diskimage-builder Documentation

Release 2.9.1.dev12

OpenStack

Oct 29, 2017

Contents

`diskimage-builder` is a tool for automatically building customized operating-system images for use in clouds and other environments.

It includes support for building images based on many major distributions and can produce cloud-images in all common formats (`qcow2`, `vhd`, `raw`, etc), bare metal file-system images and ram-disk images. These images are composed from the many included `elements`; `diskimage-builder` acts as a framework to easily add your own elements for even further customization.

`diskimage-builder` is used extensively by the TripleO project and within OpenStack Infrastructure.

CHAPTER 1

Code

Release notes for the latest and previous versions are available at:

- http://docs.openstack.org/releasenotes/diskimage-builder/

The code is available at:

- https://git.openstack.org/cgit/openstack/diskimage-builder/

CHAPTER 2

Issues

Issues are tracked on launchpad at:

- https://bugs.launchpad.net/diskimage-builder/+bugs

CHAPTER 3

Communication

Communication among the diskimage-builder developers happens on IRC in `#openstack-dib` on freenode and on the `openstack-dev` mailing list (`openstack-dev@lists.openstack.org`).

CHAPTER 4

Table of Contents

4.1 User Guide

4.1.1 Supported Distributions

Distributions which are supported as a build host:

- Centos 6, 7
- Debian 8 (“jessie”)
- Fedora 20, 21, 22
- RHEL 6, 7
- Ubuntu 14.04 (“trusty”)
- Gentoo
- openSUSE Leap 42.2, 42.3 and Tumbleweed

Distributions which are supported as a target for an image:

- Centos 6, 7
- Debian 8 (“jessie”)
- Fedora 20, 21, 22
- RHEL 6, 7
- Ubuntu 12.04 (“precise”), 14.04 (“trusty”)
- Gentoo
- openSUSE Leap 42.2, 42.3 and Tumbleweed (opensuse-minimal only)

4.1.2 Installation

If your distribution does not proivde packages, you should install `diskimage-builder` via `pip`, mostly likely in a `virtualenv` to keep it separate.

For example, to create a `virtualenv` and install from `pip`

```
virtualenv ~/dib-virtualenv
. ~/dib-virtualenv/bin/activate
pip install diskimage-builder
```

Once installed, you will be able to *build images* using `disk-image-create` and the elements included in the main `diskimage-builder` repository.

Requirements

Most image formats require the `qemu-img` tool which is provided by the `qemu-utils` package on Ubuntu/Debian or the `qemu` package on Fedora/RHEL/opensuse/Gentoo.

When generating images with partitions, the `kpartx` tool is needed, which is provided by the `kpartx` package.

Some image formats, such as `VHD`, may require additional tools. Please see the `disk-image-create` help output for more information.

Individual elements can also have additional dependencies for the build host. It is recommended you check the documentation for each element you are using to determine if there are any additional dependencies. Of particular note is the need for the *dev-python/pyyaml* package on Gentoo hosts.

Package Installation

On Gentoo you can emerge diskimage-builder directly.

```
emerge app-emulation/diskimage-builder
```

4.1.3 Building An Image

Now that you have diskimage-builder properly *installed* you can get started by building your first disk image.

VM Image

Our first image is going to be a bootable vm image using one of the standard supported distribution *elements* (Ubuntu or Fedora).

The following command will start our image build (distro must be either 'ubuntu' or 'fedora'):

```
disk-image-create <distro> vm
```

This will create a qcow2 file 'image.qcow2' which can then be booted.

Elements

It is important to note that we are passing in a list of *elements* to disk-image-create in our above command. Elements are how we decide what goes into our image and what modifications will be performed.

Some elements provide a root filesystem, such as the ubuntu or fedora element in our example above, which other elements modify to create our image. At least one of these 'distro elements' must be specified when performing an image build. It's worth pointing out that there are many distro elements (you can even create your own), and even multiples for some of the distros. This is because there are often multiple ways to install a distro which are very different. For example: One distro element might use a cloud image while another uses a package installation tool to build a root filesystem for the same distro.

Other elements modify our image in some way. The 'vm' element in our example above ensures that our image has a bootloader properly installed. This is only needed for certain use cases and certain output formats and therefore it is not performed by default.

Output Formats

By default a qcow2 image is created by the disk-image-create command. Other output formats may be specified using the *-t <format>* argument. Multiple output formats can also be specified by comma separation. The supported output formats are:

- qcow2
- tar
- tgz
- squashfs
- vhd
- docker
- raw

Disk Image Layout

The disk image layout (like number of images, partitions, LVM, disk encryption) is something which should be set up during the initial image build: it is mostly not possible to change these things later on.

There are currently two defaults:

- When using the *vm* element a MBR based partition layout is created with exactly one partition that fills up the whole disk and used as root device.
- When not using the *vm* element a plain filesystem image, without any partitioning, is created.

The user can overwrite the default handling by setting the environment variable *DIB_BLOCK_DEVICE_CONFIG*. This variable must hold YAML structured configuration data.

The default when using the *vm* element is:

```
DIB_BLOCK_DEVICE_CONFIG='
  - local_loop:
      name: image0

  - partitioning:
      base: image0
      label: mbr
```

```
      partitions:
        - name: root
          flags: [ boot, primary ]
          size: 100%
          mkfs:
            mount:
              mount_point: /
              fstab:
                options: "defaults"
                fsck-passno: 1'
```

The default when not using the *vm* element is:

```
DIB_BLOCK_DEVICE_CONFIG='
  - local_loop:
      name: image0
      mkfs:
        name: mkfs_root
        mount:
          mount_point: /
          fstab:
            options: "defaults"
            fsck-passno: 1'
```

There are a lot of different options for the different levels. The following sections describe each level in detail.

General Remarks

In general each module that depends on another module has a *base* element that points to the depending base. Also each module has a *name* that can be used to reference the module.

Tree-Like vs. Complete Digraph Configuration

The configuration is specified as a digraph. Each module is a node; a edge is the relation of the current element to its *base*.

Because the general digraph approach is somewhat complex when it comes to write it down, the configuration can also be given as a tree.

Example: The tree like notation

```
mkfs:
  name: root_fs
  base: root_part
  mount:
    mount_point: /
```

is exactly the same as writing

```
mkfs:
  name: root_fs
  base: root_part

mount:
  name: mount_root_fs
```

```
  base: root_fs
  mount_point: /
```

Non existing *name* and *base* entries in the tree notation are automatically generated: the *name* is the name of the base module prepended by the type-name of the module itself; the *base* element is automatically set to the parent node in the tree.

In mostly all cases the much simpler tree notation can be used. Nevertheless there are some use cases when the more general digraph notation is needed. Example: when there is the need to combine two or more modules into one new, like combining a couple of physical volumes into one volume group.

Tree and digraph notations can be mixed as needed in a configuration.

Limitations

To provide an interface towards the existing elements, there are currently three fixed keys used - which are not configurable:

- *root-label*: this is the label of the block device that is mounted at */*.
- *image-block-partition*: if there is a block device with the name *root* this is used else the block device with the name *image0* is used.
- *image-path*: the path of the image that contains the root file system is taken from the *image0*.

Level 0

Module: Local Loop

This module generates a local image file and uses the loop device to create a block device from it. The symbolic name for this module is *local_loop*.

Configuration options:

name (mandatory) The name of the image. This is used as the name for the image in the file system and also as a symbolic name to be able to reference this image (e.g. to create a partition table on this disk).

size (optional) The size of the disk. The size can be expressed using unit names like TiB (1024^4 bytes) or GB (1000^3 bytes). Examples: 2.5GiB, 12KB. If the size is not specified here, the size as given to disk-image-create (–image-size) or the automatically computed size is used.

directory (optional) The directory where the image is created.

Example:

```
local_loop:
  name: image0

local_loop:
  name: data_image
  size: 7.5GiB
  directory: /var/tmp
```

This creates two image files and uses the loop device to use them as block devices. One image file called *image0* is created with default size in the default temp directory. The second image has the size of 7.5GiB and is created in the */var/tmp* folder.

Level 1

Module: Partitioning

This module generates partitions on existing block devices. This means that it is possible to take any kind of block device (e.g. LVM, encrypted, ...) and create partition information in it.

The symbolic name for this module is *partitioning*.

Currently the only supported partitioning layout is Master Boot Record *MBR*.

It is possible to create primary or logical partitions or a mix of them. The numbering of the primary partitions will start at 1, e.g. */dev/vda1*; logical partitions will typically start with *5*, e.g. */dev/vda5* for the first partition, */dev/vda6* for the second and so on.

The number of logical partitions created by this module is theoretical unlimited and it was tested with more than 1000 partitions inside one block device. Nevertheless the Linux kernel and different tools (like *parted*, *sfdisk*, *fdisk*) have some default maximum number of partitions that they can handle. Please consult the documentation of the appropriate software you plan to use and adapt the number of partitions.

Partitions are created in the order they are configured. Primary partitions - if needed - must be first in the list.

There are the following key / value pairs to define one partition table:

base (mandatory) The base device where to create the partitions in.

label (mandatory) Possible values: 'mbr' This uses the Master Boot Record (MBR) layout for the disk. (There are currently plans to add GPT later on.)

align (optional - default value '1MiB') Set the alignment of the partition. This must be a multiple of the block size (i.e. 512 bytes). The default of 1MiB (~ 2048 * 512 bytes blocks) is the default for modern systems and known to perform well on a wide range of targets. For each partition there might be some space that is not used - which is *align* - 512 bytes. For the default of 1MiB exactly 1048064 bytes (= 1 MiB - 512 byte) are not used in the partition itself. Please note that if a boot loader should be written to the disk or partition, there is a need for some space. E.g. grub needs 63 * 512 byte blocks between the MBR and the start of the partition data; this means when grub will be installed, the *align* must be set at least to 64 * 512 byte = 32 KiB.

partitions (mandatory) A list of dictionaries. Each dictionary describes one partition.

The following key / value pairs can be given for each partition:

name (mandatory) The name of the partition. With the help of this name, the partition can later be referenced, e.g. when creating a file system.

flags (optional) List of flags for the partition. Default: empty. Possible values:

boot Sets the boot flag for the partition

primary Partition should be a primary partition. If not set a logical partition will be created.

size (mandatory) The size of the partition. The size can either be an absolute number using units like *10GiB* or *1.75TB* or relative (percentage) numbers: in the later case the size is calculated based on the remaining free space.

type (optional) The partition type stored in the MBR partition table entry. The default value is '0x83' (Linux Default partition). Any valid one byte hexadecimal value may be specified here.

Example:

```
- partitioning:
    base: image0
    label: mbr
    partitions:
      - name: part-01
```

```
        flags: [ boot ]
        size: 1GiB
      - name: part-02
        size: 100%

- partitioning:
    base: data_image
    label: mbr
    partitions:
      - name: data0
        size: 33%
      - name: data1
        size: 50%
      - name: data2
        size: 100%
```

On the *image0* two partitions are created. The size of the first is 1GiB, the second uses the remaining free space. On the *data_image* three partitions are created: all are about 1/3 of the disk size.

Module: Lvm ····------

This module generates volumes on existing block devices. This means that it is possible to take any previous created partition, and create volumes information in it.

The symbolic name for this module is *lvm*.

There are the following key / value pairs to define one set of volumes:

pvs (mandatory) A list of dictionaries. Each dictionary describes one physical volume.

vgs (mandatory) A list of dictionaries. Each dictionary describes one volume group.

lvs (mandatory) A list of dictionaries. Each dictionary describes one logical volume.

The following key / value pairs can be given for each *pvs*:

name (mandatory) The name of the physical volume. With the help of this name, the physical volume can later be referenced, e.g. when creating a volume group.

base (mandatory) The name of the partition where the physical volume needs to be created.

options (optional) List of options for the physical volume. It can contain any option supported by the *pvcreate* command.

The following key / value pairs can be given for each *vgs*:

name (mandatory) The name of the volume group. With the help of this name, the volume group can later be referenced, e.g. when creating a logical volume.

base (mandatory) The name(s) of the physical volumes where the volume groups needs to be created. As a volume group can be created on one or more physical volumes, this needs to be a list.

options (optional) List of options for the volume group. It can contain any option supported by the *vgcreate* command.

The following key / value pairs can be given for each *lvs*:

name (mandatory) The name of the logical volume. With the help of this name, the logical volume can later be referenced, e.g. when creating a filesystem.

base (mandatory) The name of the volume group where the logical volume needs to be created.

size (optional) The exact size of the volume to be created. It accepts the same syntax as the -L flag of the *lvcreate* command.

extents (optional) The relative size in extents of the volume to be created. It accepts the same syntax as the -l flag of the *lvcreate* command. Either size or extents need to be passed on the volume creation.

options (optional) List of options for the logical volume. It can contain any option supported by the *lvcreate* command.

Example:

On the *root* partition a physical volume is created. On that physical volume, a volume group is created. On top of this volume group, six logical volumes are created.

Please note that in order to build images that are bootable using volumes, your ramdisk image will need to have that support. If the image you are using does not have it, you can add the needed modules and regenerate it, by including the *dracut-regenerate* element when building it.

Level 2

Module: Mkfs

This module creates file systems on the block device given as *base*. The following key / value pairs can be given:

base (mandatory) The name of the block device where the filesystem will be created on.

name (mandatory) The name of the partition. This can be used to reference (e.g. mounting) the filesystem.

type (mandatory) The type of the filesystem, like *ext4* or *xfs*.

label (optional - defaults to the name) The label of the filesystem. This can be used e.g. by grub or in the fstab.

opts (optional - defaults to empty list) Options that will passed to the mkfs command.

uuid (optional - no default / not used if not givem) The UUID of the filesystem. Not all file systems might support this. Currently there is support for *ext2*, *ext3*, *ext4* and *xfs*.

Example:

```
- mkfs:
    name: mkfs_root
    base: root
    type: ext4
    label: cloudimage-root
    uuid: b733f302-0336-49c0-85f2-38ca109e8bdb
    opts: "-i 16384"
```

Level 3

Module: Mount

This module mounts a filesystem. The options are:

base (mandatory) The name of the filesystem that will be mounted.

name (mandatory) The name of the mount point. This can be used for reference the mount (e.g. creating the fstab).

mount_point (mandatory) The mount point of the filesystem.

There is no need to list the mount points in the correct order: an algorithm will automatically detect the mount order.

Example:

```
- mount:
    name: root_mnt
    base: mkfs_root
    mount_point: /
```

Level 4

Module: fstab

This module creates fstab entries. The following options exists. For details please consult the fstab man page.

base (mandatory) The name of the mount point that will be written to fstab.

name (mandatory) The name of the fstab entry. This can be used later on as reference - and is currently unused.

options (optional, defaults to *default*) Special mount options can be given. This is used as the fourth field in the fstab entry.

dump-freq (optional, defaults to 0 - don't dump) This is passed to dump to determine which filesystem should be dumped. This is used as the fifth field in the fstab entry.

fsck-passno (optional, defaults to 2) Determines the order to run fsck. Please note that this should be set to 1 for the root file system. This is used as the sixth field in the fstab entry.

Example:

```
- fstab:
    name: var_log_fstab
    base: var_log_mnt
    options: nodev,nosuid
    dump-freq: 2
```

Filesystem Caveat

By default, disk-image-create uses a 4k byte-to-inode ratio when creating the filesystem in the image. This allows large 'whole-system' images to utilize several TB disks without exhausting inodes. In contrast, when creating images intended for tenant instances, this ratio consumes more disk space than an end-user would expect (e.g. a 50GB root disk has 47GB avail.). If the image is intended to run within a tens to hundrededs of gigabyte disk, setting the byte-to-inode ratio to the ext4 default of 16k will allow for more usable space on the instance. The default can be overridden by passing `--mkfs-options` like this:

```
disk-image-create --mkfs-options '-i 16384' <distro> vm
```

You can also select a different filesystem by setting the `FS_TYPE` environment variable.

Note `--mkfs-options` are options passed to the mfks *driver*, rather than `mkfs` itself (i.e. after the initial *-t* argument).

Speedups

If you have 4GB of available physical RAM (as reported by /proc/meminfo MemTotal), or more, diskimage-builder will create a tmpfs mount to build the image in. This will improve image build time by building it in RAM. By default, the tmpfs file system uses 50% of the available RAM. Therefore, the RAM should be at least the double of the minimum tmpfs size required.

For larger images, when no sufficient amount of RAM is available, tmpfs can be disabled completely by passing –no-tmpfs to disk-image-create. ramdisk-image-create builds a regular image and then within that image creates ramdisk.

If tmpfs is not used, you will need enough room in /tmp to store two uncompressed cloud images. If tmpfs is used, you would still need /tmp space for one uncompressed cloud image and about 20% of that image for working files.

Chosing an Architecture

If needed you can specify an override the architecture selection by passing a `-a` argument like:

```
disk-image-create -a <arch> ...
```

Notes about PowerPC Architectures

PowerPC can operate in either Big or Little Endian mode. `ppc64` always refers to Big Endian operation. When running in little endian mode it can be referred to as `ppc64le` or `ppc64el`.

Typically `ppc64el` refers to a `.deb` based distribution architecture, and `ppc64le` refers to a `.rpm` based distribution. Regardless of the distribution the kernel architecture is always `ppc64le`.

4.1.4 Install Types

Install types permit elements to be installed from different sources, such as git repositories, distribution packages, or pip. The default install type is 'source' but it can be modified on the disk-image-create command line via the –install-type option. For example you can set:

> –install-type=package

to enable package installs by default. Alternately, you can also set DIB_DEFAULT_INSTALLTYPE.

Many elements expose different install types. The different implementations live under *<install-dir-prefix>-<install-type>-install* directories under an element's install.d. The base element enables the chosen install type by symlinking the correct hook scripts under install.d directly. *<install-dir-prefix>* can be a string of alphanumeric and '-' characters, but typically corresponds to the element name.

For example, the nova element would provide:

> nova/install.d/nova-package-install/74-nova nova/install.d/nova-source-install/74-nova

The following symlink would be created for the package install type:

> install.d/74-nova -> nova-package-install/74-nova

Or, for the source install type:

> install.d/74-nova -> nova-source-install/74-nova

All other scripts that exist under install.d for an element will be executed as normal. This allows common install code to live in a script under install.d.

To set the install type for an element define an environment variable *DIB_INSTALLTYPE_<install_dir_prefx>*. Note that if you used - characters in your install directory prefix, those need to be replaced with _ in the environment variable.

For example, to enable the package install type for the set of nova elements that use *nova* as the install type prefix, define the following:

> export DIB_INSTALLTYPE_nova=package

4.2 Developer Guide

4.2.1 Design

Images are built using a chroot and bind mounted /proc /sys and /dev. The goal of the image building process is to produce blank slate machines that have all the necessary bits to fulfill a specific purpose in the running of an OpenStack cloud: e.g. a nova-compute node. Images produce either a filesystem image with a label of cloudimg-rootfs, or can be customised to produce whole disk images (but will still contain a filesystem labelled cloudimg-rootfs). Once the file system tree is assembled a loopback device with filesystem (or partition table and file system) is created and the tree copied into it. The file system created is an ext4 filesystem just large enough to hold the file system tree and can be resized up to 1PB in size.

To produce the smallest image the utility fstrim is used. When deleting a file the space is simply marked as free on the disk, the file is still there until it is overwritten. fstrim informs the underlying disk to drop those bytes the end result of which is like writting zeros over those sectors. The same effect could be achieved by creating a large file full of zeros and removing that file, however that method is far more IO intensive.

An element is a particular set of code that alters how the image is built, or runs within the chroot to prepare the image. E.g. the local-config element copies in the http proxy and ssh keys of the user running the image build process into the image, whereas the vm element makes the image build a regular VM image with partition table and installed grub boot sector. The mellanox element adds support for mellanox infiniband hardware to both the deploy ramdisk and the built images.

Images must specify a base distribution image element. Currently base distribution elements exist for fedora, rhel, ubuntu, debian and opensuse. Other distributions may be added in future, the infrastructure deliberately makes few assumptions about the exact operating system in use. The base image has opensshd running (a new key generated on first boot) and accepts keys via the cloud metadata service, loading them into the distribution specific default user account.

The goal of a built image is to have any global configuration ready to roll, but nothing that ties it to a specific cloud instance: images should be able to be dropped into a test cloud and validated, and then deployed into a production cloud (usually via bare metal nova) for production use. As such, the image contents can be modelled as three distinct portions:

- global content: the actual code, kernel, always-applicable config (like disabling password authentication to sshd).
- metadata / config management provided configuration: user ssh keys, network address and routes, configuration management server location and public key, credentials to access other servers in the cloud. These are typically refreshed on every boot.
- persistent state: sshd server key, database contents, swift storage areas, nova instance disk images, disk image cache. These would typically be stored on a dedicated partition and not overwritten when re-deploying the image.

The goal of the image building tools is to create machine images that contain the correct global content and are ready for 'last-mile' configuration by the nova metadata API, after which a configuration management system can take over (until the next deploy, when it all starts over from scratch).

4.2.2 Components

disk-image-create [-a i386\|amd64\|armhf\|arm64] -o filename {element} [{element} . . .]

> Create an image of element {element}, optionally mixing in other elements. Element dependencies are automatically included. Support for other architectures depends on your environment being able to run binaries of that platform and/or packages being available for the architecture. For instance, to enable armhf on Ubuntu install the qemu-user-static package, or to enable arm64 on CentOS setup the RDO

aarch64 package repositories. The default output format from disk-image-create is qcow2. To instead output a tarball pass in "-t tar". This tarball could then be used as an image for a linux container(see docs/docker.md).

ramdisk-image-create -o filename {element} [{element} . . .]

Create a kernel+ ramdisk pair for running maintenance on bare metal machines (deployment, inventory, burnin etc).

To generate kernel+ramdisk pair for use with nova-baremetal, use:

```
ramdisk-image-create -o deploy.ramdisk deploy-baremetal
```

To generate kernel+ramdisk pair for use with ironic, use:

```
ramdisk-image-create -o deploy.ramdisk ironic-agent
```

element-info

Extract information about elements.

tests/run_functests.sh

This runs a set of functional tests for diskimage-builder.

elements can be found in the top level elements directory.

4.2.3 Developer Installation

Note that for non-development use you can use distribution packages or install the latest release via `pip` in a `virtualenv`.

For development purposes, you can use `pip -e` to install the latest git tree checkout into a local development/testing `virtualenv`, or use `tox -e venv -- disk-image-create` to run within a `tox` created environment.

For example, to create a `virtualenv` and install

```
$ mkdir dib
$ cd dib
$ virtualenv env
$ source env/bin/activate
$ git clone https://git.openstack.org/openstack/diskimage-builder
$ cd diskimage-builder
$ pip install -e .
```

4.2.4 Invocation

The scripts can generally just be run. Options can be set on the command line or by exporting variables to override those present in lib/img-defaults. -h to get help.

The image building scripts expect to be able to invoke commands with sudo, so if you want them to run non-interactively, you should either run them as root, with sudo -E, or allow your build user to run any sudo command without password.

The variable `ELEMENTS_PATH` is a colon (:) separated path list to search for elements. The included `elements` tree is used when no path is supplied and is always added to the end of the path if a path is supplied. Earlier elements will override later elements, i.e. with `ELEMENTS_PATH=foo:bar` the element `my-element` will be chosen from `foo/my-element` over `bar/my-element`, or any in-built element of the same name.

By default, the image building scripts will not overwrite existing disk images, allowing you to compare the newly built image with the existing one. To change that behaviour, set the variable `OVERWRITE_OLD_IMAGE` to any value that isn't `0`. If this value is zero then any existing image will be moved before the new image is written to the destination.

Setting the variable `DIB_SHOW_IMAGE_USAGE` will print out a summarised disk-usage report for the final image of files and directories over 10MiB in size. Setting `DIB_SHOW_IMAGE_USAGE_FULL` will show all files and directories. These settings can be useful additions to the logs in automated build situations where debugging image-growth may be important.

4.2.5 Caches and offline mode

Since retrieving and transforming operating system image files, git repositories, Python or Ruby packages, and so on can be a significant overhead, we cache many of the inputs to the build process.

The cache location is read from `DIB_IMAGE_CACHE`. *Developing Elements* describes the interface within disk-image-builder for caching.

When invoking disk-image-builder, the `--offline` option will instruct disk-image-builder to not refresh cached resources. Alternatively you can set `DIB_OFFLINE=1`.

Note that we don't maintain operating system package caches, instead depending on your local infrastructure (e.g. Squid cache, or an APT or Yum proxy) to facilitate caching of that layer, so you need to arrange independently for offline mode. For more information about setting up a squid proxy, consult the TripleO documentation.

Base images

These are cached by the standard elements - *fedora*, *redhat-common*, *ubuntu*, *debian* and *opensuse*.

source-repositories

Git repositories and tarballs obtained via the *source-repositories* element will be cached.

PyPI

The *pypi* element will bind mount a PyPI mirror from the cache dir and configure pip and easy-install to use it.

4.2.6 Developing Elements

Conform to the following conventions:

- Use the environment for overridable defaults, prefixing environment variable names with `DIB_`. For example:

```
DIB_MYDEFAULT=${DIB_MYDEFAULT:-default}
```

  If you do not use the `DIB` prefix you may find that your overrides are discarded as the build environment is sanitised.

- Consider that your element co-exists with many others and try to guard against undefined behaviours. Some examples:
  - Two elements use the source-repositories element, but use the same filename for the source-repositories config file. Files such as these (and indeed the scripts in the various .d directories *listed below*) should be named such that they are unique. If they are not unique, when the combined tree is created by disk-image-builder for injecting into the build environment, one of the files will be overwritten.

  - Two elements copy different scripts into `/usr/local/bin` with the same name. If they both use `set -e` and `cp -n` then the conflict will be caught and cause the build to fail.

- If your element mounts anything into the image build tree (`$TMP_BUILD_DIR`) then it will be automatically unmounted when the build tree is unmounted - and not remounted into the filesystem image - if the mount point is needed again, your element will need to remount it at that point.
- If caching is required, elements should use a location under `$DIB_IMAGE_CACHE`.
- Elements should allow for remote data to be cached. When `$DIB_OFFLINE` is set, this cached data should be used if possible. See the *Global image-build variables* section of this document for more information.
- Elements in the upstream diskimage-builder elements should not create executables which run before 10- or after 90- in any of the phases if possible. This is to give downstream elements the ability to easily make executables which run after our upstream ones.

Phase Subdirectories

Make as many of the following subdirectories as you need, depending on what part of the process you need to customise. The subdirectories are executed in the order given here. Scripts within the subdirectories should be named with a two-digit numeric prefix, and are executed in numeric order.

Only files which are marked executable (+x) will be run, so other files can be stored in these directories if needed. As a convention, we try to only store executable scripts in the phase subdirectories and store data files elsewhere in the element.

The phases are:

1. `root.d`
2. `extra-data.d`
3. `pre-install.d`
4. `install.d`
5. `post-install.d`
6. `block-device.d`
7. `finalise.d`
8. `cleanup.d`

`root.d` Create or adapt the initial root filesystem content. This is where alternative distribution support is added, or customisations such as building on an existing image.

Only one element can use this at a time unless particular care is taken not to blindly overwrite but instead to adapt the context extracted by other elements.

- runs: **outside chroot**
- inputs:
  - `$ARCH=i386|amd64|armhf|arm64`
  - `$TARGET_ROOT=/path/to/target/workarea`

`extra-data.d` Pull in extra data from the host environment that hooks may need during image creation. This should copy any data (such as SSH keys, http proxy settings and the like) somewhere under `$TMP_HOOKS_PATH`.

- runs: **outside chroot**
- inputs: `$TMP_HOOKS_PATH`

- outputs: None

pre-install.d Run code in the chroot before customisation or packages are installed. A good place to add apt repositories.

- runs: **in chroot**

install.d Runs after `pre-install.d` in the chroot. This is a good place to install packages, chain into configuration management tools or do other image specific operations.

- runs: **in chroot**

post-install.d Run code in the chroot. This is a good place to perform tasks you want to handle after the OS/application install but before the first boot of the image. Some examples of use would be:

Run `chkconfig` to disable unneeded services

Clean the cache left by the package manager to reduce the size of the image.

- runs: **in chroot**

block-device.d Customise the block device that the image will be made on (for example to make partitions). Runs after the target tree has been fully populated but before the `cleanup.d` phase runs.

- runs: **outside chroot**
- inputs:
  - `$IMAGE_BLOCK_DEVICE={path}`
  - `$TARGET_ROOT={path}`
- outputs: `$IMAGE_BLOCK_DEVICE={path}`

finalise.d Perform final tuning of the root filesystem. Runs in a chroot after the root filesystem content has been copied into the mounted filesystem: this is an appropriate place to reset SELinux metadata, install grub bootloaders and so on.

Because this happens inside the final image, it is important to limit operations here to only those necessary to affect the filesystem metadata and image itself. For most operations, `post-install.d` is preferred.

- runs: **in chroot**

cleanup.d Perform cleanup of the root filesystem content. For instance, temporary settings to use the image build environment HTTP proxy are removed here in the dpkg element.

- runs: outside chroot
- inputs:
  - `$ARCH=i386|amd64|armhf|arm64`
  - `$TARGET_ROOT=/path/to/target/workarea`

Other Subdirectories

Elements may have other subdirectories that are processed by specific elements rather than the diskimage-builder tools themselves.

One example of this is the `bin` directory. The *rpm-distro*, *dpkg* and *opensuse* elements install all files found in the `bin` directory into `/usr/local/bin` within the image as executable files.

Environment Variables

To set environment variables for other hooks, add a file to your element `environment.d`. This directory contains bash script snippets that are sourced before running scripts in each phase. Note that because environment includes are sourced together, they should not set global flags like `set -x` because they will affect all preceeding imports.

Dependencies

Each element can use the following files to define or affect dependencies:

`element-deps` A plain text, newline separated list of elements which will be added to the list of elements built into the image at image creation time.

`element-provides` A plain text, newline separated list of elements which are provided by this element. These elements will be excluded from elements built into the image at image creation time.

For example if element A depends on element B and element C includes element B in its `element-provides` file and A and C are included when building an image, then B is not used.

Operating system elements

Some elements define the base structure for an operating system – for example, the `opensuse` element builds a base openSUSE system. Such elements have more requirements than the other elements:

- they must have `operating-system` in their element-provides, so this indicates they are an "operating system".
- they must export the `DISTRO_NAME` environment variable with the name of the distribution built, using an environment.d script. For example, the `opensuse` element exports `DISTRO_NAME=opensuse`.

Ramdisk Elements

Ramdisk elements support the following files in their element directories:

`binary-deps.d` Text files listing executables required to be fed into the ramdisk. These need to be present in `$PATH` in the build chroot (i.e. need to be installed by your elements as described above).

`init.d` POSIX shell script fragments that will be appended to the default script executed as the ramdisk is booted (`/init`).

`ramdisk-install.d` Called to copy files into the ramdisk. The variable `$TMP_MOUNT_PATH` points to the root of the tree that will be packed into the ramdisk.

`udev.d` `udev` rules files that will be copied into the ramdisk.

Element coding standard

- lines should not include trailing whitespace.
- there should be no hard tabs in the file.
- indents are 4 spaces, and all indentation should be some multiple of them.
- *do* and *then* keywords should be on the same line as the if, while or for conditions.

Global image-build variables

DIB_OFFLINE This is always set. When not empty, any operations that perform remote data access should avoid it if possible. If not possible the operation should still be attempted as the user may have an external cache able to keep the operation functional.

DIB_IMAGE_ROOT_FS_UUID This contains the UUID of the root filesystem, when diskimage-builder is building a disk image. This works only for ext filesystems.

DIB_IMAGE_CACHE Path to where cached inputs to the build process are stored. Defaults to `~/.cache/image_create`.

Structure of an element

The above-mentioned global content can be further broken down in a way that encourages composition of elements and reusability of their components. One possible approach to this would be to label elements as either a "driver", "service", or "config" element. Below are some examples.

- Driver-specific elements should only contain the necessary bits for that driver:

```
elements/
   driver-mellanox/
      init           - modprobe line
      install.d/
         10-mlx      - package installation
```

- An element that installs and configures Nova might be a bit more complex, containing several scripts across several phases:

```
elements/
   service-nova/
      source-repository-nova - register a source repository
      pre-install.d/
         50-my-ppa           - add a PPA
      install.d/
         10-user             - common Nova user accts
         50-my-pack          - install packages from my PPA
         60-nova             - install nova and some dependencies
```

- In the general case, configuration should probably be handled either by the meta-data service (eg, o-r-c) or via normal CM tools (eg, salt). That being said, it may occasionally be desirable to create a set of elements which express a distinct configuration of the same software components.

In this way, depending on the hardware and in which availability zone it is to be deployed, an image would be composed of:

- zero or more driver-elements
- one or more service-elements
- zero or more config-elements

It should be noted that this is merely a naming convention to assist in managing elements. Diskimage-builder is not, and should not be, functionally dependent upon specific element names.

diskimage-builder has the ability to retrieve source code for an element and place it into a directory on the target image during the extra-data phase. The default location/branch can then be overridden by the process running diskimage-builder, making it possible to use the same element to track more then one branch of a git repository or to get source for a local cache. See *source-repositories* for more information.

Finding other elements

DIB exposes an internal `$IMAGE_ELEMENT_YAML` variable which provides elements access to the full set of included elements and their paths. This can be used to process local in-element files across all the elements (`pkg-map` for example).

```
import os
import yaml

elements = yaml.load(os.getenv('IMAGE_ELEMENT_YAML'))
for element, path in elements:
   ...
```

For elements written in Bash, there is a function `get_image_element_array` that can be used to instantiate an associative-array of elements and paths (note arrays can not be exported in bash).

```
# note eval to expand the result of the get function
eval declare -A image_elements=($(get_image_element_array))
for i in ${!image_elements[$i]}; do
  element=$i
  path=${image_elements[$i]}
done
```

Debugging elements

Export `break` to drop to a shell during the image build. Break points can be set either before or after any of the hook points by exporting "break=[beforelafter]-hook-name". Multiple break points can be specified as a comma-delimited string. Some examples:

- `break=before-block-device-size` will break before the block device size hooks are called.
- `break=before-pre-install` will break before the pre-install hooks.
- `break=after-error` will break after an error during an in target hookpoint.

The *manifests* element will make a range of manifest information generated by other elements available for inspection inside and outside the built image. Environment and command line arguments are captured as described in the documentation and can be useful for debugging.

Images are built such that the Linux kernel is instructed not to switch into graphical consoles (i.e. it will not activate KMS). This maximises compatibility with remote console interception hardware, such as HP's iLO. However, you will typically only see kernel messages on the console - init daemons (e.g. upstart) will usually be instructed to output to a serial console so nova's console-log command can function. There is an element in the tripleo-image-elements repository called "remove-serial-console" which will force all boot messages to appear on the main console.

Ramdisk images can be debugged at run-time by passing `troubleshoot` as a kernel command line argument, or by pressing "t" when an error is reached. This will spawn a shell on the console (this can be extremely useful when network interfaces or disks are not detected correctly).

Testing Elements

An element can have functional tests encapsulated inside the element itself. The tests can be written either as shell or python unit tests.

shell

In order to create a test case, follow these steps:

- Create a directory called `test-elements` inside your element.
- Inside the test-elements directory, create a directory with the name of your test case. The test case directory should have the same structure as an element. For example:

```
elements/apt-sources/test-elements/test-case-1
```

- Assert state during each of the element build phases you would like to test. You can exit 1 to indicate a failure.
- To exit early and indicate a success, touch a file `/tmp/dib-test-should-fail` in the image chroot, then exit 1.

Tests are run with `tools/run_functests.sh`. Running `run_functests.sh -l` will show available tests (the example above would be called `apt-sources/test-case-1`, for example). Specify your test (or a series of tests as separate arguments) on the command line to run it. If it should not be run as part of the default CI run, you can submit a change with it added to `DEFAULT_SKIP_TESTS` in that file.

Running the functional tests is time consuming. Multiple parallel jobs can be started by specifying `-j <job count>`. Each of the jobs uses a lot resources (CPU, disk space, RAM) - therefore the job count must carefully be chosen.

python

To run functional tests locally, install and start docker, then use the following tox command:

```
tox -efunc
```

Note that running functional tests requires *sudo* rights, thus you may be asked for your password.

To run functional tests for one element, append its name to the command:

```
tox -efunc ironic-agent
```

Additionally, elements can be tested using python unittests. To create a a python test:

- Create a directory called `tests` in the element directory.
- Create an empty file called `__init__.py` to make it into a python package.
- Create your test files as `test\whatever.py`, using regular python test code.

To run all the tests use testr - `testr run`. To run just some tests provide one or more regex filters - tests matching any of them are run - `testr run apt-proxy`.

Third party elements

Additional elements can be incorporated by setting `ELEMENTS_PATH`, for example if one were building tripleo-images, the variable would be set like:

```
export ELEMENTS_PATH=tripleo-image-elements/elements
disk-image-create rhel7 cinder-api
```

Linting

You should always run `bin/dib-lint` over your elements. It will warn you of common issues.

sudo

Using `sudo` outside the chroot environment can cause breakout issues where you accidentally modify parts of the host system. `dib-lint` will warn if it sees `sudo` calls that do not use the path arguments given to elements running outside the chroot.

To disable the error for a call you know is safe, add

```
# dib-lint: safe_sudo
```

to the end of the `sudo` command line. To disable the check for an entire file, add

```
# dib-lint: disable=safe_sudo
```

4.2.7 dib-lint

dib-lint provides a way to check for common errors in diskimage-builder elements. To use it, simply run the `dib-lint` script in a directory containing an `elements` directory. The checks will be run against every file found under `elements`.

The following is a list of what is currently caught by dib-lint:

- executable: Ensure any files that begin with #! are executable
- indent: Ensure that all source code is using an indent of four spaces
- element-deps ordering: Ensure all element-deps files are alphabetized
- /bin/bash: Ensure all scripts are using bash explicitly
- sete: Ensure all scripts are set -e
- setu: Ensure all scripts are set -u
- setpipefail: Ensure all scripts are set -o pipefail
- dibdebugtrace: Ensure all scripts respect the DIB_DEBUG_TRACE variable
- tabindent: Ensure no tabs are used for indentation
- newline: Ensure that every file ends with a newline
- mddocs: Ensure that only markdown-formatted documentation is used
- yaml validation: Ensure that any yaml files in the repo have valid syntax

Some of the checks can be omitted, either for an entire project or for an individual file. Project exclusions go in tox.ini, using the following section format:

```
[dib-lint]
ignore=sete setpipefail
```

This will cause the set -e and set -o pipefail checks to be ignored.

File-specific exclusions are specified as a comment in the relevant file, using the following format:

```
# dib-lint: disable=sete setpipefail
```

This will exclude the same tests, but only for the file in which the comment appears.

Only some of the checks can be disabled. The ones available for exclusion are:

- executable
- indent
- sete
- setu
- setpipefail
- dibdebugtrace
- tabindent
- newline
- mddocs

4.2.8 Stable Interfaces

diskimage-builder and the elements provide several 'stable' interfaces for both developers and users which we aim to preserve during a major version release. These interfaces consist of:

The names and arguments of the executable scripts included with diskimage-builder in the bin directory will remain stable.

The environment variables that diskimage-builder provides for elements to use will remain stable.

The environment variables documented in each element and the values accepted by these environment variables will remain stable.

Required environment variables for an element will not be added.

Support for build or target distributions will not be removed.

This documentation explains how to get started with creating your own disk-image-builder elements as well as some high level concepts for element creation.

4.2.9 Quickstart

To get started developing with `diskimage-builder`, install to a `virtualenv`:

```
$ mkdir dib
$ cd dib
$ virtualenv env
$ source env/bin/activate
$ git clone https://git.openstack.org/openstack/diskimage-builder
$ cd diskimage-builder
$ pip install -e .
```

You can now simply use `disk-image-create` to start building images and testing your changes. When you are done editing, use `git review` to submit changes to the upstream gerrit.

4.2.10 Python module documentation

For internal documentation on the DIB python components, see the modindex

4.2.11 Finding Work

We maintain a list of low-hanging-fruit tags on launchpad:

- *https://bugs.launchpad.net/diskimage-builder/+bugs?field.tag=low-hanging-fruit* <https://bugs.launchpad.net/diskimage-builder/+bugs?field.tag=low-hanging-fruit>

4.3 Elements

Elements are found in the subdirectory elements. Each element is in a directory named after the element itself. Elements *should* have a README.rst in the root of the element directory describing what it is for.

4.3.1 apt-conf

This element overrides the default apt.conf for APT based systems.

Environment Variables

DIB_APT_CONF:

Required No

Default None

Description To override *DIB_APT_CONF*, set it to the path to your apt.conf. The new apt.conf will take effect at build time and run time.

Example `DIB_APT_CONF=/etc/apt/apt.conf`

4.3.2 apt-preferences

This element generates the APT preferences file based on the provided manifest provided by the *manifests* element.

The APT preferences file can be used to control which versions of packages will be selected for installation. APT uses a priority system to make this determination. For more information about APT preferences, see the apt_preferences(5) man page.

Environment Variables

DIB_DPKG_MANIFEST:

Required No

Default None

Description The manifest file to generate the APT preferences file from.

Example
```
DIB\_DPKG\_MANIFEST=~/image.d/dib-manifests/
    dib-manifest-dpkg-image
```

4.3.3 apt-sources

Specify an apt sources.list file which is used during image building and then remains on the image when it is run.

Environment Variables

DIB_APT_SOURCES

Required No

Default None (Does not replace sources.list file)

Description Path to a file on the build host which is used in place of `/etc/apt/sources.list`

Example `DIB_APT_SOURCES=/etc/apt/sources.list` will use the same sources.list as the build host.

4.3.4 baremetal

This is the baremetal (IE: real hardware) element.

Does the following:

- extracts the kernel and initial ramdisk of the built image.

Optional parameters:

- DIB_BAREMETAL_KERNEL_PATTERN and DIB_BAREMETAL_INITRD_PATTERN may be supplied to specify which kernel files are preferred; this can be of use when using custom kernels that don't fit the standard naming patterns. Both variables must be provided in order for them to have any effect.

4.3.5 base

This is the base element.

Almost all users will want to include this in their disk image build, as it includes a lot of useful functionality.

The *DIB_CLOUD_INIT_ETC_HOSTS* environment variable can be used to customize cloud-init's management of */etc/hosts*:

- If the variable is set to something, write that value as cloud-init's manage_etc_hosts.
- If the variable is set to an empty string, don't create manage_etc_hosts setting (cloud-init will use its default value).
- If the variable is not set, use "localhost" for now. Later, not setting the variable will mean using cloud-init's default. (To preserve diskimage-builder's current default behavior in the future, set the variable to "localhost" explicitly.)

Notes:

- If you are getting warnings during the build about your locale being missing, consider installing/generating the relevant locale. This may be as simple as having language-pack-XX installed in the pre-install stage

4.3.6 bootloader

Installs `grub[2]` on boot partition on the system. In case GRUB2 is not available in the system, a fallback to Extlinux will happen. It's also possible to enforce the use of Extlinux by exporting a `DIB_EXTLINUX` variable to the environment.

Arguments

- `DIB_GRUB_TIMEOUT` sets the `grub` menu timeout. It defaults to 5 seconds. Set this to 0 (no timeout) for fast boot times.
- `DIB_BOOTLOADER_DEFAULT_CMDLINE` sets the CMDLINE parameters that are appended to the grub.cfg configuration. It defaults to 'nofb nomodeset vga=normal'

4.3.7 cache-url

A helper script to download images into a local cache.

4.3.8 centos-minimal

Create a minimal image based on CentOS 7.

Use of this element will require 'yum' and 'yum-utils' to be installed on Ubuntu and Debian. Nothing additional is needed on Fedora or CentOS.

By default, `DIB_YUM_MINIMAL_CREATE_INTERFACES` is set to enable the creation of `/etc/sysconfig/network-scripts/ifcfg-eth[0|1]` scripts to enable DHCP on the `eth0` & `eth1` interfaces. If you do not have these interfaces, or if you are using something else to setup the network such as cloud-init, glean or network-manager, you would want to set this to `0`.

4.3.9 centos7

Use Centos 7 cloud images as the baseline for built disk images.

For further details see the redhat-common README.

DIB_DISTRIBUTION_MIRROR:

Required No

Default None

Description To use a CentOS Yum mirror, set this variable to the mirror URL before running bin/disk-image-create. This URL should point to the directory containing the `5/6/7` directories.

Example `DIB_DISTRIBUTION_MIRROR=http://amirror.com/centos`

DIB_CLOUD_IMAGES

Required No

Description Set the desired URL to fetch the images from. ppc64le: Currently the CentOS community is working on providing the ppc64le images until then you'll need to set this to a local image file.

4.3.10 cleanup-kernel-initrd

Remove unused kernel/initrd from the image.

4.3.11 cloud-init-datasources

Configures cloud-init to only use an explicit list of data sources.

Cloud-init contains a growing collection of data source modules and most are enabled by default. This causes cloud-init to query each data source on first boot. This can cause delays or even boot problems depending on your environment.

Including this element without setting *DIB_CLOUD_INIT_DATASOURCES* will cause image builds to fail.

Environment Variables

DIB_CLOUD_INIT_DATASOURCES

Required Yes

Default None

Description A comma-separated list of valid data sources to limit the data sources that will be queried for metadata on first boot.

Example `DIB_CLOUD_INIT_DATASOURCES="Ec2"` will enable only the Ec2 data source.

Example `DIB_CLOUD_INIT_DATASOURCES="Ec2, ConfigDrive, OpenStack"` will enable the Ec2, ConfigDrive and OpenStack data sources.

4.3.12 cloud-init-disable-resizefs

The cloud-init resizefs module can be extremely slow and will also unwittingly create a root filesystem that cannot be booted by grub if the underlying partition is too big. This removes it from cloud.cfg, putting the onus for resizing on the user post-boot.

4.3.13 cloud-init-nocloud

Configures cloud-init to only use on-disk metadata/userdata sources. This will avoid a boot delay of 2 minutes while polling for cloud data sources such as the EC2 metadata service.

Empty on-disk sources are created in /var/lib/cloud/seed/nocloud/.

4.3.14 cloud-init

Install's and enables cloud-init for systems that don't come with it pre-installed

Currently only supports Gentoo.

Environment Variables

DIB_CLOUD_INIT_ALLOW_SSH_PWAUTH

Required No

Default password authentication disabled when cloud-init installed

Description customize cloud-init to allow ssh password authentication.

4.3.15 debian-minimal

The `debian-minimal` element uses debootstrap for generating a minimal image.

By default this element creates the latest stable release. The exact setting can be found in the element's `environment.d` directory in the variable `DIB_RELEASE`. If a different release of Debian should be created, the variable `DIB_RELEASE` can be set appropriately.

Note that this element installs `systemd-sysv` as the init system

Element Dependencies

Uses

- *dib-python*
- *pkg-map*
- *debootstrap*

Used by

- *debian*

4.3.16 debian-systemd

You may want to use *systemd* instead of the classic sysv init system. In this case, include this element in your element list.

Note that this works with the `debian` element, not the `debian-minimal` element.

Element Dependencies

Uses

- *debian*

4.3.17 debian-upstart

By default Debian will use sysvinit for booting. If you want to experiment with Upstart, or have need of it due to a need for upstart jobs, this element will build the image with upstart as the init system.

Note that this works with the `debian` element, not the `debian-minimal` element.

Element Dependencies

Uses

- *debian*

4.3.18 debian

This element is based on `debian-minimal` with `cloud-init` and related tools installed. This produces something more like a standard upstream cloud image.

By default this element creates the latest stable release. The exact setting can be found in the `debian-minimal/environment.d` directory in the variable `DIB_RELEASE`. If a different release of Debian should be created, the variable `DIB_RELEASE` can be set appropriately.

Element Dependencies

Uses

- *openssh-server*
- *debian-minimal*

Used by

- *debian-upstart*
- *debian-systemd*

4.3.19 debootstrap

Base element for creating minimal debian-based images.

This element is incomplete by itself, you'll want to use elements like debian-minimal or ubuntu-minimal to get an actual base image.

There are two ways to configure apt-sources:

1. Using the standard way of defining the default, backports, updates and security repositories is the default. In this case you can overwrite the two environment variables to adapt the behavior:

   - `DIB_DISTRIBUTION_MIRROR`: the mirror to use (default: http://deb.debian.org/debian)
   - `DIB_DEBIAN_COMPONENTS`: (default: `main`) a comma separated list of components. For Debian this can be e.g. `main,contrib,non-free`.

   By default only the `main` component is used. If `DIB_DEBIAN_COMPONENTS` (comma separated) from the `debootstrap` element has been set, that list of components will be used instead.

   Backports, updates and security are included unless `DIB_RELEASE` is `unstable`.

2. Complete configuration given in the variable `DIB_APT_SOURCES_CONF`.

   Each line contains exactly one entry for the sources.list.d directory. The first word must be the logical name (which is used as file name with `.list` automatically appended), followed by a colon `:`, followed by the complete repository specification.

```
DIB_APT_SOURCES_CONF=\
  "default:deb http://10.0.0.10/ stretch main contrib
   mysecurity:deb http://10.0.0.10/ stretch-security main contrib"
```

If necessary, a custom apt keyring and debootstrap script can be supplied to the `debootstrap` command via `DIB_APT_KEYRING` and `DIB_DEBIAN_DEBOOTSTRAP_SCRIPT` respectively. Both options require the use of absolute rather than relative paths.

Use of this element will also require the tool 'debootstrap' to be available on your system. It should be available on Ubuntu, Debian, and Fedora. It is also recommended that the 'debian-keyring' package be installed.

The `DIB_OFFLINE` or more specific `DIB_DEBIAN_USE_DEBOOTSTRAP_CACHE` variables can be set to prefer the use of a pre-cached root filesystem tarball.

The `DIB_DEBOOTSTRAP_EXTRA_ARGS` environment variable may be used to pass extra arguments to the debootstrap command used to create the base filesystem image. If –keyring is is used in `DIB_DEBOOTSTRAP_EXTRA_ARGS`, it will override `DIB_APT_KEYRING` if that is used as well.

For further information about `DIB_DEBIAN_DEBOOTSTRAP_SCRIPT`, `DIB_DEBIAN_USE_DEBOOTSTRAP_CACHE` and `DIB_DEBOOTSTRAP_EXTRA_ARGS` please consult "README.rst" of the debootstrap element.

Note on ARM systems

Because there is not a one-to-one mapping of `ARCH` to a kernel package, if you are building an image for ARM on debian, you need to specify which kernel you want in the environment variable `DIB_ARM_KERNEL`. For instance, if you want the `linux-image-mx5` package installed, set `DIB_ARM_KERNEL` to `mx5`.

Element Dependencies

Uses

- *pkg-map*
- *dpkg*

Used by

- *debian-minimal*
- *ubuntu-minimal*

4.3.20 deploy-baremetal

A ramdisk that will expose the machine primary disk over iSCSI and reboot once baremetal-deploy-helper signals it is finished.

4.3.21 deploy-kexec

Boots into the new image once baremetal-deploy-helper signals it is finished by downloading the kernel and ramdisk via tftp, and using the kexec utilities.

4.3.22 deploy-targetcli

Use targetcli for the deploy ramdisk

Provides the necessary scripts and dependencies to use targetcli for exporting the iscsi target in the deploy ramdisk.

Implemented as a dracut module, so will only work with dracut-based ramdisks.

4.3.23 deploy-tgtadm

Use tgtadm and tgtd for the deploy ramdisk

Provides the necessary scripts and dependencies to use tgtadm and tgtd for exporting the iscsi target in the deploy ramdisk.

Will only work with the standard (not dracut) ramdisk.

4.3.24 devuser

Creates a user that is useful for development / debugging. The following environment variables can be useful for configuration:

Environment Variables

DIB_DEV_USER_USERNAME

Required No

Default devuser

Description Username for the created user.

DIB_DEV_USER_SHELL

Required No

Default System default (The useradd default is used)

Description Full path for the shell of the user. This is passed to useradd using the -s parameter. Note that this does not install the (possibly) required shell package.

DIB_DEV_USER_PWDLESS_SUDO

Required No

Default No

Description Enable passwordless sudo for the user.

DIB_DEV_USER_AUTHORIZED_KEYS

Required No

Default $HOME/.ssh/id_{rsa,dsa}.pub

Description Path to a file to copy into this users' .ssh/authorized_keys If this is not specified then an attempt is made to use a the building user's public key. To disable this behavior specify an invalid path for this variable (such as /dev/null).

DIB_DEV_USER_PASSWORD

Required No

Default Password is disabled

Description Set the default password for this user. This is a fairly insecure method of setting the password and is not advised.

4.3.25 dhcp-all-interfaces

Autodetect network interfaces during boot and configure them for DHCP

The rationale for this is that we are likely to require multiple network interfaces for use cases such as baremetal and there is no way to know ahead of time which one is which, so we will simply run a DHCP client on all interfaces with real MAC addresses (except lo) that are visible on the first boot.

On non-Gentoo based distributions the script /usr/local/sbin/dhcp-all-interfaces.sh will be called early in each boot and will scan available network interfaces and ensure they are configured properly before networking services are started.

On Gentoo based distributions we will install the dhcpcd package and ensure the service starts at boot. This service automatically sets up all interfaces found via dhcp and/or dhcpv6 (or SLAAC).

Environment Variables

DIB_DHCP_TIMEOUT

Required No

Default 30

Description Amount of time in seconds that the systemd service will wait to get an address.

Example DIB_DHCP_TIMEOUT=300

4.3.26 dib-init-system

Installs a script (dib-init-system) which outputs the type of init system in use on the target image. Also sets an environment variable `DIB_INIT_SYSTEM` to this value.

Any files placed in a `init-scripts/INIT_SYSTEM` directory inside the element will be copied into the appropriate directory if `INIT_SYSTEM` is in use on the host.

Environment Variables

DIB_INIT_SYSTEM

Description One of upstart, systemd, or sysv depending on the init system in use for the target image.

4.3.27 dib-python

Adds a symlink to */usr/local/bin/dib-python* which points at either a *python2* or *python3* executable as appropriate.

In-chroot scripts should use this as their interpreter (*#!/usr/local/bin/dib-python*) to make scripts that are compatible with both *python2* and *python3*. We can not assume */usr/bin/python* exists, as some platforms have started shipping with only Python 3.

DIB_PYTHON will be exported as the python interpreter. You should use this instead of *python script.py* (e.g. *${DIB_PYTHON} script.py*). Note you can also call */usr/local/bin/dib-python script.py* but in some circumstances, such as creating a *virtualenv*, it can create somewhat confusing references to *dib-python* that remain in the built image.

This does not install a python if one does not exist, and instead fails.

This also exports a variable *DIB_PYTHON_VERSION* which will either be '2' or '3' depending on the python version which dib-python points to.

4.3.28 dib-run-parts

Warning: This element is deprecated and is left only for compatibility. Please read the notes.

This element install the `dib-utils` package to provide `dib-run-parts`.

Previously this element was a part of most base images and copied the internal version of `dib-run-parts` to `/usr/local/bin` during the build. Due to a (longstanding) oversight this was never removed and stayed in the final image. The image build process now uses a private copy of `dib-run-parts` during the build, so this element has become deprecated.

For compatibility this element simply installs the `dib-utils` package, which will provide `dib-run-parts`. However, this is probably better expressed as a dependency in individual elements.

4.3.29 dkms

This is the dkms (Dynamic Kernel Module System) element.

Some distributions such as Fedora and Ubuntu include DKMS in their packaging. In these distros, it is reasonable to include dkms. Other RHEL based derivatives do not include DKMS, so those distros should not use the DKMS element.

4.3.30 docker

Base element for creating images from docker containers.

This element is incomplete by itself, you'll want to add additional elements, such as dpkg or yum to get richer features. At its heart, this element simply exports a root tarball from a named docker container so that other diskimage-builder elements can build on top of it.

The variables *DISTRO_NAME* and *DIB_RELEASE* will be used to decide which docker image to pull, and are required for most other elements. Additionally, the *DIB_DOCKER_IMAGE* environment variable can be set in addition to *DISTRO_NAME* and *DIB_RELEASE* if a different docker image is desired.

4.3.31 dpkg

Provide dpkg specific image building glue.

The ubuntu element needs customisations at the start and end of the image build process that do not apply to RPM distributions, such as using the host machine HTTP proxy when installing packages. These customisations live here, where they can be used by any dpkg based element.

The dpkg specific version of install-packages is also kept here.

Environment Variables

DIB_ADD_APT_KEYS

Required No

Default None

Description If an extra or updated apt key is needed then define `DIB_ADD_APT_KEYS` with the path to a folder. Any key files inside will be added to the key ring before any apt-get commands take place.

Example `DIB_ADD_APT_KEYS=/etc/apt/trusted.gpg.d`

DIB_APT_LOCAL_CACHE

Required No

Default 1

Description By default the `$DIB_IMAGE_CACHE/apt/$DISTRO_NAME` directory is mounted in `/var/cache/apt/archives` to cache the .deb files downloaded during the image creation. Use this variable if you wish to disable the internal cache of the `/var/cache/apt/archives` directory

Example `DIB_APT_LOCAL_CACHE=0` will disable internal caching.

DIB_DISABLE_APT_CLEANUP

Required No

Default 0

Description At the end of a dib run we clean the apt cache to keep the image size as small as possible. Use this variable to prevent cleaning the apt cache at the end of a dib run.

Example `DIB_DISABLE_APT_CLEANUP=1` will disable cleanup.

Element Dependencies

Uses

- *package-installs*
- *install-bin*
- *manifests*

Used by

- *ubuntu-core*
- *debootstrap*
- *ubuntu*

4.3.32 dracut-network

This element was removed in the Pike cycle. Please consider using the dracut-regenerate element instead.

4.3.33 dracut-ramdisk

Build Dracut-based ramdisks

This is an alternative to the *ramdisk* element that uses Dracut to provide the base system functionality instead of Busybox.

For elements that need additional drivers in the ramdisk image, a dracut-drivers.d feature is included that works in a similar fashion to the binary-deps.d feature. The element needing to add drivers should create a dracut-drivers.d directory and populate it with a single file listing all of the kernel modules it needs added to the ramdisk. Comments are not supported in this file. Note that these modules must be installed in the chroot first.

By default, the virtio, virtio_net, and virtio_blk modules are included so that ramdisks are able to function properly in a virtualized environment.

4.3.34 dracut-regenerate

Adds the possibility of regenerating dracut on image build time, giving the possibility to load extra modules. It relies on the `DIB_DRACUT_ENABLED_MODULES` setting, that will accept a yaml blob with the following format:

```
- name: <module1>
  packages:
    - <package1>
    - <package2>
- name: <module2>
  packages:
    - <package3>
    - <package4>
```

By default, this element will bring lvm and crypt modules.

4.3.35 dynamic-login

This element insert a helper script in the image that allows users to dynamically configure credentials at boot time. This is specially useful for troubleshooting.

Troubleshooting an image can be quite hard, specially if you can not get a prompt you can enter commands to find out what went wrong. By default, the images (specially ramdisks) doesn't have any SSH key or password for any user. Of course one could use the `devuser` element to generate an image with SSH keys and user/password in the image but that would be a massive security hole and very it's discouraged to run in production with a ramdisk like that.

This element allows the operator to inject a SSH key and/or change the root password dynamically when the image boots. Two kernel command line parameters are used to do it:

sshkey

> **Description** If the operator append sshkey="$PUBLIC_SSH_KEY" to the kernel command line on boot, the helper script will append this key to the root user authorized_keys.

rootpwd

> **Description** If the operator append rootpwd="$ENCRYPTED_PASSWORD" to the kernel command line on boot, the helper script will set the root password to the one specified by this option. Note that this password must be **encrypted**. Encrypted passwords can be generated using the `openssl` command, e.g: *openssl passwd -1*.

Note: The value of these parameters must be **quoted**, e.g: sshkey="ssh-rsa BBBA1NBzaC1yc2E . . . "

Warning: Some base operational systems might require selinux to be in **permissive** or **disabled** mode so that you can log in the image. This can be achieved by building the image with the `selinux-permissive` element for diskimage-builder or by passing `selinux=0` in the kernel command line. RHEL/CentOS are examples of OSs which this is true.

4.3.36 element-manifest

Writes a manifest file that is the full list of elements that were used to build the image. The file path can be overridden by setting $DIB_ELEMENT_MANIFEST_PATH, and defaults to /etc/dib-manifests/element-manifest.

4.3.37 enable-serial-console

Start getty on active serial consoles.

With ILO and other remote admin environments, having a serial console can be useful for debugging and troubleshooting.

For upstart: If ttyS1 exists, getty will run on that, otherwise on ttyS0.

For systemd: We dynamically start a getty on any active serial port via udev rules.

4.3.38 epel

This element installs the Extra Packages for Enterprise Linux (EPEL) repository GPG key as well as configuration for yum.

Note this element only works with platforms that have EPEL support such as CentOS and RHEL

DIB_EPEL_MIRROR:

Required No

Default None

Description To use a EPEL Yum mirror, set this variable to the mirror URL before running bin/disk-image-create. This URL should point to the directory containing the `5/6/7` directories.

Example `DIB\_EPEL\_MIRROR=http://dl.fedoraproject.org/pub/epel`

DIB_EPEL_DISABLED:

Required No

Default 0

Description To disable the EPEL repo (but leave it available if used with an explicit `--enablerepo`) set this to 1

4.3.39 fedora-minimal

Create a minimal image based on Fedora.

Use of this element will require 'yum' and 'yum-utils' to be installed on Ubuntu and Debian. Nothing additional is needed on Fedora or CentOS. The element will need *python-lzma* everywhere.

Due to a bug in the released version of urlgrabber, on many systems an installation of urlgrabber from git is required. The git repository can be found here: http://yum.baseurl.org/gitweb?p=urlgrabber.git;a=summary

This element sets the `DIB_RELEASE` var to 'fedora'. The release of fedora to be installed can be controlled through the `DIB_RELEASE` variable, which defaults the latest supported release.

4.3.40 fedora

Use Fedora cloud images as the baseline for built disk images. For further details see the redhat-common README.

Environment Variables

DIB_DISTRIBUTION_MIRROR:

Required No

Default None

Description To use a Fedora Yum mirror, set this variable to the mirror URL before running bin/disk-image-create. This URL should point to the directory containing the `releases/updates/development` and `extras` directories.

Example `DIB\_DISTRIBUTION\_MIRROR=http://download.fedoraproject.org/pub/fedora/linux`

4.3.41 Gentoo

Use a Gentoo cloud image as the baseline for built disk images. The images are located in profile specific sub directories:

http://distfiles.gentoo.org/releases/amd64/autobuilds/

As of this writing, only x86_64 images are available.

Notes:

- There are very frequently new automated builds that include changes that happen during the product maintenance. The download directories contain an unversioned name and a versioned name. The unversioned name will always point to the latest image, but will frequently change its content. The versioned one will never change content, but will frequently be deleted and replaced by a newer build with a higher version-release number.
- In order to run the package-installs element you will need to make sure *dev-python/pyyaml* is installed on the host.
- In order to run the vm element you will need to make sure *sys-block/parted* is installed on the host.
- Other profiles can be used by exporting GENTOO_PROFILE with a valid profile. A list of valid profiles follows:

  default/linux/amd64/13.0 default/linux/amd64/13.0/no-multilib hardened/linux/amd64 hardened/linux/amd64/no-multilib

- You can set the GENTOO_PORTAGE_CLEANUP environment variable to true (or anything other than False) to clean up portage from the system and get the image size smaller.
- Gentoo supports many diferent versions of python, in order to select one you may use the *GENTOO_PYTHON_TARGETS* environment variable to select the versions of python you want on your image. The format of this variable is a string as follows *"python2_7 python3_5"*.
- In addition you can select the primary python version you wish to use (that which will be called by running the *python* command. The *GENTOO_PYTHON_ACTIVE_VERSION* is used to set that mapping. The variable contents can be something like *python3.5*.

4.3.42 growroot

Grow the root partition on first boot.

This is only supported on:

- ubuntu trusty or later
- gentoo
- fedora & centos
- suse & opensuse

4.3.43 grub2

This image installs grub2 bootloader on the image, that's necessary for the local boot feature in Ironic to work. This is being made a separated element because usually images have grub2 removed for space reasons and also because they are going to boot from network (PXE boot).

4.3.44 hpdsa

This is the hpdsa element.

This element enables the 'hpdsa' driver to be included in the ramdisk when invoked during ramdisk/image creation.

This driver is required for deploying the HP Proliant Servers Gen9 with Dynamic Smart Array Controllers.

Note: This element supports only Ubuntu image/ramdisk to be updated with the hpdsa driver. It installs hp certificate from https://downloads.linux.hp.com/SDR/hpPublicKey2048_key1.pub. Since HP has released this currently only for trusty, It has been restricted to work only for trusty.

4.3.45 hwburnin

A hardware test ramdisk - exercises the machine RAM and exercises the hard disks

4.3.46 hwdiscovery

A ramdisk to report the hardware of a machine to an inventory service.

This will collect up some basic information about the hardware it boots on:

- CPU cores
- RAM

- Disk
- NIC mac address

This information will then be collated into a JSON document, base64 encoded and passed, via HTTP POST, to a URL that you must specify on the kernel commandline, thus:

HW_DISCOVERY_URL=http://1.2.3.4:56/hw_script.asp

This is currently fairly fragile as there can be a huge variability in the number of disks/NICs in servers and how they are configured.

If other elements wish to inject data into the hardware discovery data, they can - they should be specified before hwdiscovery to the image building script, and they should contain something like this in their init fragment:

_vendor_hwdiscovery_data="$_vendor_hwdiscovery_data "some vendor key" : "some data you care about", "some other vendor key" : "some other data you care about","

Note that you are essentially feeding JSON into the main hwdiscovery JSON.

This will allow any number of vendor specific hwdiscovery elements to chain together their JSON fragments and maintain consistency.

4.3.47 ilo

Ramdisk support for applying HP iLO firmware.

The firmware files are copied in via an extra-data hook: the variable DIB_ILO_FIRMWARE_PATH specifies a directory, and every file in that directory will be unpacked into a same-named directory in the ramdisk (using –unpack=...). If the path is not specified, a diagnostic is output but no error is triggered.

During ramdisk init every found firmware installer will be executed using –silent –log=log The log is displayed after the firmware has executed.

If the firmware exits with status 0 (ok), status 2 (same or older version) or 4 (ilo not detected) a diagnostic message is logged and init proceeds.

Any other status code is treated as an error.

4.3.48 install-bin

Add any files in an element's bin directory to /usr/local/bin in the created image with permissions 0755.

4.3.49 install-static

Copy static files into the built image.

The contents of any `static/` subdirs of elements will be installed into these images at build time using `rsync -lCr`. So to install a file `/etc/boo`, include `static/etc/boo` in your element.

Note: This installs all files with owner and group of root.

4.3.50 install-types

Enable install-types support for elements.

Install types permit elements to be installed from different sources, such as git repositories, distribution packages, or pip. The default install type is 'source' but it can be modified on the disk-image-create command line via the –install-type option. For example you can set:

–install-type=package

to enable package installs by default. Alternately, you can also set DIB_DEFAULT_INSTALLTYPE.

Many elements expose different install types. The different implementations live under *<install-dir-prefix>-<install-type>-install* directories under an element's install.d. The base element enables the chosen install type by symlinking the correct hook scripts under install.d directly. *<install-dir-prefix>* can be a string of alphanumeric and '-' characters, but typically corresponds to the element name.

For example, the nova element would provide:

nova/install.d/nova-package-install/74-nova nova/install.d/nova-source-install/74-nova

The following symlink would be created for the package install type:

install.d/74-nova -> nova-package-install/74-nova

Or, for the source install type:

install.d/74-nova -> nova-source-install/74-nova

All other scripts that exist under install.d for an element will be executed as normal. This allows common install code to live in a script under install.d.

Environment Variables

DIB_INSTALLTYPE_<install_dir_prefix>

Required No

Default source

Description Set the install type for the dir prefix. '-' characters can be replaced with a '_'.

Example `DIB_INSTALLTYPE_simple_init=repo` Sets the simple-init element install type to be repo.

4.3.51 ironic-agent

Builds a ramdisk with ironic-python-agent. More information can be found at: https://git.openstack.org/cgit/openstack/ironic-python-agent/

Beyond installing the ironic-python-agent, this element does the following:

- Installs the `dhcp-all-interfaces` so the node, upon booting, attempts to obtain an IP address on all available network interfaces.
- Disables the `iptables` service on SysV and systemd based systems.
- Disables the `ufw` service on Upstart based systems.
- **Installs packages required for the operation of the ironic-python-agent::** `qemu-utils` `parted` `hdparm` `util-linux` `genisoimage`
- When installing from source, `python-dev` and `gcc` are also installed in order to support source based installation of ironic-python-agent and its dependencies.
- Install the certificate if any, which is set to the environment variable `DIB_IPA_CERT` for validating the authenticity by ironic-python-agent. The certificate can be self-signed certificate or CA certificate.

- Compresses initramfs with command specified in environment variable `DIB_IPA_COMPRESS_CMD`, which is 'gzip' by default. This command should listen for raw data from stdin and write compressed data to stdout. Command can be with arguments.

This element outputs three files:

- `$IMAGE-NAME.initramfs`: The deploy ramdisk file containing the ironic-python-agent (IPA) service.
- `$IMAGE-NAME.kernel`: The kernel binary file.
- `$IMAGE-NAME.vmlinuz`: A hard link pointing to the `$IMAGE-NAME.kernel` file; this is just a backward compatibility layer, please do not rely on this file.

Note: The package based install currently only enables the service when using the systemd init system. This can easily be changed if there is an agent package which includes upstart or sysv packaging.

Note: Using the ramdisk will require at least 1.5GB of ram

4.3.52 iso

Generates a bootable ISO image from the kernel/ramdisk generated by the elements `baremetal`, `ironic-agent` or `ramdisk`. It uses isolinux to boot on BIOS machines and grub to boot on EFI machines.

This element has been tested on the following distro(s): * ubuntu * fedora * debian

NOTE: For other distros, please make sure the `isolinux.bin` file exists at `/usr/lib/syslinux/isolinux.bin`.

baremetal element

When used with `baremetal` element, this generates a bootable ISO image named `<image-name>-boot.iso` booting the generated kernel and ramdisk. It also automatically appends kernel command-line argument 'root=UUID=<uuid-of-the-root-partition>'. Any more kernel command-line arguments required may be provided by specifying them in `DIB_BOOT_ISO_KERNEL_CMDLINE_ARGS`.

NOTE: It uses pre-built efiboot.img by default to work for UEFI machines. This is because of a bug in latest version of grub[1]. The user may choose to avoid using pre-built binary and build efiboot.img on their own machine by setting the environment variable DIB_UEFI_ISO_BUILD_EFIBOOT to 1 (this might work only on certain versions of grub). The current efiboot.img was generated by the method build_efiboot_img() in 100-build-iso on Ubuntu 13.10 with grub 2.00-19ubuntu2.1.

ramdisk element

When used with `ramdisk` element, this generates a bootable ISO image named `<image-name>.iso` booting the generated kernel and ramdisk. It also automatically appends kernel command-line argument 'boot_method=vmedia' which is required for Ironic drivers `iscsi_ilo`.

ironic-agent element

When used with `ironic-agent` element, this generates a bootable ISO image named `<image-name>.iso` which boots the agent kernel and agent ramdisk.

REFERENCES

[1] https://bugs.launchpad.net/ubuntu/+source/grub2/+bug/1378658

4.3.53 local-config

Copies local user settings such as .ssh/authorized_keys and $http_proxy into the image.

Environment Variables

DIB_LOCAL_CONFIG_USERNAME

Required No

Default root

Description Username used when installing .ssh/authorized_keys.

4.3.54 manifests

An framework for saving manifest information generated during the build for later inspection. Manifests are kept in the final image and also copied to the build area post-image creation.

Elements that wish to save any form of manifest should depend on this element and can save their data to into the `DIB_MANIFEST_IMAGE_DIR` (which defaults to `/etc/dib-manifests`). Note this is created in `extra-data.d` rather than `pre-install.d` to allow the `source-repositories` element to make use of it

The manifests are copied to `DIB_MANIFEST_SAVE_DIR`, which defaults to `${IMAGE_NAME}.d/`, resulting in the manifests being available as `${IMAGE_NAME}.d/dib-manifests` by default after the build.

Extra status

This element will also add the files `dib_environment` and `dib_arguments` to the manifest recording the `diskimage-builder` specific environment (`DIB_*` variables) and command-line arguments respectively.

4.3.55 mellanox

Force support for mellanox hardware

4.3.56 modprobe-blacklist

Blacklist specific modules using modprobe.d/blacklist.conf.

In order to use set DIB_MODPROBE_BLACKLIST to the name of your module. To disable multiple modules you can set DIB_MODPROBE_BLACKLIST to a list of string separated by spaces.

Example:

export DIB_MODPROBE_BLACKLIST=”igb”

4.3.57 no-final-image

This is a noop element which can be used to indicate to diskimage-builder that it should not bother creating a final image out of the generated filesystem. It is useful in cases where an element handles all of the image building itself, such as ironic-agent or Docker images. In those cases the final image normally generated by diskimage-builder is not the desired output, so there's no reason to spend time creating it.

Elements that wish to behave this way should include this element in their element-deps file.

4.3.58 oat-client

This element installs oat-client on the image, that's necessary for trusted boot feature in Ironic to work.

Intel TXT will measure BIOS, Option Rom and Kernel/Ramdisk during trusted boot, the oat-client will securely fetch the hash values from TPM.

Note: This element only works on Fedora.

Put *fedora-oat.repo* into */etc/yum.repos.d/*:

```
export DIB_YUM_REPO_CONF=/etc/yum.repos.d/fedora-oat.repo
```

Note: OAT Repo is lack of a GPG signature check on packages, which can be tracked on: https://github.com/OpenAttestation/OpenAttestation/issues/26

4.3.59 openssh-server

This element ensures that openssh server is installed and enabled during boot.

Note

Most cloud images come with the openssh server service installed and enabled during boot. However, certain cloud images, especially those created by the *-minimal elements may not have it installed or enabled. In these cases, using this element may be helpful to ensure your image will accessible via SSH. It's usually helpful to combine this element with others such as the *runtime-ssh-host-keys*.

4.3.60 openstack-ci-mirrors

This element contains various settings to setup mirrors for openstack ci gate testing in a generic fashion. It is intended to be used as a dependency of testing elements that run in the gate. It should do nothing outside that environment.

4.3.61 opensuse-minimal

This element will build a minimal openSUSE image. It requires 'zypper' to be installed on the host.

These images should be considered experimental. There are currently only x86_64 images.

Environment Variables

DIB_RELEASE

Required No

Default 42.3

Description Set the desired openSUSE release.

4.3.62 opensuse

Use an openSUSE cloud image as the baseline for built disk images. The images are located in distribution specific sub directories under

http://download.opensuse.org/repositories/Cloud:/Images:/

These images should be considered experimental. There are currently only x86_64 images.

Environment Variables

DIB_RELEASE

Required No

Default 42.3

Description Set the desired openSUSE release.

DIB_CLOUD_IMAGES

Required No

Default http://download.opensuse.org/repositories/Cloud:/Images:/(openSUSE|Leap)_${DIB_RELEASE}

Description Set the desired URL to fetch the images from.

Notes:

- There are very frequently new automated builds that include changes that happen during the product maintenance. The download directories contain an unversioned name and a versioned name. The unversioned name will always point to the latest image, but will frequently change its content. The versioned one will never change content, but will frequently be deleted and replaced by a newer build with a higher version-release number.

4.3.63 package-installs

The package-installs element allows for a declarative method of installing and uninstalling packages for an image build. This is done by creating a package-installs.yaml or package-installs.json file in the element directory.

In order to work on Gentoo hosts you will need to manually install *dev-python/pyyaml*.

example `package-installs.yaml`

```
libxml2:
grub2:
  phase: pre-install.d
networkmanager:
  uninstall: True
os-collect-config:
  installtype: source
```

```
linux-image-amd64:
  arch: amd64
dmidecode:
  not-arch: ppc64, ppc64le
lshw:
  arch: ppc64, ppc64le
python-dev:
  dib_python_version: 2
python3-dev:
  dib_python_version: 3
```

example package-installs.json

```
{
"libxml2": null,
"grub2": {"phase": "pre-install.d"},
"networkmanager": {"uninstall": true}
"os-collect-config": {"installtype": "source"}
}
```

Setting phase, uninstall, or installtype properties for a package overrides the following default values:

```
phase: install.d
uninstall: False
installtype: * (Install package for all installtypes)
arch: * (Install package for all architectures)
dib_python_version: (2 or 3 depending on DIB_PYTHON_VERSION, see dib-python)
```

Setting the installtype property causes the package only to be installed if the specified installtype would be used for the element. See the diskimage-builder docs for more information on installtypes.

The `arch` property is a comma-separated list of architectures to install for. The `not-arch` is a comma-separated list of architectures the package should be excluded from. Either `arch` or `not-arch` can be given for one package - not both. See documentation about the ARCH variable for more information.

DEPRECATED: Adding a file under your elements pre-install.d, install.d, or post-install.d directories called package-installs-<element-name> will cause the list of packages in that file to be installed at the beginning of the respective phase. If the package name in the file starts with a "-", then that package will be removed at the end of the install.d phase.

Using post-install.d for cleanup

Package removal is done in post-install.d at level 95. If you a running cleanup functions before this, you need to be careful not to clean out any temporary files relied upon by this element. For this reason, generally post-install cleanup functions should occupy the higher levels between 96 and 99.

4.3.64 pip-and-virtualenv

This element installs pip and virtualenv in the image.

Package install

If the package installtype is used then these programs are installed from distribution packages. In this case, `pip` and `virtualenv` will be installed *only* for the python version identified by `dib-python` (i.e. the default python for the platform).

Distribution packages have worked out name-spacing such that only python2 or python3 owns common scripts like `/usr/bin/pip` (on most platforms, `pip` refers to python2 pip, and `pip3` refers to python3 pip, although some may choose the reverse).

To install pip and virtualenv from package:

```
export DIB_INSTALLTYPE_pip_and_virtualenv=package
```

Source install

Source install is the default. If the source installtype is used, `pip` and `virtualenv` are installed from the latest upstream releases.

Source installs from these tools are not name-spaced. It is inconsistent across platforms if the first or last install gets to own common scripts like `/usr/bin/pip` and `virtualenv`.

To avoid inconsistency, we firstly install the packaged python 2 **and** 3 versions of `pip` and `virtualenv`. This prevents a later install of these distribution packages conflicting with the source install. We then overwrite `pip` and `virtualenv` via `get-pip.py` and `pip` respectively.

The system will be left in the following state:

- `/usr/bin/pip` : python2 pip
- `/usr/bin/pip2` : python2 pip (same as prior)
- `/usr/bin/pip3` : python3 pip
- `/usr/bin/virtualenv` : python2 virtualenv

(note python3 `virtualenv` script is *not* installed, see below)

Source install is supported on limited platforms. See the code, but this includes Ubuntu and RedHat platforms.

Using the tools

Due to the essentially unsolvable problem of "who owns the script", it is recommended to *not* call `pip` or `virtualenv` directly. You can directly call them with the `-m` argument to the python interpreter you wish to install with.

For example, to create a python3 environment do:

```
# python3 -m virtualenv myenv
# myenv/bin/pip install mytool
```

To install a python2 tool from pip:

```
# python2 -m pip install mytool
```

In this way, you can always know which interpreter is being used (and affected by) the call.

Ordering

Any element that uses these commands must be designated as 05-* or higher to ensure that they are first installed.

4.3.65 pip-cache

Use a cache for pip

Using a download cache speeds up image builds.

Including this element in an image build causes $HOME/.cache/image-create/pip to be bind mounted as /tmp/pip inside the image build chroot. The $PIP_DOWNLOAD_CACHE environment variable is then defined as /tmp/pip, which causes pip to cache all downloads to the defined location.

Note that pip and its use of $PIP_DOWNLOAD_CACHE is not concurrency safe. Running multiple instances of diskimage-builder concurrently can cause issues. Therefore, it is advised to only have one instance of diskimage-builder that includes the pip-cache element running at a time.

The pip concurrency issue is being tracked upstream at https://github.com/pypa/pip/issues/1141

4.3.66 pkg-map

Map package names to distro specific packages.

Provides the following:

- bin/pkg-map:

```
usage: pkg-map [-h] [--element ELEMENT] [--distro DISTRO]

Translate package name to distro specific name.

optional arguments:
  -h, --help         show this help message and exit
  --element ELEMENT  The element (namespace) to use for translation.
  --distro DISTRO    The distro name to use for translation. Defaults to
                     DISTRO_NAME
  --release RELEASE  The release to use for translation.  Defaults to
                     DIB_RELEASE
```

- Any element may create its own pkg-map JSON config file using the one of 4 sections for the release/distro/family/ and or default. The family is set automatically within pkg-map based on the supplied distro name. Families include:
  - redhat: includes centos, fedora, and rhel distros
  - debian: includes debian and ubuntu distros
  - suse: includes the opensuse distro

  The release is a specification of distro; i.e. the distro and release must mach for a translation.

  The most specific section takes priority.

  An empty package list can be provided.

  Example for Nova and Glance (NOTE: using fictitious package names for Fedora and package mapping for suse family to provide a good example!)

  Example format:

```
{
  "release": {
    "fedora": {
      "23": {
```

```
        "nova_package": "foo" "bar"
      }
    }
  },
  "distro": {
    "fedora": {
      "nova_package": "openstack-compute",
      "glance_package": "openstack-image"
    }
  },
  "family": {
    "redhat": {
      "nova_package": "openstack-nova",
      "glance_package": "openstack-glance"
    },
    "suse": {
      "nova_package": ""
    }
  },
  "default": {
    "nova_package": "nova",
    "glance_package": "glance"
  }
}
```

Example commands using this format:

pkg-map –element nova-compute –distro fedora nova_package

Returns: openstack-compute

pkg-map –element nova-compute –distro rhel nova_package

Returns: openstack-nova

pkg-map –element nova-compute –distro ubuntu nova_package

Returns: nova

pkg-map –element nova-compute –distro opensuse nova_package

Returns:

- This output can be used to filter what other tools actually install (install-packages can be modified to use this for example)
- Individual pkg-map files live within each element. For example if you are created an Apache element your pkg-map JSON file should be created at elements/apache/pkg-map.

4.3.67 posix

- This element installs packages to ensure that the resulting image has binaries necessary to meet the requirements of POSIX, laid out in the following URL:
  - http://pubs.opengroup.org/onlinepubs/9699919799/idx/utilities.html
- This has been tested to work on Ubuntu, Debian, and CentOS, although should work on Red Hat Enterprise Linux.
- To add support for other distros please consult the URL for binaries, then add the providing packages to pkg-map.

4.3.68 proliant-tools

- This element can be used when building ironic-agent ramdisk. It enables ironic-agent ramdisk to do in-band cleaning operations specific to HPE ProLiant hardware.
- Works with ubuntu and fedora distributions (on which ironic-agent element is supported).
- Currently the following utilities are installed:
  - proliantutils - This module registers an ironic-python-agent hardware manager for HPE ProLiant hardware, which implements in-band cleaning steps. The latest version of `proliantutils` available is installed. This python module is released with Apache license.
  - HPE Smart Storage Administrator (HPE SSA) CLI for Linux 64-bit - This utility is used by `proliantutils` library above for doing in-band RAID configuration on HPE ProLiant hardware. Currently installed version is 2.60. Newer version of `ssacli` when available, may be installed to the ramdisk by using the environment variable `DIB_SSACLI_URL`. `DIB_SSACLI_URL` should contain the HTTP(S) URL for downloading the RPM package for `ssacli` utility. The old environmental variable `DIB_HPSSACLI_URL`,a HTTP(S) URL for downloading the RPM package for `hpssacli` utility, is deprecated. The `hpssacli` utility is not supported anymore, use `ssacli` instead for the same functionality. Availability of newer versions can be in the Revision History in the above link. This utility is closed source and is released with HPE End User License Agreement – Enterprise Version.

4.3.69 pypi

Inject a PyPI mirror

Use a custom PyPI mirror to build images. The default is to bind mount one from ~/.cache/image-create/pypi/mirror into the build environment as mirror URL file:///tmp/pypi. The element temporarily overwrites /root/.pip.conf and .pydistutils.cfg to use it.

When online, the official pypi.python.org pypi index is supplied as an extra-url, so uncached dependencies will still be available. When offline, only the mirror is used - be warned that a stale mirror will cause build failures. To disable the pypi.python.org index without using –offline (e.g. when working behind a corporate firewall that prohibits pypi.python.org) set DIB_NO_PYPI_PIP to any non-empty value.

To use an arbitrary mirror set DIB_PYPI_MIRROR_URL=http[s]://somevalue/

Additional mirrors can be added by exporting DIB_PYPI_MIRROR_URL_1=... etc. Only the one mirror can be used by easy-install, but since wheels need to be in the first mirror to be used, the last listed mirror is used as the pydistutils index. NB: The sort order for these variables is a simple string sort - if you have more than 9 additional mirrors, some care will be needed.

You can also set the number of retries that occur on failure by setting the DIB_PIP_RETRIES environment variable. If setting fallback pip mirrors you typically want to set this to 0 to prevent the need to fail multiple times before falling back.

A typical use of this element is thus: export DIB_PYPI_MIRROR_URL=http://site/pypi/Ubuntu-13.10 export DIB_PYPI_MIRROR_URL_1=http://site/pypi/ export DIB_PYPI_MIRROR_URL_2=file:///tmp/pypi export DIB_PIP_RETRIES=0

[devpi-server](https://git.openstack.org/cgit/openstack-infra/pypi-mirro://pypi.python.org/pypi/devpi-server) can be useful in making a partial PyPI mirror suitable for building images. For instance:

- pip install -U devpi
- devpi-server quickstart
- devpi use http://machinename:3141

- Re-export your variables to point at the new mirror:

  export DIB_PYPI_MIRROR_URL=http://machinename:3141/ unset DIB_PYPI__MIRROR_URL_1 unset DIB_PYPI__MIRROR_URL_2

The next time packages are installed, they'll be cached on the local devpi server; subsequent runs pointed at the same mirror will use the local cache if the upstream can't be contacted.

Note that this process only has the server running temporarily; see [Quickstart: Permanent install on server/laptop](http://doc.devpi.net/latest/quickstart-server.html) guide from the devpi developers for more information on a more permanent setup.

4.3.70 python-brickclient

- This element is aimed for providing cinder local attach/detach functionality.
- Currently the feature has a dependency on a known bug https://launchpad.net/bugs/1623549, which has been resolved and will be part of the upstream with the next release of `python-brick-cinderclient-ext`. Note: Current version of `python-brick-cinderclient-ext` i.e. 0.2.0 requires and update to be made in Line32 for `/usr/share/python-brickclient/venv/lib/python2.7/site-packages/brick_cinderclient_ext/__init__.py`: update `brick-python-cinderclient-ext` to `python-brick-cinderclient-ext`.

Usage

Pass the below shell script to parameter `user-data` and set `config-drive=true` at the time of provisioning the node via nova-boot to make cinder local attach/detach commands talk to your cloud controller.

```
  #!/bin/bash
  FILE="/etc/bash.bashrc"
  [ ! -f "$FILE" ] && touch "$FILE"
  echo 'export OS_AUTH_URL="http://<controller_ip>:5000/v2.0"' >> "$FILE"
  echo 'export OS_PASSWORD="password"'  >> "$FILE"
  echo 'export OS_USERNAME="demo"' >> "$FILE"
  echo 'export OS_TENANT_NAME="demo"'  >> "$FILE"
  echo 'export OS_PROJECT_NAME="demo"' >> "$FILE"
  exec bash

To attach: ``/usr/share/python-brickclient/venv/bin/cinder local-attach <volume_id>``
To detach: ``/usr/share/python-brickclient/venv/bin/cinder local-detach <volume_id>``
```

Alternatively, the same action can be completed manually at the node which does not require setting up of config drive such as:

```
/usr/share/python-brickclient/venv/bin/cinder \
  --os-username demo --os-password password \
  --os-tenant-name demo --os-project-name demo \
  --os-auth-url=http://<controller_ip>:5000/v2.0 local-attach <volume_id>
```

4.3.71 ramdisk-base

Shared functionality required by all of the different ramdisk elements.

4.3.72 ramdisk

This is the ramdisk element.

Almost any user building a ramdisk will want to include this in their build, as it triggers many of the vital functionality from the basic diskimage-builder libraries (such as init script aggregation, busybox population, etc).

An example of when one might want to use this toolchain to build a ramdisk would be the initial deployment of baremetal nodes in a TripleO setup. Various tools and scripts need to be injected into a ramdisk that will fetch and apply a machine image to local disks. That tooling/scripting customisation can be easily applied in a repeatable and automatable way, using this element.

NOTE: ramdisks require 1GB minimum memory on the machines they are booting.

See the top-level README.md of the project, for more information about the mechanisms available to a ramdisk element.

4.3.73 rax-nova-agent

Images for Rackspace Cloud currently require nova-agent to get networking information.

Many of the things here are adapted from:

https://developer.rackspace.com/blog/bootstrap-your-qcow-images-for-the-rackspace-public-cloud/

4.3.74 redhat-common

Image installation steps common to RHEL, CentOS, and Fedora.

Requirements:

If used to build an image form a cloud image compress with xz (the default in centos), this element uses "unxz" to decompress the image. Depending on your distro you may need to install either the xz or xz-utils package.

Environment Variables

DIB_LOCAL_IMAGE

Required No

Default None

Description Use the local path of a qcow2 cloud image. This is useful in that you can use a customized or previously built cloud image from diskimage-builder as input. The cloud image does not have to have been built by diskimage-builder. It should be a full disk image, not just a filesystem image.

Example `DIB_LOCAL_IMAGE=rhel-guest-image-7.1-20150224.0.x86_64.qcow2`

DIB_DISABLE_KERNEL_CLEANUP

Required No

Default 0

Description Specify if kernel needs to be cleaned up or not. When set to true, the bits that cleanup old kernels will not be executed.

Example DIB_DISABLE_KERNEL_CLEANUP=1

4.3.75 rhel-common

This element contains the common installation steps between RHEL os releases.

RHEL Registration

This element provides functionality for registering RHEL images during the image build process with the disk-image-create script from diskimage-builder. The RHEL image will register itself with either the hosted Red Hat Customer Portal or Satellite to enable software installation from official repositories. After the end of the image creation process, the image will unregister itself so an entitlement will not be decremented from the account.

SECURITY WARNING:

While the image building workflow will allow you to register with a username and password combination, that feature is deprecated in the boot process via Heat as it will expose your username and password in clear text for anyone that has rights to run heat stack-show. A compromised username and password can be used to login to the Red Hat Customer Portal or an instance of Satellite. An activation key can only be used for registration purposes using the subscription-manager command line tool and is considered a lower security risk.

IMPORTANT NOTE:

The 00-rhsm script is specific to RHEL6. If you use the REG_ variables to use with RHEL7, you do not need to set any DIB_RHSM variables. The scripts named with "rhel-registration" have not been developed or tested for RHEL6. For information on building RHEL6 images, please see the rhel element README.

Environment Variables For Image Creation

The following environment variables are used for registering a RHEL instance with either the Red Hat Customer Portal or Satellite 6.

REG_ACTIVATION_KEY Attaches existing subscriptions as part of the registration process. The subscriptions are pre-assigned by a vendor or by a systems administrator using Subscription Asset Manager.

REG_AUTO_ATTACH Automatically attaches the best-matched compatible subscription. This is good for automated setup operations, since the system can be configured in a single step.

REG_BASE_URL Gives the hostname of the content delivery server to use to receive updates. Both Customer Portal Subscription Management and Subscription Asset Manager use Red Hat's hosted content delivery services, with the URL https://cdn.redhat.com. Since Satellite 6 hosts its own content, the URL must be used for systems registered with Satellite 6.

REG_ENVIRONMENT Registers the system to an environment within an organization.

REG_FORCE Registers the system even if it is already registered. Normally, any register operations will fail if the machine is already registered.

REG_HALT_UNREGISTER At the end of the image build process, the element runs a cleanup script that will unregister it from the system it registered with. There are some cases when building an image where you may want to stop this from happening so you can verify the registration or to build a one off-image where the boot-time registration will not be enabled. Set this value to '1' to stop the unregistration process.

REG_MACHINE_NAME Sets the name of the system to be registered. This defaults to be the same as the hostname.

REG_METHOD Sets the method of registration. Use "portal" to register a system with the Red Hat Customer Portal. Use "satellite" to register a system with Red Hat Satellite 6. Use "disable" to skip the registration process.

REG_ORG Gives the organization to which to join the system.

REG_POOL_ID The pool ID is listed with the product subscription information, which is available from running the list subcommand of subscription-manager.

REG_PASSWORD Gives the password for the user account.

REG_RELEASE Sets the operating system minor release to use for subscriptions for the system. Products and updates are limited to that specific minor release version. This is used only used with the REG_AUTO_ATTACH option. Possible values for this include 5Server, 5.7, 5.8, 5.9, 5.10, 6.1,... 6.6, 7.0. It will change over time as new releases come out. There are also variants 6Server, 6Client, 6Workstation, 7Server, etc.

REG_REPOS A single string representing a list of repository names separated by a comma (No spaces). Each of the repositories in this string are enabled through subscription manager. Once you've attached a subscription, you can find available repositories by running subscription-manager repos –list.

REG_SERVER_URL Gives the hostname of the subscription service to use. The default is for Customer Portal Subscription Management, subscription.rhn.redhat.com. If this option is not used, the system is registered with Customer Portal Subscription Management.

REG_SERVICE_LEVEL Sets the service level to use for subscriptions on that machine. This is only used with the REG_AUTO_ATTACH option.

REG_USER Gives the content server user account name.

REG_TYPE Sets what type of consumer is being registered. The default is system, which is applicable to both physical systems and virtual guests. Other types include hypervisor for virtual hosts, person, domain, rhui, and candlepin for some subscription management applications.

Image Build Registration Examples

To register with Satellite 6, a common example would be to set the following variables:

REG_SAT_URL='http://my-sat06.server.org' REG_ORG='tripleo' REG_ENV='Library' REG_USER='tripleo' REG_PASSWORD='tripleo' REG_METHOD=satellite

To register with the Red Hat Customer Portal, a common example would be to set the following variables:

REG_REPOS='rhel-7-server-optional-rpms,rhel-7-server-extras-rpms' REG_AUTO_ATTACH=true REG_USER='tripleo' REG_PASSWORD='tripleo' REG_METHOD=portal

Configuration

Heat metadata can be used to configure the rhel-common element.

rh_registration:

activation_key: # Attaches existing subscriptions as part of the registration # process. The subscriptions are pre-assigned by a vendor or by # a systems administrator using Subscription Asset Manager.

auto_attach: 'true' # Automatically attaches the best-matched compatible subscription. # This is good for automated setup operations, since the system can # be configured in a single step.

base_url: # Gives the hostname of the content delivery server to use to # receive updates. Both Customer Portal Subscription Management # and Subscription Asset Manager use Red Hat's hosted content # delivery services, with the URL https://cdn.redhat.com. Since # Satellite 6 hosts its own content, the URL must be used for # systems registered with Satellite 6.

environment: # Registers the system to an environment within an organization.

force: # Registers the system even if it is already registered. Normally, # any register operations will fail if the machine is already # registered.

machine_name: # Sets the name of the system to be registered. This defaults to be # the same as the hostname.

org: # Gives the organization to which to join the system.

password: # DEPRECATED # Gives the password for the user account.

release: # Sets the operating system minor release to use for subscriptions # for the system. Products and updates are limited to that specific # minor release version. This is only used with the auto_attach # option.

repos: # A single string representing a list of repository names separated by a # comma (No spaces). Each of the repositories in this string are enabled # through subscription manager.

satellite_url: # The url of the Satellite instance to register with. Required for # Satellite registration.

server_url: # Gives the hostname of the subscription service to use. The default # is for Customer Portal Subscription Management, # subscription.rhn.redhat.com. If this option is not used, the system # is registered with Customer Portal Subscription Management.

service_level: # Sets the service level to use for subscriptions on that machine. # This is only used with the auto_attach option.

user: # DEPRECATED # Gives the content server user account name.

type: # Sets what type of consumer is being registered. The default is # "system", which is applicable to both physical systems and virtual # guests. Other types include "hypervisor" for virtual hosts, # "person", "domain", "rhui", and "candlepin" for some subscription # management applications.

method: # Sets the method of registration. Use "portal" to register a # system with the Red Hat Customer Portal. Use "satellite" to # register a system with Red Hat Satellite 6. Use "disable" to # skip the registration process.

Configuration Registration Examples

To register with Satellite 6, a common example would be to use the following metadata:

```
{
    "rh_registration":{
        "satellite_url": "http://my-sat06.server.org",
        "org": "tripleo",
        "environment": "Library",
        "activation_key": "my-key-SQQkh4",
        "method":"satellite",
        "repos": "rhel-ha-for-rhel-7-server-rpms"
    }
}
```

To register with the Red Hat Customer Portal, a common example would be to use the following metadata:

```
{
    "rh_registration":{
        "repos":"rhel-7-server-optional-rpms,rhel-7-server-extras-rpms",
        "auto_attach":true,
        "activation_key": "my-key-SQQkh4",
```

```
        "org": "5643002",
        "method":"portal"
    }
}
```

4.3.76 rhel7

Use RHEL 7 cloud images as the baseline for built disk images.

Because RHEL 7 base images are not publicly available, it is necessary to first download the RHEL 7 cloud image from the Red Hat Customer Portal and pass the path to the resulting file to disk-image-create as the `DIB_LOCAL_IMAGE` environment variable.

The cloud image can be found at (login required): https://access.redhat.com/downloads/content/69/ver=/rhel—7/7.1/x86_64/product-downloads

Then before running the image build, define DIB_LOCAL_IMAGE (replace the file name with the one downloaded, if it differs from the example):

```
export DIB_LOCAL_IMAGE=rhel-guest-image-7.1-20150224.0.x86_64.qcow2
```

The downloaded file will then be used as the basis for any subsequent image builds.

For further details about building RHEL 7 images, see the rhel-common and redhat-common element README files.

Environment Variables

DIB_LOCAL_IMAGE

Required Yes

Default None

Description The RHEL 7 base image you have downloaded. See the element description above for more details.

Example `DIB_LOCAL_IMAGE=/tmp/rhel7-cloud.qcow2`

4.3.77 runtime-ssh-host-keys

An element to generate SSH host keys on first boot.

Since ssh key generation is not yet common to all operating systems, we need to create a DIB element to manage this. We force the removal of the SSH host keys, then add init scripts to generate them on first boot.

This element currently supports Debian and Ubuntu (both systemd and upstart).

4.3.78 select-boot-kernel-initrd

A helper script to get the kernel and initrd image.

It uses the function select_boot_kernel_initrd from the library img-functions to find the newest kernel and ramdisk in the image, and returns them as a concatenated string separating the values with a colon (:).

4.3.79 selinux-permissive

Puts selinux into permissive mode by writing SELINUX=permissive to /etc/selinux/config

Enable this element when debugging selinux issues.

4.3.80 simple-init

Basic network and system configuration that can't be done until boot

Unfortunately, as much as we'd like to bake it in to an image, we can't know in advance how many network devices will be present, nor if DHCP is present in the host cloud. Additionally, in environments where cloud-init is not used, there are a couple of small things, like mounting config-drive and pulling ssh keys from it, that need to be done at boot time.

Autodetect network interfaces during boot and configure them

The rationale for this is that we are likely to require multiple network interfaces for use cases such as baremetal and there is no way to know ahead of time which one is which, so we will simply run a DHCP client on all interfaces with real MAC addresses (except lo) that are visible on the first boot.

The script */usr/local/sbin/simple-init.sh* will be called early in each boot and will scan available network interfaces and ensure they are configured properly before networking services are started.

Processing startup information from config-drive

On most systems, the DHCP approach desribed above is fine. But in some clouds, such as Rackspace Public cloud, there is no DHCP. Instead, there is static network config via *config-drive*. *simple-init* will happily call *glean* which will do nothing if static network information is not there.

Finally, glean will handle ssh-keypair-injection from config drive if cloud-init is not installed.

Chosing glean installation source

By default glean is installed using pip using the latest release on pypi. It is also possible to install glean from a specified git repository location. This is useful for debugging and testing new glean changes for example. To do this you need to set these variables:

```
DIB_INSTALLTYPE_simple_init=repo
DIB_REPOLOCATION_glean=/path/to/glean/repo
DIB_REPOREF_glean=name_of_git_ref
```

For example to test glean change 364516 do:

```
git clone https://git.openstack.org/openstack-infra/glean /tmp/glean
cd /tmp/glean
git review -d 364516
git checkout -b my-test-ref
```

Then set your DIB env vars like this before running DIB:

```
DIB_INSTALLTYPE_simple_init=repo
DIB_REPOLOCATION_glean=/tmp/glean
DIB_REPOREF_glean=my-test-ref
```

4.3.81 source-repositories

With this element other elements can register their installation source by placing their details in the file `source-repository-*`.

source-repository-* file format

The plain text file format is space separated and has four mandatory fields optionally followed by fields which are type dependent:

```
<name> <type> <destination> <location> [<ref>]
```

`name` Identifier for the source repository. Should match the file suffix.

`type` Format of the source. Either `git`, `tar`, `package` or `file`.

`destination` Base path to place sources.

`location` Resource to fetch sources from. For `git` the location is cloned. For `tar` it is extracted.

`ref` (optional). Meaning depends on the `type`: `file`: unused/ignored.

`git`: a git reference to fetch. A value of "`*`" prunes and fetches all heads and tags. Defaults to `master` if not specified.

`tar`:

"`.`" extracts the entire contents of the tarball.

"`*`" extracts the contents within all its subdirectories.

A subdirectory path may be used to extract only its contents.

A specific file path within the archive is **not supported**.

The lines in the source-repository scripts are eval'd, so they may contain environment variables.

The `package` type indicates the element should install from packages onto the root filesystem of the image build during the `install.d` phase. If the element provides an <element-name>-package-install directory, symlinks will be created for those scripts instead.

`git` and `tar` are treated as source installs. If the element provides an <element-name>-source-install directory under it's `install.d` hook directory, symlinks to the scripts in that directory will be created under `install.d` for the image build.

For example, the nova element would provide:

```
nova/install.d/nova-package-install/74-nova
nova/install.d/nova-source-install/74-nova
```

source-repositories will create the following symlink for the package install type:

```
install.d/74-nova -> nova-package-install/74-nova
```

Or, for the source install type:

```
install.d/74-nova -> nova-source-install/74-nova
```

All other scripts that exist under `install.d` for an element will be executed as normal. This allows common install code to live in a script outside of <element-name>-package-install or <element-name>-source-install.

If multiple elements register a source location with the same <destination> then source-repositories will exit with an error. Care should therefore be taken to only use elements together that download source to different locations.

The repository paths built into the image are stored in etc/dib-source-repositories, one repository per line. This permits later review of the repositories (by users or by other elements).

The repository names and types are written to an environment.d hook script at 01-source-repositories-environment. This allows later hook scripts during the install.d phase to know which install type to use for the element.

An example of an element "custom-element" that wants to retrieve the ironic source from git and pbr from a tarball would be:

Element file: elements/custom-element/source-repository-ironic:

```
ironic git /usr/local/ironic git://git.openstack.org/openstack/ironic.git
```

File : elements/custom-element/source-repository-pbr:

```
pbr tar /usr/local/pbr http://tarballs.openstack.org/pbr/pbr-master.tar.gz .
```

diskimage-builder will then retrieve the sources specified and place them at the directory `<destination>`.

Override per source

A number of environment variables can be set by the process calling diskimage-builder which can change the details registered by the element, these are:

`DIB_REPOTYPE_<name>` : change the registered type

`DIB_REPOLOCATION_<name>` : change the registered location

`DIB_REPOREF_<name>` : change the registered reference

For example if you would like diskimage-builder to get ironic from a local mirror you would override the `<location>` field and could set:

```
DIB_REPOLOCATION_ironic=git://localgitserver/ironic.git
```

As you can see above, the <name> of the repo is used in several bash variables. In order to make this syntactically feasible, any characters not in the set [A-Za-z0-9_] will be converted to _

For instance, a repository named "diskimage-builder" would set a variable called "DIB_REPOTYPE_diskimage_builder"

Alternatively if you would like to use the keystone element and build an image with keystone from a stable branch *stable/grizzly* then you would set:

```
DIB_REPOREF_keystone=stable/grizzly
```

If you wish to build an image using code from a Gerrit review, you can set `DIB_REPOLOCATION_<name>` and `DIB_REPOREF_<name>` to the values given by Gerrit in the fetch/pull section of a review. For example, setting up Nova with change 61972 at patchset 8:

```
DIB_REPOLOCATION_nova=https://review.openstack.org/openstack/nova
DIB_REPOREF_nova=refs/changes/72/61972/8
```

Alternate behaviors

Override git remote

The base url for all git repositories can be set by use of:

DIB_GITREPOBASE

So setting DIB_GITREPOBASE=https://github.com/ when the repo location is set to http://git.openstack.org/openstack/nova.git will result in use of the https://github.com/openstack/nova.git repository instead.

Disable external fetches

When doing image builds in environments where external resources are not allowed, it is possible to disable fetching of all source repositories by including an element in the image that sets NO_SOURCE_REPOSITORIES=1 in an environment.d script.

4.3.82 stable-interface-names

Does the following:

- Enables stable network interface naming (em1, em2, etc) by installing the biosdevname and removes any symlinks which may disable udev rules, etc.

4.3.83 svc-map

Map service names to distro specific services.

Provides the following:

- bin/svc-map

  usage: svc-map [-h] SERVICE

  Translate service name to distro specific name.

  optional arguments:

  -h, --help show this help message and exit

- Any element may create its own svc-map YAML config file using the one of 3 sections for the distro/family/ and or default. The family is set automatically within svc-map based on the supplied distro name. Families include:
  - redhat: includes centos, fedora, and rhel distros
  - debian: includes debian and ubuntu distros
  - suse: includes the opensuse distro

  The most specific section takes priority. Example for Nova and Glance (NOTE: default is using the common value for redhat and suse families)

  The key used for the service name should always be the same name used for the source installation of the service. The svc-map script will check for the source name against systemd and upstart and return that name if it exists instead of the mapped name.

  Example format for Nova:

```
nova-api:
  default: openstack-nova-api
  debian: nova-api
nova-cert:
  default: openstack-nova-cert
  debian:  nova-cert
nova-compute:
```

```
  default: openstack-nova-compute
  debian: nova-compute
nova-conductor:
  default: openstack-nova-conductor
  debian: nova-conductor
nova-consoleauth:
  default: openstack-nova-console
  debian: nova-console
```

Example format for Glance:

```
glance-api:
  debian: glance-api
  default: openstack-glance-api
glance-reg:
  debian: glance-reg
  default: openstack-glance-registry
```

If the distro is of the debian family the combined services file would be:

```
nova-cert: nova-cert
nova-compute: nova-compute
glance-api: glance-api
nova-conductor: nova-conductor
nova-api: nova-api
glance-reg: glance-reg
nova-consoleauth: nova-console
```

If the distro is of the suse or redhat families the combined services file would be:

```
nova-cert: openstack-nova-cert
nova-compute: openstack-nova-compute
glance-reg: openstack-glance-registry
nova-conductor: openstack-nova-conductor
glance-api: openstack-glance-api
nova-consoleauth: openstack-nova-console
nova-api: openstack-nova-api
```

Example commands using this format:

```
svc-map nova-compute

Returns: openstack-nova-compute

svc-map nova-compute

Returns: openstack-nova-compute

svc-map nova-compute

Returns: nova-compute
```

- This output can be used to filter what other tools actually install (install-services can be modified to use this for example)
- If you pass more than one service argument, the result for each service is printed on its own line.
- Individual svc-map files live within each element. For example if you have created an Apache element your svc-map YAML file should be created at elements/apache/svc-map.

4.3.84 sysctl

Add a sysctl-set-value command which can be run from within an element. Running this command will cause the sysctl value to be set on boot (by writing the value to /etc/sysctl.d).

Example usage

:: sysctl-set-value net.ipv4.ip_forward 1

4.3.85 uboot

Perform kernel/initrd post-processing for UBoot.

This element helps post-process the kernel/initrd for use with uboot. It works with ramdisk images as well as user images.

This element needs u-boot-tools to be installed on the host.

The load address and entry point for UBoot kernel can be specified as shown in the example below.

Example: export UBOOT_KERNEL_ADDR=0x80000 export UBOOT_KERNEL_EP=0x80000

4.3.86 ubuntu-common

This element holds configuration and scripts that are common for all Ubuntu images.

4.3.87 ubuntu-core

Use Ubuntu Core cloud images as the baseline for built disk images.

Overrides:

- To use a non-default URL for downloading base Ubuntu cloud images, use the environment variable DIB_CLOUD_IMAGES
- To download a non-default release of Ubuntu cloud images, use the environment variable DIB_RELEASE
- To use different mirrors rather than the default of archive.ubuntu.com and security.ubuntu.com, use the environment variable DIB_DISTRIBUTION_MIRROR

Element Dependencies

Uses

- *cache-url*
- *ubuntu-common*
- *dpkg*

4.3.88 ubuntu-minimal

The `ubuntu-minimal` element uses debootstrap for generating a minimal image. In contrast the `ubuntu` element uses the cloud-image as the initial base.

By default this element creates the latest LTS release. The exact setting can be found in the element's `environment.d` directory in the variable `DIB_RELEASE`. If a different release of Ubuntu should be created, the variable `DIB_RELEASE` can be set appropriately.

Element Dependencies

Uses

- *ubuntu-common*
- *package-installs*
- *debootstrap*

4.3.89 ubuntu-signed

The `ubuntu-signed` element installs `linux-signed-image-generic` that provides signed kernel that can be used for deploy in UEFI secure boot mode.

Element Dependencies

Uses

- *ubuntu*

4.3.90 ubuntu

Use Ubuntu cloud images as the baseline for built disk images.

Overrides:

- To use a non-default URL for downloading base Ubuntu cloud images, use the environment variable `DIB_CLOUD_IMAGES`
- To download a non-default release of Ubuntu cloud images, use the environment variable `DIB_RELEASE`. This element will export the `DIB_RELEASE` variable.

Element Dependencies

Uses

- *cloud-init-datasources*
- *cache-url*
- *ubuntu-common*
- *dpkg*

- *dkms*

Used by

- *ubuntu-signed*

4.3.91 vm

Sets up a partitioned disk (rather than building just one filesystem with no partition table).

4.3.92 yum-minimal

Base element for creating minimal yum-based images.

This element is incomplete by itself, you'll want to use the centos-minimal or fedora-minimal elements to get an actual base image.

Use of this element will require 'yum' and 'yum-utils' to be installed on Ubuntu and Debian. Nothing additional is needed on Fedora or CentOS.

If you wish to have DHCP networking setup for eth0 & eth1 via /etc/sysconfig/network-config scripts/ifcfg-eth[0|1], set the environment variable *DIB_YUM_MINIMAL_CREATE_INTERFACES* to *1*.

If you wish to build from specific mirrors, set `DIB_YUM_MINIMAL_BOOTSTRAP_REPOS` to a directory with the `.repo` files to use during bootstrap and build. The repo files should be named with a prefix `dib-mirror-` and will be removed from the final image.

4.3.93 yum

Provide yum specific image building glue.

RHEL/Fedora/CentOS and other yum based distributions need specific yum customizations.

Customizations include caching of downloaded yum packages outside of the build chroot so that they can be reused by subsequent image builds. The cache increases image building speed when building multiple images, especially on slow connections. This is more effective than using an HTTP proxy as a yum cache since the same rpm from different mirrors is often requested.

Custom yum repository configurations can also be applied by defining *DIB_YUM_REPO_CONF* to a space separated list of repo configuration files. The files will be copied to /etc/yum.repos.d/ during the image build, and then removed at the end of the build. Each repo file should be named differently to avoid a filename collision.

4.3.94 zypper-minimal

Base element for creating minimal SUSE-based images

This element is incomplete by itself so you probably want to use it along with the opensuse-minimal one. It requires 'zypper' to be installed on the host.

4.3.95 zypper

This element provides some customizations for zypper based distributions like SLES and openSUSE. It works in a very similar way as the yum element does for yum based distributions.

Zypper is reconfigured so that it keeps downloaded packages cached outside of the build chroot so that they can be reused by subsequent image builds. The cache increases image building speed when building multiple images, especially on slow connections. This is more effective than using an HTTP proxy for caching packages since the download servers will often redirect clients to different mirrors.

4.4 diskimage-builder Specifications

4.4.1 Overview

This directory is used to hold approved design specifications for changes to the diskimage-builder project. Reviews of the specs are done in gerrit, using a similar workflow to how we review and merge changes to the code itself. For specific policies around specification review, refer to the end of this document.

The layout of this directory is:

```
specs/v<major_version>/
```

Where there are two sub-directories:

- specs/v<major_version>/approved: specifications approved but not yet implemented
- specs/v<major_version>/implemented: implemented specifications
- specs/v<major_version>/backlog: unassigned specifications

The lifecycle of a specification

Developers proposing a specification should propose a new file in the `approved` directory. diskimage-builder-core will review the change in the usual manner for the project, and eventually it will get merged if a consensus is reached.

When a specification has been implemented either the developer or someone from diskimage-builder-core will move the implemented specification from the `approved` directory to the `implemented` directory. It is important to create redirects when this is done so that existing links to the approved specification are not broken. Redirects aren't symbolic links, they are defined in a file which sphinx consumes. An example is at `specs/v1/redirects`.

This directory structure allows you to see what we thought about doing, decided to do, and actually got done. Users interested in functionality in a given release should only refer to the `implemented` directory.

Example specifications

You can find an example spec in *Example Spec - The title of your specification*

Backlog specifications

Additionally, we allow the proposal of specifications that do not have a developer assigned to them. These are proposed for review in the same manner as above, but are added to:

```
specs/backlog/approved
```

Specifications in this directory indicate the original author has either become unavailable, or has indicated that they are not going to implement the specification. The specifications found here are available as projects for people looking to get involved with diskimage-builder. If you are interested in claiming a spec, start by posting a review for the specification that moves it from this directory to the next active release. Please set yourself as the new *primary assignee* and maintain the original author in the *other contributors* list.

4.4.2 Specification review policies

There are some special review policies which diskimage-builder-core will apply when reviewing proposed specifications. They are:

Trivial specifications

Proposed changes which are trivial (very small amounts of code) and don't change any of our public APIs are sometimes not required to provide a specification. The decision of whether something is trivial or not is a judgement made by the author or by consensus of the project cores, generally trying to err on the side of spec creation.

4.4.3 Approved Specifications

Block Device Setup Level 1: Partitioning

During the creation of a disk image (e.g. for a VM), there is the need to create, setup, configure and afterwards detach some kind of storage where the newly installed OS can be copied to or directly installed in.

Remark

The implementation for this proposed changed already exists, was discussed and is currently waiting for reviews [1]. To have a complete overview over the block device setup, this document is provided.

The dependencies are not implemented as they should be, because

- the spec process is currently in the phase of discussion and not finalized [2],
- the implementation was finished and reviewed before the spec process was described. [1]

Problem description

When setting up a block device there is the need to partitioning the block device.

Use Cases

User (Actor: End User) wants to create multiple partitions in multiple block devices where the new system is installed in.

The user wants to specify if the image should be optimized for speed or for size.

The user wants the same behavior independently of the current host or target OS.

Proposed change

Move the partitioning functionality from *elements/vm/block-device.d/10-partition* to a new block_device python module: *level1/partitioning.py*.

Instead of using a program or a library, the data is written directly with the help of python *file.write()* into the disk image.

Alternatives

The existing implementation uses the *parted* program (old versions of DIB were using *sfdisk*). The first implementations of this change used the python-parted library.

All these approaches have a major drawback: they automatically *optimize* based on information collected on the host system - and not of the target system. Therefore the resulting partitioning layout may lead to a degradation of performance on the target system. A change in these external programs and libraries also lead to errors during a DIB run [4] or there are general issues [7].

Also everything build around GNU parted falls under the GPL2 (not LGPL2) license - which is incompatible with the currently used Apache license in diskimage-builder.

API impact

Extends the (optional) environment variable `DIB_BLOCK_DEVICE_CONFIG`: a JSON structure to configure the (complete) block device setup. For this proposal the second entry in the original list will be used (the first part (as described in [5]) is used by the level 0 modules).

The name of this module is *partitioning* (element[0]). The value (element[1]) is a dictionary.

For each disk that should be partitioned there exists one entry in the dictionary. The key is the name of the disk (see [5] how to specify names for block device level 0). The value is a dictionary that defines the partitioning of each disk.

There are the following key / value pairs to define one disk:

label (mandatory) Possible values: 'mbr' This uses the Master Boot Record (MBR) layout for the disk. (Later on this can be extended, e.g. using GPT).

align (optional - default value '1MiB') Set the alignment of the partition. This must be a multiple of the block size (i.e. 512 bytes). The default of 1MiB (~ 2048 * 512 bytes blocks) is the default for modern systems and known to perform well on a wide range of targets [6]. For each partition there might be some space that is not used - which is *align* - 512 bytes. For the default of 1MiB exactly 1048064 bytes (= 1 MiB - 512 byte) are not used in the partition itself. Please note that if a boot loader should be written to the disk or partition, there is a need for some space. E.g. grub needs 63 * 512 byte blocks between the MBR and the start of the partition data; this means when grub will be installed, the *align* must be set at least to 64 * 512 byte = 32 KiB.

partitions (mandatory) A list of dictionaries. Each dictionary describes one partition.

The following key / value pairs can be given for each partition:

name (mandatory) The name of the partition. With the help of this name, the partition can later be referenced, e.g. while creating a file system.

flags (optional) List of flags for the partition. Default: empty. Possible values:

boot Sets the boot flag for the partition

size (mandatory) The size of the partition. The size can either be an absolute number using units like *10GiB* or *1.75TB* or relative (percentage) numbers: in the later case the size is calculated based on the remaining free space.

Example:

```
["partitioning",
   {"rootdisk": {
     "label": "mbr",
     "partitions":
        [{"name": "part-01",
          "flags": ["boot"],
          "size": "100%"}]}}]
```

Security impact

None - functionality stays the same.

Other end user impact

None.

Performance Impact

Measurements showed there is a performance degradation for the target system of the partition table is not correctly aligned: writing takes about three times longer on an incorrect aligned system vs. one that is correctly aligned.

Implementation

Assignee(s)

Primary assignee: ansreas (andreas@florath.net)

Work Items

None - this is already a small part of a bigger change [1].

Dependencies

None.

Testing

The refactoring introduces no new test cases: the functionality is tested during each existing test building VM images.

Documentation Impact

End user: the additional environment variable is described.

References

[1] Refactor: block-device handling (partitioning) https://review.openstack.org/322671

[2] Add specs dir https://review.openstack.org/336109

[3] Old implementation using parted-lib https://review.openstack.org/#/c/322671/1..7/elements/block-device/pylib/block-device/level1/Partitioning.py

[4] ERROR: embedding is not possible, but this is required for cross-disk install http://lists.openstack.org/pipermail/openstack-dev/2016-June/097789.html

[5] Refactor: block-device handling (local loop) https://review.openstack.org/319591

[6] Proper alignment of partitions on an Advanced Format HDD using Parted http://askubuntu.com/questions/201164/proper-alignment-of-partitions-on-an-advanced-format-hdd-using-parted

[7] Red Hat Enterprise Linux 6 - Creating a 7TB Partition Using parted Always Shows "The resulting partition is not properly aligned for best performance" http://h20564.www2.hpe.com/hpsc/doc/public/display?docId=emr_na-c03479326&DocLang=en&docLocale=en_US&jumpid=reg_r11944_uken_c-001_title_r0001

[8] Spec for changing the block device handling https://review.openstack.org/336946

Block Device Setup

During the creation of a disk image (e.g. for a VM), there is the need to create, setup, configure and afterwards detach some kind of storage where the newly installed OS can be copied to or directly installed in.

Problem description

Currently dib is somewhat limited when it comes to setting up the block device: only one partition that can be used for data. LVM, encryption, multi-device or installation in an already existing block device is not supported.

In addition there are several places (main, lib, elements) where the current way of handling the block device is used (spread knowledge and implementation).

Also it is not possible, to implement the handling as different elements: it is not possible to pass results of one element in the same phase to another element. Passing results from one phase to dib main is limited.

Use Cases

Possible use cases are (Actor: End User)

1. User wants to use an existing block device to install an system image in (like hd, iSCSI, SAN lun, . . .).
2. User wants that the system will be installed in multiple partitions.
3. User wants that the partitioning is done in a specific way (optimize for speed, optimize for size).
4. User wants to use LVM to install the system in (multiple PV, VG and LV).
5. User wants to encrypt a partition or a LV where (parts) of the system are installed in.
6. User wants specific file systems on specific partitions or LVs.

Please note that these are only examples and details are described and implemented by different sub-specs.

Proposed change

Because of the current way to execute elements, it is not possible to have different elements for each feature. Instead the changes will be implemented in a python module 'block_device' placed in the 'diskimage_builder' directory.

The entry-point mechanism is used to create callable python programs. These python programs are directly called from within the dib-main.

There is the need to implement some functions or classes that take care about common used new functionality: e.g. storing state between phases, calling python sub-modules and passing arguments around. These functionality is implemented as needed - therefore it is most likely that the first patch implements also big parts of these infrastructure tasks.

Alternatives

1. Rewrite DIB in the way that elements can interchange data, even if they are called during one phase. This would influence the way all existing elements are called - and might lead to unpredictable results.
2. In addition there is the need to introduce at least two additional phases: because major parts of the block device handling are currently done in main and these must be passed over to elements.
3. Another way would be to implement everything in one element: this has the disadvantage, that other elements are not allowed to use the 'block_device' phase any longer and also passing around configuration and results is still not possible (see [3]).

API impact

Is described in the sub-elements.

Security impact

Is described in the sub-elements.

Other end user impact

Paradigm changes from execute script to configuration for block_device phase.

Performance Impact

Is described in the sub-elements.

Implementation

Assignee(s)

Primary assignee: ansreas (andreas@florath.net)

Would be good, if other people would support this - and specify and implement modules.

Work Items

This is an overview over changes in the block device layer. Each level or module needs it's own spec.

A first step is to reimplement the existing functionality, this contains:

1. Level 0: Local Loop module Use loop device on local image file (This is already implemented: [1])
2. Level 1: partitioning module (This is already implemented: [4])
3. Level 2: Create File System An initial module uses ext4 only
4. Level 3: Mounting

As a second step the following functionality can be added:

- Level 1: LVM module
- Level 2: Create File System (swap)
- Level 2: Create File System (vfat, needed for UEFI)
- Level 2: Create File System (xfs)

Of course any other functionality can also be added when needed and wanted.

Dependencies

Is described in the sub-elements.

Testing

Is described in the sub-elements.

Documentation Impact

Is described in the sub-elements.

References

[1] Implementation of Level 0: Local Loop module https://review.openstack.org/319591

[2] 'Block Device Setup for Disk-Image-Builder' https://etherpad.openstack.org/p/C80jjsAs4x

[3] partitioning-parted This was a first try to implement everything as an element - it shows the limitation. https://review.openstack.org/313938

[4] Implementation of Level 1: partitioning module https://review.openstack.org/322671

Example Spec - The title of your specification

Introduction paragraph – why are we doing anything? A single paragraph of prose that operators can understand. The title and this first paragraph should be used as the subject line and body of the commit message respectively.

Some notes about the diskimage-bulider spec process:

- Not all changes need a spec. For more information see <add_url_here>

- The aim of this document is first to define the problem we need to solve, and second agree the overall approach to solve that problem.
- This is not intended to be extensive documentation for a new feature.
- You should aim to get your spec approved before writing your code. While you are free to write prototypes and code before getting your spec approved, its possible that the outcome of the spec review process leads you towards a fundamentally different solution than you first envisaged.
- But, API changes are held to a much higher level of scrutiny. As soon as an API change merges, we must assume it could be in production somewhere, and as such, we then need to support that API change forever. To avoid getting that wrong, we do want lots of details about API changes upfront.

Some notes about using this template:

- Your spec should be in ReSTructured text, like this template.
- Please wrap text at 79 columns.
- Please do not delete any of the sections in this template. If you have nothing to say for a whole section, just write: None
- For help with syntax, see http://sphinx-doc.org/rest.html
- If you would like to provide a diagram with your spec, ascii diagrams are required. http://asciiflow.com/ is a very nice tool to assist with making ascii diagrams. The reason for this is that the tool used to review specs is based purely on plain text. Plain text will allow review to proceed without having to look at additional files which can not be viewed in gerrit. It will also allow inline feedback on the diagram itself.

Problem description

A detailed description of the problem. What problem is this blueprint addressing?

Use Cases

What use cases does this address? What impact on actors does this change have? Ensure you are clear about the actors in each use case: Developer, End User, etc.

Proposed change

Here is where you cover the change you propose to make in detail. How do you propose to solve this problem?

If this is one part of a larger effort make it clear where this piece ends. In other words, what's the scope of this effort?

At this point, if you would like to just get feedback on if the problem and proposed change fit in diskimage-builder, you can stop here and post this for review to get preliminary feedback. If so please say: Posting to get preliminary feedback on the scope of this spec.

Alternatives

What other ways could we do this thing? Why aren't we using those? This doesn't have to be a full literature review, but it should demonstrate that thought has been put into why the proposed solution is an appropriate one.

API impact

Describe how this will effect our public interfaces. Will this be adding new environment variables? Deprecating existing ones? Adding a new command line argument?

Security impact

Describe any potential security impact on the system.

Other end user impact

Aside from the API, are there other ways a user will interact with this feature?

Performance Impact

Describe any potential performance impact on the system, for example how often will new code be called, does it perform any intense processing or data manipulation.

Implementation

Assignee(s)

Who is leading the writing of the code? Or is this a blueprint where you're throwing it out there to see who picks it up?

If more than one person is working on the implementation, please designate the primary author and contact.

Primary assignee: <launchpad-id or None>

Other contributors: <launchpad-id or None>

Work Items

Work items or tasks – break the feature up into the things that need to be done to implement it. Those parts might end up being done by different people, but we're mostly trying to understand the timeline for implementation.

Dependencies

- Include specific references to specs in diskimage-builder or in other projects, that this one either depends on or is related to.
- If this requires functionality of another project that is not currently used by diskimage-builder document that fact.

Testing

Please discuss the important scenarios needed to test here, as well as specific edge cases we should be ensuring work correctly. For each scenario please specify if this requires specialized hardware, or software.

Is this untestable in gate given current limitations (specific hardware / software configurations available)? If so, are there mitigation plans (gate enhancements, etc).

Documentation Impact

Which audiences are affected most by this change, and which documentation files need to be changed. Do we need to add information about this change to the developer guide, the user guide, certain elements, etc.

References

Please add any useful references here. You are not required to have any reference. Moreover, this specification should still make sense when your references are unavailable. Examples of what you could include are:

- Links to mailing list or IRC discussions
- Links to notes from a summit session
- Links to relevant research, if appropriate
- Related specifications as appropriate
- Anything else you feel it is worthwhile to refer to