Contributing to ceph-container

==============================

1. Become familiar with the [project structure](#project-structure).

2. Follow the appropriate [contribution workflow](#contribution-workflow)

3. Follow the [commit guidelines](#commit-guidelines)

Project structure

-----------------

The primary deliverables of this project are two container images: `daemon-base`, and `daemon`. As

such, the project structure is influenced by what files and configurations go into each image. The

main source base (including configuration files and build specifications) of the project is located

in `src/` and is further specified in `src/daemon-base` and `src/daemon`.

Because this project supports several different Ceph versions and many OS distros, the structure

also allows individual Ceph versions, individual distros, and combinations of

Ceph-version-and-distro (we will call these \*\*flavors\*\*) to override the base source, configuration,

and specification files by specifying their own versions of the files or new files entirely.

Mentally modeling the end result of overrides for any given flavor is difficult. Similarly,

programmatically selecting the correct files for each flavor build is also difficult. In order to

effectively work with this project structure, we introduce the concept of \*\*staging\*\*.

### The concept of staging

Special tooling has been built to collect all source files with appropriate overrides into a unique

staging directory for each flavor (each staging directory is also specified by a target

architecture). From a staging directory, containers can be built directly from the

`<staging>/daemon-base/` and `<staging>/daemon/` image directories. Additionally, developers can

inspect a staging directory's files to view exactly what will be (or has been) built into the

container images. Additionally, in order to maintain a core source base that is as reusable as

possible for all flavors, staging also supports a very basic form of templating. Some tooling has

been developed to make working with staging as easy as possible.

#### Staging override order

It is key that staging is deterministic; therefore, a clear definition of override priority is

needed. More specific files will override (overwrite) less specific files when staging. Note here

that `FILE` may be a file or a directory containing further files.

```

# Most specific

ceph-releases/<ceph release>/<base os repository>/<base os release>/{daemon-base,daemon}/FILE

ceph-releases/<ceph release>/<base os repository>/<base os release>/FILE

ceph-releases/<ceph release>/<base os repository>/{daemon-base,daemon}/FILE

ceph-releases/<ceph release>/<base os repository>/FILE

ceph-releases/<ceph release>/{daemon-base,daemon}/FILE

ceph-releases/<ceph release>/FILE

ceph-releases/ALL/<base os repository>/<base os release>/{daemon-base,daemon}/FILE

ceph-releases/ALL/<base os repository>/<base os release>/FILE

ceph-releases/ALL/<base os repository>/{daemon-base,daemon}/FILE

ceph-releases/ALL/<base os repository>/FILE

ceph-releases/ALL/{daemon-base,daemon}/FILE

ceph-releases/ALL/FILE

src/{daemon-base,daemon}/FILE

src/FILE

# Least specific

```

#### Basic templating in staging

##### Variable file replacements

In any source file, a special variable in the form `\_\_VAR\_NAME\_\_` (two leading and trailing

underscores with capital letters, digits, and underscores between) can be placed. Once all files are

staged, the `\_\_VAR\_NAME\_\_` variable will be replaced with the raw contents of the file with the

named `\_\_VAR\_NAME\_\_`. Trailing whitespace is stripped from the variable file before insertion.

`\_\_VAR\_NAME\_\_` files are allowed to be empty, but they are not allowed to be nonexistent if a file

declares them. A `\_\_VAR\_NAME\_\_` definition file may contain nested `\_\_OTHER\_VAR\_NAME\_\_` variables as

well as nested `\_\_ENV\_[ENV\_VAR]\_\_` variables (documented below).

If the `\_\_DO\_STUFF\_\_` file is supposed to contain actions that need done it generally needs to

return true. As an example, `echo 'first' && \_\_DO\_STUFF\_\_ && echo 'last'` will print 'first' and

'last' correctly only if `\_\_DO\_STUFF\_\_` returns true. A take-no-action override needs to have the

content `/bin/true`, as an empty file will cause an error.

##### Environment variable replacements

In any source file, a special variable in the form `\_\_ENV\_[ENV\_VAR]\_\_` can placed. Once all files

are staged, the `\_\_ENV\_[ENV\_VAR]\_\_` variable will be replaced with the raw contents of the

environment variable named `ENV\_VAR`. Only environment variables with all-caps, underscores, and

digits are supported. Staging will report an error if an environment variable's value is unset.

Environment variable replacements can also be nested inside of other environment variable

replacements. both `\_\_ENV\_[]\_\_` definitions and `\_\_VAR\_\_` file definitions may specify environment

variable replacements.

A typical usage is to use ``\_\_ENV\_[HOST\_ARCH]\_\_`` when you need to specify the building

architecture.

#### Staging development aids

To practically aid developers, helpful tools have been built for staging:

- To create all default staging directories: `make stage`

- To create specific staging directory(-ies): `make FLAVORS=<flavors> stage`

- Find the source of a staged file: `cd <staging dir> ; ./find-src <file-path>`

- List of staged files and their sources: `<staging dir>/files-sources`

- List of all possible buildable flavors: `make show.flavors`

- Show flavors affected by branch changes: `make flavors.modified`

- Stage log: `stage.log`

#### Building images from staging

It is possible (but not usually necessary) to build container images directly from staging in the

event that `make build` is not appealing. Simply stage the flavor(s) to be built, and then execute

the desired build command for `daemon-base` and `daemon`.

```

# Example

cd <staging>

docker build -t <daemon base tag> daemon-base/

docker build -t <daemon tag> daemon/

```

### Where does (should) source code live?

#### Core source base of ceph-container - `src/`

The `src/` directory contains the core code that is applicable for all flavors -- all Ceph versions

and all distros -- and is the "least specific" definition of source files for this project. This

includes the ceph-container-specific code that is built into the container images as well as build

configuration files. `src/` should be viewed as the base set of functionality all flavors must

implement. To maximize reuse and keep the core source base in one place, care should be taken to try

to make the core source applicable to all flavors whenever possible. Note the existence of the

"CEPH\_VERSION" environment variable (and other similar variables) built into the containers.

ceph-container source code can use container environment variables to execute code paths

conditionally, making an override of source files for specific Ceph versions unnecessary.

Where possible `src/` provides a specification of a "sane default" that may need to be

overridden for certain flavors. For example, the `src/daemon/\_\_DAEMON\_PACKAGES\_\_` file defines the

daemon packages which should be installed in the base image. A distro is not required to install

these packages via its package manager or to use this list (though it is recommended to gain as much

reuse as possible), as each distro may have a different preferred method of installation. This list

should, however, serve as a a guide to what must be installed.

#### Source shared by all Ceph releases - `ceph-releases/ALL/`

The `ceph-releases/All/` directory contains code that is generic to all Ceph releases. It is more

specific than `src/`. Source that is shared between distros (but is not part of ceph-container's

core functionality) can be placed in this directory.

Distro-specific source is placed in `ceph-releases/ALL/<distro>` and is yet more specific.

#### Source specific to a single Ceph release - `ceph-releases/<ceph release>`

A `ceph-releases/<ceph release>` directory is more specific than `ceph-releases/ALL` and contains

source that is specific to a particular Ceph release but is generic for all distros.

A `ceph-releases/<ceph release>/<distro>` directory is the most specific source directory and

contains source that is specific to a particular Ceph-release-and-distro combination.

Contribution workflow

---------------------

The goal when adding contributions should be to make modifications or additions to the least

specific files in order to reuse as much code as possible. Only specify specific changes when they

absolutely apply only to the specific flavor(s) and not to others.

### General suggestions

- Make use of already-defined `\_\_VAR\_\_` files when possible to maximize reuse.

- Make changes in the least specific (see [override order](#staging-override-order)) directory as

makes sense for the flavor to maximize reuse across flavors.

### Fixing a bug

1. Stage the flavor on which the bug was found (`make FLAVORS=<bugged flavor> stage`).

2. Use the `find-src` script or `files-sources` list to locate the bugged file's source location.

3. Edit the source location to fix the bug.

4. Build test versions of the images (`make FLAVORS=<bugged flavor> build`).

5. Test the images you built in your environment.

6. Make a PR of your changes.

### Adding a feature

1. Determine the scope of the feature

- To which Ceph versions should the feature be added?

- To which distros should the feature be added?

- As a general guideline, new features should usually be added to all Ceph versions and distros.

2. Add relevant changes to files in the project structure such that they will be added to only the

Ceph versions and distros in scope for the feature.

3. Build test versions of the images (`make FLAVORS=<in-scope-flavors> build`).

4. Test the images in your environment.

5. Make a PR of your changes.

### Adding a Ceph release

Ideally, adding a new Ceph release is fairly easy. In the best case, all that needs done is adding

flavors for the new Ceph version to the Makefile. At minimum, `ALL\_BUILDABLE\_FLAVORS` must be

updated in the Makefile. If distro source is properly configured to support multiple Ceph releases

and there are no special updates required, they are likely to work with just this minimal change.

Note the `$CEPH\_VERSION` and `$CEPH\_POINT\_RELEASE` variables usually used in `\_\_DOCKERFILE\_INSTALL\_\_`

are extracted from the first field of the flavor name.

In this example, `luminous,centos,7`, `$CEPH\_VERSION` will be set to \*\*luminous\*\*.

Adding a new flavor name like `mimic,centos,7` is enough to create a new \*\*mimic\*\*

Ceph release.

In the worst case, trying to make as few modifications as possible:

1. Add flavors for new Ceph versions to the Makefile.

- At minimum: `ALL\_BUILDABLE\_FLAVORS`.

2. Edit `src/` files to support the new version if necessary, making sure not to break previous

versions. Keep container environment variables in mind here.

3. Edit `ceph-releases/ALL/<distro>` files to support the new version if necessary, making sure not

to break previous versions.

4. Add `ceph-releases/<new ceph version>` files to support the new version if necessary.

5. Build test versions of the images (`make FLAVORS=<new release flavors> build`).

6. Test the images in your environment.

7. Make a PR of your changes.

### Adding a distro build

1. Add flavors for the new distro to the Makefile.

- At minimum: `ALL\_BUILDABLE\_FLAVORS`.

2. Add a `ceph-releases/ALL/<new distro>` directory for the new distro

- Make sure to install all the required packages for `daemon-base` (see

`src/daemon-base/\_\_CEPH\_BASE\_PACKAGES\_\_)` and for `daemon` (see

`src/daemon/\_\_DAEMON\_PACKAGES\_\_`).

- Make sure to specify all `\_\_VAR\_\_` files without sane defaults for the container builds.

- Make sure to override any `\_\_VAR\_\_` file sane defaults that do not apply to the new distro.

- Refer to other distros for inspiration.

3. If necessary, add a `ceph-release/<ceph release>/<new distro>` directory for the new distro. Try

to avoid this as much as possible and focus on code reuse from the less specific dirs.

4. Build test versions of the images (`make FLAVORS=<new distro flavors> build`).

5. Test the images in your environment.

6. Make a PR of your changes.

Commit guidelines

-----------------

- All commits should have a subject and a body

- The commit subject should briefly describe what the commit changes

- The commit body should describe the problem addressed and the chosen solution

- What was the problem and solution? Why that solution? Were there alternative ideas?

- Wrap commit subjects and bodies to 72 characters

- Sign-off your commits

- Add a best-effort scope designation to commit subjects. This could be a directory name, file name,

or the name of a logical grouping of code. Examples:

- \*\*[dir]\*\* ceph-releases: change flavor specification(s) in the `ceph-container` dir

- \*\*[dir]\*\* mimic: change flavor spec for mimic

- \*\*[dir]\*\* kubernetes: edit a Kubernetes example in `examples/kubernetes`

- \*\*[file]\*\* osd\_disk\_activate: edit the `src/daemon/osd\_scenarios/osd\_disk\_activate.sh` file

- \*\*[logical group]\*\* osd prep: change how OSDs are prepared, with changes in multiple files

- \*\*[combinations]\*\* mimic osd prep: change how OSDs are prepared only in mimic

Suggested reading: https://chris.beams.io/posts/git-commit/

CI

-----

### Jenkins

We use Jenkins to run several tests on each pull request.

If you don't want to run a build for a particular pull request, because all you are changing is the

README for example, add the text `[skip ci]` to the PR title.

### Travis

We use Travis to run tests for the demo scenario.

You can check the files in `travis-builds` to learn more about this process.