . . . 18

7 Indices and tables **19**

CHAPTER 1

Introduction to distgen

Distgen is a distribution-oriented templating system.

The problem this project tries to mitigate is "portable" scripting for variety of operating systems (currently Linux distributions only) in the wild. While writing a "universal" script, one needs to take into account small or bigger differences among operating systems (like package installation tools, versions of utilities, expected directories for binaries, libraries, etc.).

The *distgen* project is thus something like database of OS differences together with convenience tool-set that allows you to instantiate valid script for particular distribution. The concept is to have *template* file (mostly raw jinja2 template) together with "declarative" *spec* file (YAML file) that fulfils the needs of particular template.

Download/Installation

Stable releases of distgen are available (for Red Hat distributions) in Copr.

You can also run development version directly from Github, simply use the `dg` shell wrapper available in git root directory. In order to use distgen from git checkout, you'll need to install dependencies manually. You can do that e.g. using pip: `pip install --user -r requirements.txt`.

Simple Example with Dockerfile

Typical example is the need to instantiate working Dockerfile for list of supported **Fedora/CentOS** versions.

1. Create `common.yaml` spec:

```
# This file provides basic values that are the same for all various rendering␣
↪combinations
name: myawesomeimage
description: "This is a simple container that just tells you how awesome it is.␣
↪That's it."
```

2. Create `multispec.yaml` spec:

```
# This file specifies rendering "matrix" - the different combinations of values
# that the templates can be rendered for
version: 1

# "specs" contains named "spec groups"
specs:
  # "distroinfo" is a mandatory "spec group"
  # - each of its members must contain "distros" list
  # - it can also contain any extra values
  distroinfo:
    fedora:
      distros:
        - fedora-26-x86_64
        - fedora-25-x86_64
      vendor: "Fedora Project"
      authoritative_source_url: "some.url.fedoraproject.org"
      distro_specific_help: "Some Fedora specific help"
    centos:
      distros:
        - centos-7-x86_64
      vendor: "CentOS"
      authoritative_source_url: "some.url.centos.org"
      distro_specific_help: "Some CentOS specific help"
  # apart from "distroinfo", you can specify as many arbitrary spec groups as you␣
↪want
  # - any of the members of these spec groups can contain arbitrary values
  version:
    "2.2":
      version: "2.2"
    "2.4":
      version: "2.4"
```

3. Create a `Dockerfile.template` template

```
# "config.*" values usually come from distribution configs shipped with␣
↪distgen;
# the config is specified by "--distro" argument to "dg" on command line
FROM {{ config.docker.from }}

LABEL MAINTAINER ...

ENV NAME=mycontainer VERSION=0 RELEASE=1 ARCH=x86_64

# "spec.*" values are result of merging any specs passed by "--spec" to "dg"
# and values selected from multispec file (if used) - see below
LABEL summary="A container that tells you how awesome it is." \
      com.redhat.component="$NAME" \
      version="$VERSION" \
      release="$RELEASE.$DISTTAG" \
      architecture="$ARCH" \
      usage="docker run -p 9000:9000 mycontainer" \
      help="Runs mycontainer, which listens on port 9000 and tells you how␣
↪awesome it is. No dependencies." \
      description="{{ spec.description }}" \
      vendor="{{ spec.vendor }}" \
      org.fedoraproject.component="postfix" \
      authoritative-source-url="{{ spec.authoritative_source_url }}" \
```

```
      io.k8s.description="{{ spec.description }}" \
      io.k8s.display-name="Awesome container with SW version {{ spec.
↪software_version }}" \
      io.openshift.expose-services="9000:http" \
      io.openshift.tags="some,tags"

EXPOSE 9000

# We don't actually use the "software_version" here, but we could,
#  e.g. to install a module with that ncat version
RUN {{ commands.pkginstaller.install(['nmap-ncat']) }} && \
    {{ commands.pkginstaller.cleancache() }}

RUN echo '#!/bin/bash' > /usr/bin/script.sh && \
    echo "exec nc -kl 9000 -c 'echo -e \"HTTP/1.1 200 OK\n\";echo \"I am␣
↪awesome\"'" >> /usr/bin/script.sh && \
    chmod +x /usr/bin/script.sh

CMD ["/usr/bin/script.sh"]
```

4. Run the `dg` tool to generate a **Fedora 26** Dockerfile with software version **2.4**:

```
# when using "--multispec", "--multispec-selector" must be used for all
# spec groups except "distroinfo"
$ dg --template Dockerfile.template \
     --spec common.yaml \
     --multispec multispec.yaml \
     --multispec-selector version=2.4 \
     --distro fedora-26-x86_64.yaml \
> Dockerfile
```

5. Run the `dg` tool again to generate a **CentOS 7** dockerfile with software version **2.2**:

```
$ dg --template Dockerfile.template \
     --spec common.yaml \
     --multispec multispec.yaml \
     --multispec-selector version=2.2 \
     --distro centos-7-x86_64.yaml \
> Dockerfile
```

There are more nuances and features of distgen that you can utilize, all of them are documented in the following sections of this documentation.

CHAPTER 2

Configs in distgen

distgen provides lots of useful predefined values that you can use in your templates. These are called *configs* or *distros*. When executing distgen from commandline, you can use `--distro <file>` to select desired config. You can either select a config that's shipped with distgen or you can create and pass your own config.

You can browse through configs shipped with your distgen version in the `/usr/share/distgen/distconf` directory.

Builtin Configs

Following is a list of values that configs shipped with distgen provide. Each item in the list contains a value example for `centos-7-x86_64` config.

- `config.os.arch` (e.g. `x86_64`) - Architecture of the selected distro
- `config.os.id` (e.g. `centos`) - Id of this distro inside distgen
- `config.os.name` (e.g. `CentOS Linux`) - A verbose name of the distro
- `config.os.version` (e.g. `7`) - Version of the distro
- `config.docker.from` (e.g. `centos:7`) - Name (and possibly a tag) of the base image with this distro
- `config.docker.registry` (e.g. `index.docker.io`) - Name of the registry where image specified by `docker.from` can be obtained
- `macros` - Macros provide paths to some useful directories of given distro; for more information on "why and how", see *macros documentation section*. Complete list of macros follows:
  - `macros.bindir` (e.g. `/usr/bin`)
  - `macros.datadir` (e.g. `/usr/share`)
  - `macros.docdir` (e.g. `/usr/share/doc`)
  - `macros.libdir` (e.g. `/usr/lib64`)
  - `macros.libexecdir` (e.g. `/usr/libexec`)

  - `macros.pkgdatadir` - this will expand to `/usr/share/$name`, if `name` is defined in the config - this is not true for default configs
  - `macros.pkgdocdir` - this will expand to `/usr/share/doc/$name`, if `name` is defined in the config - this is not true for default configs
  - `macros.prefix` (e.g. `/usr`)
  - `macros.sbindir` (e.g. `/usr/sbin`)
  - `macros.sysconfdir` (e.g. `/etc`)
  - `macros.unitdir` (e.g. `/usr/lib/systemd/system`)
  - `macros.userunitdir` (e.g. `/usr/lib/systemd/user`)
- `config.package_installer.name` (e.g. `yum`) - name of the command that invokes distro package installer

Using Config Values in Templates

Usage of config values in templates is simple. Here's a very simple example:

```
FROM {{ config.docker.from }}

COPY script.sh {{ macros.bindir }}
```

Creating Your Own Config

When creating your own config, you don't need to specify any of these values, a config can contain any values you want. In that case however, your template must only use the values that your config has.

Dynamic Values in Templates

It may happen to you, that you need a value available in template which is not static – not known before. This could be time, values to generate a help file or others.

The way to this in distgen is to create a new file: `project.py` in root of your project dir. The python code present in the file will be executed by distgen.

```
import subprocess
from distgen.project import AbstractProject

class Project(AbstractProject):
    """ This class has to be named "Project" """

    def inst_init(self, specfiles, template, sysconfig):
        """
        Executed before the project.py/spec files/template is loaded and
        before all the dynamic stuff and specification is calculated.
        Now is still time to dynamically change the list of specfiles or
        adjust the system configuration. You can define a variable as an
        attribute of this project:
```

```
          self.variable = "42"

        which can be later utilized in a template like this:

          {{ project.variable }}
        """
        self.current_date = subprocess \
            .check_output(["date"]) \
            .decode("utf-8")

    def inst_finish(self, specfiles, template, sysconfig, spec):
        """
        Executed after the project.py/spec files/template is loaded, and
        the specification (spec) calculated (== instantiated).  This is
        the last chance to dynamically change sysconfig or spec.
        """
        # you can easily add or change values here based on sourced
        # spec, template, config...
        if spec["..."]:
            sysconfig["..."] = "..."
```

And then in your template, you can use the `current_date` values like this:

```
LABEL build_time="{{ project.current_date }}"
```

CHAPTER 3

Specs and Multispecs

There are two ways in which the distgen command obtains values for template rendering: *config* and specs. These two differ in their purpose. While config values should provide template-agnostic values (e.g. facts about a Linux distro), specs provide template-specific values (e.g. information about version of software being built into a Docker image).

Spec values can be provided either through spec files or through multispec files.

Specs

Specs are simple key-value files that you can use in your templates. You pass them to the `dg` command via `--spec <file>` (can be specified multiple times).

Example spec:

```
version: 2.4
```

Example template:

```
This is documentation for version {{ spec.version }} of some software.
```

By using specs with different `version` in the example above, you could render the template for various software versions. While this is ok for simpler usecases, it might become impractical on bigger scale: imagine you want to render a Dockerfile for an image, that will be based on several different distributions and contain a combination of several versions of 2 different packages. This would mean you'd need lots of small spec files, each with couple of lines and you'd need to manually select and pass them to the `dg` command. This is why the *multispec* mechanism was added to distgen.

Multispecs

Multispec is a file that solves two problems:

- Merges several different smaller spec files into a single file for better readability and convenience.

- Puts smaller specs in logical groups and defines a "matrix" - a list of all combinations of distro config and other features to render templates for.

Here's an example multispec file:

```
# This file specifies rendering "matrix" - the different combinations of values
# that the templates can be rendered for
version: 1

# "specs" contains named "spec groups"
specs:
  # "distroinfo" is a mandatory "spec group"
  # - each of its members must contain "distros" list
  # - it can also contain any extra values
  distroinfo:
    fedora:
      distros:
        - fedora-26-x86_64
        - fedora-25-x86_64
      vendor: "Fedora Project"
      authoritative_source_url: "some.url.fedoraproject.org"
      distro_specific_help: "Some Fedora specific help"
    centos:
      distros:
        - centos-7-x86_64
      vendor: "CentOS"
      authoritative_source_url: "some.url.centos.org"
      distro_specific_help: "Some CentOS specific help"
  # apart from "distroinfo", you can specify as many arbitrary spec groups as you want
  # - any of the members of these spec groups can contain arbitrary values
  version:
    "2.2":
      version: "2.2"
    "2.4":
      version: "2.4"

# in the "matrix" section, you can specify combinations that you want
# to explicitly exclude from the rendering matrix
matrix:
  exclude:
    - distros:
        - fedora-26-x86_64
      version: 2.2
```

A multispec has 3 attributes (see below for the explanation of mechanics behind this file):

- `version` (mandatory) - The version of the multispec file, currently there's only version `1`.
- `specs` (mandatory) - contains list of *groups* (`distroinfo` and `version` in the example above). Each *group* contains named specs - these are exactly like the specs that you would otherwise write into separate files and pass to distgen via `--spec`.
  - The `distroinfo` *group* is mandatory and each of its members *must* contain the `distros` list. These are names of the distro configs shipped with distgen.
  - The `specs` *groups* implicitly define a rendering matrix, which is the cartesian product of all *groups* except `distroinfo`. The `distroinfo` *group* is an exception, as its members `distros` list are used in the cartesian product.
- `matrix` (optional) - currently, this attribute can only contain the `exclude` member. When used, the `exclude`

attribute contains a list of combinations excluded from the matrix. The `distroinfo` members must be referred to via `distro` list.

Hence the above example produces a following rendering matrix:

- `distroinfo: fedora` (for `fedora-25-x86_64` distro), `version: "2.2"`
- `distroinfo: fedora` (for `fedora-25-x86_64` distro), `version: "2.4"`
- `distroinfo: fedora` (for `fedora-26-x86_64` distro), `version: "2.4"`
- `distroinfo: fedora` (for `centos-7-x86_64` distro), `version: "2.2"`
- `distroinfo: fedora` (for `centos-7-x86_64` distro), `version: "2.4"`

Note that `version: "2.2"` is excluded for `fedora-26-x86_64`.

Using Multispecs

Let's consider the example above. We could use it like this:

```
$ dg --template docker.tpl \
     --spec common.yaml \
     --multispec multispec.yaml \
     --multispec-selector version=2.4 \
     --distro fedora-26-x86_64.yaml \
> Dockerfile
```

On calling this command, distgen will:

- Take values from `common.yaml` for base of the result values used for rendering the template.
- It will then add values from `multispec.yaml`:
  - The `--distro fedora-26-x86_64` argument will automatically select the `distroinfo.fedora` section of multispec and add it to result values.
  - The `--multispec-selector version=2.4` will make the `version."2.4"` section of multispec added to the result values.
- Render the template providing the result of operations above accessible under `spec.*` values.

Notes on Multispec Usage

- There can be as many *groups* as you want, not just `distroinfo` and `version`. This also means that you need to use `--multispec-selector` multiple times on commandline.
- The `--multispec-selector` must be used for all groups except `distroinfo`. A proper section to be used from `distroinfo` is implicitly specified by passing the `--distro` argument.
- Only a combination of specs belonging to *groups* can be used when using multispec. In the example above, you can't use fedora-22_i686, since it's not listed in any `distroinfo` section.
- Combinations explicitly listed in `matrix.exclude` cannot be used.
- You can use `dg --multispec <path> --multispec-combinations` to print out all available combinations of distros and selectors based on the given multispec file.

Combining Specs and Multispecs

As shown in the example above, it is perfectly possible to combine specs and multispec. In this case, the specs will be used as a base and values from multispec will be added on top of that (overwriting values if their names conflict).

CHAPTER 4

Builtins

distgen provides some builtins that might come convenient to you while writing templates. Following is an overview and usage instructions of these builtins.

Commands

If you use one of the builtin *configs* of distgen or your config contains `package_installer.name` that is known (currently either `yum` or `dnf`), the `commands.pkginstaller` will be available. Here's a list of valuable attributes and functions that become available with `commands.pkginstaller`:

- `commands.pkginstaller.binary` - the name of the binary of the installer
- `commands.pkginstaller.install(['foo', 'bar'])` - install `foo` and `bar` packages
- `commands.pkginstaller.reinstall(['foo', 'bar'])` - reinstall `foo` and `bar` packages
- `commands.pkginstaller.remove(['foo', 'bar'])` - remove `foo` and `bar` packages
- `commands.pkginstaller.update(['foo', 'bar'])` - update `foo` and `bar` packages
- `commands.pkginstaller.update_all()` - update all installed packages
- `commands.pkginstaller.cleancache()` - clean installer cache

CHAPTER 5

Macros

Macros are distgen's way to provide values that need to be expanded and reexpanded. In other words, the yaml configuration files have no means of value interpolation. This is why the macros system exists.

Macros can be created using any macros from the passed *config*. They can also be passed from commandline via `--macro "name value"` or loaded from a custom project file (TODO: custom project file needs to be documented).

Several macro examples:

- `foo $bindir/executable` - define macro `foo` to expand using `bindir` macro provided in passed config (usually this will expand to `/usr/bin/executable`)
- `bar $foo.sh` - define macro `bar` to expand using previously defined `foo` macro (hence getting `/usr/bin/executable.sh`)

The examples above will be available as `macros.foo` and `macros.bar` in the template.

CHAPTER 6

Recursive Rendering

Note: This is an experimental feature introduced in version 0.16. It is subject to change or removal in future versions. Use at your own risk.

Recursive rendering is a concept similar to *macros*, perhaps aimed at obsoleting it altogether one day. For now, both of these live side by side.

The idea behind recursive rendering is simple:

- The template is rendered recursively until it stops changing.
- Spec values can contain references to other spec or config values.

Note two things:

- Maximum number of rendering passes is always limited to a reasonably small integer (currently passed via `--max-passes X` on command line) to make sure the re-rendering loop ends.
- Recursive rendering is currently turned off, as `--max-passes` is set to `1`.

Example

Let's consider the following spec:

```
name: "myname"
help: "This is a help for {{ config.os.id }}/{{ spec.myname }} image."
```

Let's try rendering a very simple template that looks like this:

```
{{ spec.help }}
```

- In the first rendering pass, `{{ spec.help }}` will get substitued.
- In the second rendering pass, `{{ config.os.id }}` and `{{ spec.myname }}` will get substitued.
- The third rendering pass will find out that there are no changes and will end recursive rendering (so 3 passes are necessary).

Usage

As of now, recursive rendering is turned off by default (in other words, there's just one rendering pass). To turn it on, pass a number higher than 2 to `dg`'s `--max-passes` commandline switch. For example:

```
dg --distro fedora-26-x86_64.yaml --spec myspec.yaml --template Dockerfile --max-
↪passes 10
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search