OSBS Documentation

Release 1.0

Luiz Carvalho

Oct 15, 2020

Contents

1 Table of Contents **3**

1.1 Administration and Maintenance . . . 3
1.1.1 Understanding build parameters . . . 3
1.1.2 Configuring osbs-client . . . 3
1.1.3 Deploy OSBS on OpenShift . . . 4
1.1.4 Setting up koji for container image builds . . . 7
1.1.5 Priority of Container Image Builds . . . 9
1.1.6 Operator manifests . . . 10
1.1.7 Cachito integration . . . 11
1.1.8 Troubleshooting . . . 12
1.2 Building Container Images . . . 13
1.2.1 Building images using fedpkg . . . 13
1.2.2 Building images using koji . . . 13
1.2.3 Building images using osbs-client . . . 16
1.2.4 Accessing built images . . . 16
1.2.5 Writing a Dockerfile . . . 16
1.2.6 Image configuration . . . 17
1.2.7 Content Sets . . . 24
1.2.8 Using Artifacts from Koji . . . 24
1.2.9 Image tags . . . 26
1.2.10 Override Parent Image . . . 26
1.2.11 Koji NVR . . . 27
1.2.12 Isolated Builds . . . 27
1.2.13 Yum repositories . . . 28
1.2.14 Signing intent . . . 29
1.2.15 Base image builds . . . 30
1.2.16 Multistage builds . . . 31
1.2.17 Operator manifests . . . 31
1.3 Inspecting built image components . . . 36
1.3.1 Dockerfiles . . . 36
1.3.2 Image Content Manifests . . . 36
1.4 Building Source Container Images . . . 36
1.4.1 Signing intent resolution . . . 37
1.4.2 Koji integration . . . 37
1.4.3 Building source container images using koji . . . 37
1.4.4 Building source container images using osbs-client . . . 38

1.5 Understanding the Build Process . . . 38
1.5.1 Arrangement Version 6 (reactor_config_map) . . . 38
1.5.2 Logging . . . 40
1.5.3 Metadata Fragment Storage . . . 42
1.5.4 Autorebuilds . . . 43
1.6 Build Parameters . . . 43
1.6.1 Environment vs User Parameters . . . 43
1.6.2 Reactor Configuration . . . 43
1.6.3 Atomic Reactor Plugins and Arrangement Version 6 . . . 46
1.6.4 Creating Builds . . . 46
1.6.5 Rendering Plugins . . . 47
1.6.6 Secrets . . . 47
1.6.7 Site Customization . . . 47
1.7 Contributing to OSBS . . . 47
1.7.1 Setting up a (local) development environment . . . 47
1.7.2 Contribution guidelines . . . 48
1.8 Documentation of the OSBS tools . . . 50
1.8.1 Koji Containerbuild . . . 50
1.8.2 Atomic Reactor . . . 50
1.8.3 OSBS Client . . . 50
1.8.4 Ansible Playbook . . . 50
1.8.5 Deploying and Running OSBS . . . 50

2 Indices and tables **51**

OSBS is a collection of tools, workflows and integration points that build and release layered container images.

OSBS hooks into Koji with the help of the koji-containerbuild plugin, and uses OpenShift builds as Content Generators to produce layered images.

One can start an image build with `fedpkg`, using the `container-build` subcommand:

```
$ fedpkg container-build
```

CHAPTER 1

Table of Contents

1.1 Administration and Maintenance

1.1.1 Understanding build parameters

Please refer to *Build Parameters* for information on how options are configured within OSBS builds.

1.1.2 Configuring osbs-client

When submitting a new build request to OSBS as a user, this request is for an orchestrator build. When the orchestrator build wants to create worker builds it also does this through osbs-client.

As a result there are two osbs.conf files to consider:

- the one external to OSBS, for creating orchestrator builds, and
- the one internal to OSBS, stored in a Kubernetes secret (named by *client_config_secret*) in the orchestrator cluster

These can have the same content. The important features are discussed below.

can_orchestrate

The parameter `can_orchestrate` defaults to false. The API method `create_orchestrator_build` will fail unless `can_orchestrate` is true for the chosen instance section.

reactor_config_map

`reactor_config_map` specifies the name of a Kubernetes configmap holding *Server-side Configuration for atomic-reactor*. A pre-build plugin will read its value from REACTOR_CONFIG environment variable.

client_config_secret

When `client_config_secret` is specified this is the name of a Kubernetes secret (holding a key `osbs.conf`) for use by atomic-reactor when it creates worker builds. The *orchestrate_build* plugin is told the path to this.

token_secrets

When `token_secrets` is specified the specified secrets (space separated) will be mounted in the OpenShift build. When ":" is used, the secret will be mounted at the specified path, i.e. the format is:

```
token_secrets = secret:path secret:path ...
```

This allows an `osbs.conf` file (from *client_config_secret*) to be constructed with a known value to use for `token_file`.

Node selector

When an entry with the pattern `node_selector.platform` (for some *platform*) is specified, builds for this platform submitted to this cluster must include the given node selector, so as to run on a node of the correct architecture. This allows for installations that have mixed-architecture clusters and where node labels differentiate architecture.

If the value is `none`, this platform is the only one available and no node selector is required.

Platform description

When a section name begins with "platform:" it is interpreted not as an OSBS instance but as a platform description. The remainder of the section name is the platform name being described. The section has the following keys:

architecture (optional) the GOARCH for the platform – the platform name is assumed to be the same as the GOARCH if this is not specified

build_from

Specifies the build image (AKA "buildroot") to be used for building container images, to be set in the `Build`/`BuildConfig` OpenShift objects under the `.spec.strategy.customStrategy.from` object. This can be a full reference to a specific container image in a container registry; or it can reference an ImageStreamTag object.

Updating this globally effectively deploys a different version of OSBS.

It takes one of the following forms:

imagestream:*imagestream*:*tag* use the image from the specified OpenShift ImageStreamTag

image:*pullspec* pull the image from the specified pullspec (including registry, repository, and either tag or digest)

1.1.3 Deploy OSBS on OpenShift

Authentication

The orchestrator cluster will have a service account (with edit role) created for use by Koji builders. Those Koji builders will use the service account's persistent token to authenticate to the orchestrator cluster and submit builds to it.

Since the orchestrator build initiates worker builds on the worker cluster, it must have permission to do so. A service account should be created on each worker cluster in order to generate a persistent token. This service account should have edit role. On the orchestrator cluster, a secret for each worker cluster should be created to store the corresponding service account tokens. When osbs-client creates the orchestrator build it must specify the names of the secret files to be mounted in the BuildConfig. The orchestrator build will extract the token from the mounted secret file.

Server-side Configuration for atomic-reactor

This will list the maximum number of jobs that should be active at any given time for each cluster. It will also list worker clusters in order of preference. It may also contain additional environment configuration such as ODCS integration.

The runtime configuration will take the form of a Kubernetes secret with content as in the example below:

```
---
clusters:
  x86_64:
  - name: prod-x86_64-osd
    max_concurrent_builds: 16
  - name: prod-x86_64
    max_concurrent_builds: 6
    enabled: true
  - name: prod-other
    max_concurrent_builds: 2
    enabled: false

  ppc64le:
  - name: prod-ppc64le
    max_concurrent_builds: 6

odcs:
  signing_intents:
  - name: release
    keys: [AB123]
  - name: beta
    keys: [BT456, AB123]
  - name: unsigned
    keys: []
  # Value must match one of the names above.
  default_signing_intent: release
```

clusters

This maps each platform to a list of clusters and their concurrent build limits. For each platform to build for, a worker cluster is chosen as follows:

- clusters with the enabled key set to false are discarded
- each remaining cluster in turn will be queried to discover all currently active worker builds (not failed, complete, in error, or cancelled)
- the cluster load is computed by dividing the number of active worker builds by the specified maximum number of concurrent builds allowed on the cluster
- the worker build is submitted to whichever cluster has the lowest load; in this way, an even load distribution across all clusters is enforced

There are several throttles preventing too many worker builds being submitted. Each worker cluster can be configured to only schedule a certain number of worker builds at a time by setting a default resource request. The orchestrator cluster will similarly only run a certain number of orchestrator builds at a time based on the resource request in the orchestrator build JSON template. A Koji builder will only run a certain number of containerbuild tasks based on its configured capacity.

This mechanism can also be used to temporarily disable a worker cluster by removing it from the list or adding `enabled: false` to the cluster description for each platform.

odcs

Section used for ODCS related configuration.

signing_intents List of signing intents in their restrictive order. Since composes can be renewed in ODCS, OSBS needs to check if the signing keys used in a compose to be renewed are still valid. If the signing keys are not valid anymore, i.e., keys were removed from the OSBS signing intent definition, OSBS will request ODCS to update the compose signing keys. For OSBS to identify the proper signing intent in such cases, you should not remove signing keys from signing intents. Instead, move the keys that should not be valid anymore from the `keys` map to the `deprecated_keys` map in the relevant signing intent definitions. Failing to do so will result in build failures when renewing composes with old signing intent key sets.

default_signing_intent Name of the default signing intent to be used when one is not provided in `container.yaml`.

build_env_vars

Define variables that should be propagated to the build environment here. Note that some variables are reserved and defining them will cause an error, e.g. `USER_PARAMS`, `REACTOR_CONFIG`.

For example, you might want to set up an HTTP proxy:

```
build_env_vars:
- name: HTTP_PROXY
  value: "http://proxy.example.com"
- name: HTTPS_PROXY
  value: "https://proxy.example.com"
- name: NO_PROXY
  value: localhost,127.0.0.1
```

Limiting image size

You can check the binary image's size before it is pushed to a registry. If it exceeds the configured size, the built image will not be pushed and the build fails.

A typical configuration in reactor config map looks like:

```
image_size_limit:
  binary_image: 10000
```

The value is the size in bytes of uncompressed layers. When either `binary_image` or `image_size_limit` is omitted, or if `binary_image` is set to `0`, the check will be skipped.

1.1.4 Setting up koji for container image builds

Example configuration file: Koji builder

The configuration required for submitting an orchestrator build is different than that required for the orchestrator build itself to submit worker builds. The `osbs.conf` used by the Koji builder would include:

```
[general]
build_json_dir = /usr/share/osbs/

[platform:x86_64]
architecture = amd64

[default]
openshift_url = https://orchestrator.example.com:8443/
build_image = example.registry.com/buildroot:blue

distribution_scope = public

can_orchestrate = true  # allow orchestrator builds

# This secret contains configuration relating to which worker
# clusters to use and what their capacities are:
reactor_config_map = reactorconf

# This secret contains the osbs.conf which atomic-reactor will use
# when creating worker builds
client_config_secret = osbsconf

# These additional secrets are mounted inside the build container
# and referenced by token_file in the build container's osbs.conf
token_secrets =
  workertoken:/var/run/secrets/atomic-reactor/workertoken

# and auth options, registries, secrets, etc

[scratch]
openshift_url = https://orchestrator.example.com:8443/
build_image = example.registry.com/buildroot:blue

reactor_config_map = reactorconf
client_config_secret = osbsconf
token_secrets = workertoken:/var/run/secrets/atomic-reactor/workertoken

# All scratch builds have distribution-scope=private
distribution_scope = private

# This causes koji output not to be configured, and for the low
# priority node selector to be used.
scratch = true

# and auth options, registries, secrets, etc
```

This shows the configuration required to submit a build to the orchestrator cluster using `create_prod_build` or `create_orchestrator_build`.

Also shown is the configuration for scratch builds, which will be identical to regular builds but with "private" distribution scope for built images and with the scratch option enabled.

Example configuration file: inside builder image

The `osbs.conf` used by the builder image for the orchestrator cluster, and which is contained in the Kubernetes secret named by `client_config_secret` above, would include:

```
[general]
build_json_dir = /usr/share/osbs/

[platform:x86_64]
architecture = amd64

[prod-mixed]
openshift_url = https://worker01.example.com:8443/
node_selector.x86_64 = beta.kubernetes.io/arch=amd64
node_selector.ppc64le = beta.kubernetes.io/arch=ppc64le
use_auth = true

# This is the path to the token specified in a token_secrets secret.
token_file =
  /var/run/secrets/atomic-reactor/workertoken/worker01-serviceaccount-token

# The same builder image is used for the orchestrator and worker
# builds, but used with different configuration. It should not
# be specified here.
# build_image = registry.example.com/buildroot:blue

# and auth options, registries, secrets, etc

[prod-osd]
openshift_url = https://api.prod-example.openshift.com/
node_selector.x86_64 = none
use_auth = true
token_file =
  /var/run/secrets/atomic-reactor/workertoken/osd-serviceaccount-token
# and auth options, registries, secrets, etc
```

In this configuration file there are two worker clusters, one which builds for both x86_64 and ppc64le platforms using nodes with specific labels (prod-mixed), and another which only accepts x86_64 builds (prod-osd).

Including OpenShift build annotations in Koji task output

It is possible to include a `build_annotations.json` file in the task output of successful container image builds. This file may include any wanted OpenShift build annotations for the container build triggered by the Koji task in question.

The `koji-containerbuild` plugin looks for a `koji_task_annotations_whitelist` annotation in the OpenShift build annotations. This key should hold a list of annotations to be whitelisted for inclusion in the `build_annotations.json` file.

If an empty `build_annotations.json` file would be generated through the process described above, the file is omitted from the task output. For instance, `koji_task_annotations_whitelist` could be empty, or the whitelisted annotations not present in OpenShift build annotations.

To whitelist the desired annotations in the `koji_task_annotations_whitelist` OpenShift annotation described above, you can use the `task_annotations_whitelist koji` configuration in the `reactor_config_map`. See *Server-side Configuration for atomic-reactor* for further reference.

The `build_annotations.json` file is a JSON object with first level key/values where each key is a whitelisted OpenShift build annotation mapped to it's value.

1.1.5 Priority of Container Image Builds

For a build system it's desirable to prioritize different kinds of builds in order to better utilize resources. Unfortunately, OpenShift's scheduling algorithm does not support setting a priority value for a given build. To achieve some sort of build prioritization, we can leverage node selectors to allocate different resources to different build types.

Consider the following types of container builds:

- *scratch build*
- *explicit build*
- *auto rebuild*

As the name implies, *scratch builds* are meant to be used as a one-off unofficial container build. No guarantees are made for storing the created container images long term. It's also not meant to be shipped to customers. These are clearly low priority builds.

Explicit builds are those triggered by a user, either directly via fedpkg/koji CLI, or indirectly via pungi (as in the case of base images). These are official builds that will go through the normal life cycle of being tested and, eventually, shipped.

Auto rebuilds are created by OpenShift when a change in the parent image is detected. It's likely that layered images should be rebuilt in order to pick up changes in latest parent image.

For any *explicit build* or *auto rebuild*, they may or may not be high priority. In some cases, a build is high priority due to a security fix, for instance. In other cases, it could be due to an in-progress feature. For this reason, it cannot be said that all *explicit builds* are higher priority than *auto rebuilds*, or vice-versa.

However, *auto rebuilds* have the potential of completely consuming OSBS's infrastructure. There must be some mechanism to throttle the amount of *auto rebuilds*. For this reason, OSBS uses a different node selector for each different build type:

- *scratch build*: builds_scratch=true
- *explicit build*: builds_explicit=true
- *auto rebuild*: builds_auto=true

By controlling each type of builds individually, OSBS will have the necessary control for adjusting its infrastructure.

For example, consider an OpenShift cluster with 5 compute nodes:

Node	builds_scratch=true	builds_explicit=true	builds_auto=true
Node 1	✓	✓	✓
Node 2		✓	
Node 3			✓
Node 4		✓	✓
Node 5		✓	✓

In this case, *scratch builds* can be scheduled only on **Node 1**; *explicit builds* on any node except **Node 3**; and auto builds on any node except **Node 2**.

Worker Builds Node Selectors

The build type node selectors are only applied to worker builds. This gives more granular control over available resources. Since worker builds are the ones that actually perform the container image building steps, it requires more resources than orchestrator builds. For this reason, a deployment is more likely to have more nodes available for worker builds than orchestrator builds. This is important because the amount of nodes available defines the granularity of how builds are spread across the cluster.

For instance, consider a large deployment in which only 2 orchestrator nodes are needed. If build type node selectors are applied to orchestrator builds, builds can only be throttled by a factor of 2. In contrast, this same deployment may use 20 worker builds, allowing builds to be throttled by a factor of 20.

Orchestrator Builds Allocation

Usually in a deployment, the amount of allowed orchestrator builds matches the amount of allowed worker builds for any given platform. Additional orchestrator builds should be allowed to fully leverage the build type node selectors on worker builds since some orchestrator builds will wait longer than usual for their worker builds to be scheduled. This provides a buffer that allows OpenShift to properly schedule worker builds according to their build type via node selectors. Because OpenShift scheduling is used, worker builds of same type will run in the order they were submitted.

Koji Builder Capacity

The task load of the Koji builders used by OSBS will not reflect the actual load on the OpenShift cluster used by OSBS. The disparity is due to auto rebuilds not having a corresponding Koji task. This creates a scenario where a buildContainer Koji task is started, but the OpenShift build remains in pending state. The Koji builder capacity should be set based on how many nodes allow **scratch builds** and/or **explicit builds**. In the example above, there are 4 nodes that allow such builds.

The log file, *osbs-client.log*, in a Koji task gives users a better understanding of any delays due to scheduling.

1.1.6 Operator manifests

Supporting Operator Manifests extraction

To support the operator manifests extraction, as described in *Operator manifests*, the *operator-manifests* BType must be created in koji. This is done by running

```
koji call addBType operator-manifests
```

Enabling Operator Manifests digest pinning (and other replacements)

To enable digest pinning and other replacements of image pullspecs for *operator manifest bundle* builds, atomic-reactor config must include the `operator_manifests` section. See configuration details in config.json.

Example:

```
operator_manifests:
  allowed_registries:
    - private-registry.example.com
    - public-registry.io
  repo_replacements:
    - registry: private-registry.example.com
```

(continues on next page)

(continued from previous page)

```
    package_mappings_url: https://somewhere.net/package_mapping.yaml
  registry_post_replace:
    - old: private-registry.example.com
      new: public-registry.io
  skip_all_allow_list:
    - koji_package1
    - koji_package2
```

allowed_registries List of allowed registries for images *before* replacement. If any image is found whose registry is not in `allowed_registries`, build will fail. This key is required.

Should be a subset of `source_registry + pull_registries` (see config.json).

repo_replacements Each registry may optionally have a "package mapping" - a YAML file that contains a mapping of [package name => list of repos] (see package_mapping.json). The file needs to be uploaded somewhere that OSBS can access, and will be downloaded from there during build if necessary.

Images from registries with a package mapping will have their namespace/repo replaced. OSBS will query the registry to find the package name for the image (determined by the component label) and get the matching replacement from the mapping file. If there is no replacement, or if there is more than one, build will fail and user will have to specify one in `container.yaml`.

registry_post_replace Each registry may optionally have a replacement. After pinning digest and replacing namespace/repo, all `old` registries in image pullspecs will be replaced by their `new` replacements.

skip_all_allow_list List of koji packages which are allowed to use `skip_all` option in the `operator_manifests` section of `container.yaml`.

Enabling integration with OMPS service

To enable optional integration with OMPS service to allow automatically pushing operators manifests to application registry (like quay) `omps` configuration section must be added into atomic-reactor configuration. See configuration details in config.json.

Example:

```
omps:
  omps_url: https://omps-service.example.com
  omps_namespace: organization
  omps_secret: /dir/where/token/file/will/be/mounted
  appregistry_url: https://quay.io/cnr
```

1.1.7 Cachito integration

cachito caches specific versions of upstream projects source code along with dependencies and provides a single tarball with such content for download upon request. This is important when you want track the version of a project and its dependencies in a more robust manner, without handing control of storing and handling the source code for a third party (e.g., if tracking is performed in an external git forge, someone could force push a change to the repository or simply delete it).

OSBS is able to use cachito to handle the source code used to build a container image. The source code archive provided by cachito and the data used to perform the cachito request may then be attached to the koji build output, making it easier to track the components built in a given container image.

This section describes how to configure OSBS to use cachito as described above. *Fetching source code from external source using cachito* describes how to get OSBS to use cachito in a specific container build, as an OSBS user.

Configuring your cachito instance

To enable cachito integration in OSBS, you must use the `cachito` configuration in the `reactor_config_map`. See configuration details in config.json.

Example:

```
cachito:
  api_url: https://cachito.example.com
  auth:
    ssl_certs_dir: /dir/with/cert/file
```

Configuring koji

Adding remote-sources BType

To fully support cachito integration, as described in *Cachito integration*, the *remote-sources* BType must be created in koji. This is done by running

```
koji call addBType remote-sources
```

This new build type will hold cachito related build artifacts generated in atomic-reactor, which should include a tarball with the upstream source code for the software installed in the container image and a *remote-source.json* file, which is a JSON representation of the source request sent to cachito by atomic-reactor. This JSON file includes information such as the repository from where cachito downloaded the source code and the revision reference that was downloaded (e.g., a git commit hash).

Whitelisting *remote_source_url* build annotation

In addition to adding the new BType to koji, you may also want to whitelist the OpenShift *remote_source_url* build annotation. This is specially useful for scratch builds, where a koji build is not generated and users would not have information about how the sources were fetch for that build easily available. *whitelist-annotations* describes the steps needed to whitelist OpenShift build annotations.

1.1.8 Troubleshooting

Builds will automatically cancel themselves if any worker takes more than 3 hours to complete or the entire task takes more than 4 hours to complete. Administrators can override these run time values with the `worker_max_run_hours` and `orchestrator_max_run_hours` settings in the `osbs.conf` file.

Obtaining Atomic Reactor stack trace

atomic-reactor captures *SIGUSR1* signals. When receiving such signal, atomic-reactor responds by showing the current stack trace for every thread it was running when the signal was received.

An administrator can use this to inspect the orchestrator or a specific worker build. It is specially useful to diagnose stuck builds.

As an administrator, use `podman kill --signal=SIGUSR1 <BUILDROOT_CONTAINER>` or `podman exec <BUILDROOT_CONTAINER> kill -s SIGUSR1 1` to send the signal to the buildroot container you wish to inspect. atomic-reactor will dump stack traces for all its threads into the buildroot container logs. For instance:

```
Thread 0x7f6e88a1b700 (most recent call first):
  File "/usr/lib/python2.7/site-packages/atomic_reactor/inner.py", line 277, in run
  File "/usr/lib64/python2.7/threading.py", line 812, in __bootstrap_inner
  File "/usr/lib64/python2.7/threading.py", line 785, in __bootstrap

Current thread 0x7f6e95dbf740 (most recent call first):
  File "/usr/lib/python2.7/site-packages/atomic_reactor/util.py", line 74, in dump_
↪traceback
  File "/usr/lib/python2.7/site-packages/atomic_reactor/util.py", line 1562, in dump_
↪stacktraces
  File "/usr/lib64/python2.7/socket.py", line 476, in readline
  File "/usr/lib64/python2.7/httplib.py", line 620, in _read_chunked
  File "/usr/lib64/python2.7/httplib.py", line 578, in read
  File "/usr/lib/python2.7/site-packages/urllib3/response.py", line 203, in read
  File "/usr/lib/python2.7/site-packages/docker/client.py", line 247, in _stream_
↪helper
  File "/usr/lib/python2.7/site-packages/atomic_reactor/util.py", line 297, in wait_
↪for_command
  File "/usr/lib/python2.7/site-packages/atomic_reactor/plugins/build_docker_api.py",␣
↪line 46, in run
  File "/usr/lib/python2.7/site-packages/atomic_reactor/plugin.py", line 239, in run
  File "/usr/lib/python2.7/site-packages/atomic_reactor/plugin.py", line 449, in run
  File "/usr/lib/python2.7/site-packages/atomic_reactor/inner.py", line 444, in build_
↪docker_image
  File "/usr/lib/python2.7/site-packages/atomic_reactor/inner.py", line 547, in build_
↪inside
  File "/usr/lib/python2.7/site-packages/atomic_reactor/cli/main.py", line 95, in cli_
↪inside_build
  File "/usr/lib/python2.7/site-packages/atomic_reactor/cli/main.py", line 292, in run
  File "/usr/lib/python2.7/site-packages/atomic_reactor/cli/main.py", line 310, in run
  File "/usr/bin/atomic-reactor", line 11, in <module>
```

In this example, this build is stuck talking to the docker client (`docker/client.py`).

1.2 Building Container Images

1.2.1 Building images using fedpkg

Following command submits a build to Koji:

```
fedpkg container-build --target=<target>
```

For detailed Fedora workflow please visit Fedora Layered Image Build System guide.

1.2.2 Building images using koji

Using a koji client CLI directly you have to specify git repo URL and branch:

```
koji container-build <target> <repourl>#<branch/ref> --git-branch <branch>
```

The koji-containerbuild plugin provides the `container-build` sub-command in the `koji` CLI. Please install the plugin in order to access this sub-command:

```
sudo yum install python3-koji-containerbuild-cli
```

You will now have the `container-build` sub-command available on your workstation. For a full list of options:

```
koji container-build --help
```

Streamed build logs

When atomic-reactor in the orchestrator build runs its *orchestrate_build* plugin and watches the builds, it will stream in the logs from those builds and emit them as logs itself, with the platform name as one of the fields. The extra fields for these worker logs will be: platform, level.

Note that there will be a single Koji task with a single log output, which will contain logs from multiple builds. When watching this using `koji watch-logs <task id>` the log output from each worker build will be interleaved. To watch logs from a particular worker build image owners can use `koji watch-logs <task id> | grep -w x86_64`.

Koji Build Results

Koji Web

This is the easiest way to access information about OSBS builds.

List Builds

Navigate to the "Builds" tab in koji and set the "Type" filter to `image`.

Get Build

If you have the build ID, go to <KOJI_WEB_URL>/buildinfo?buildID=<build-ID>

If you want to search build by its name or part of name, use the search box on top of the page. For example, `redis-*` and select "Builds".

In koji build you can find a lot information about build, some noticeable are:

- pull specifications for the build in the 'Extra' section `image.index.pull`, for digest type in `image.index.digests`
- list of image archives for each specific architecture for which build was executed (for more detailed information about specific archive click on 'info')

Also in the "Extra" section, docker.config shows (parts of) the docker image JSON description, as well it indicates container image API version.

See atomic-reactor documentation for a full description of the Koji container image metadata.

Get Task

All OSBS builds triggered via koji have a task linked to them. On the Build info page, look at the "Extra" field for the `container_koji_task_id` value. When you locate this task ID integer, go to <KOJI_WEB_URL>/taskinfo?taskID=<task-ID> to find the task responsible for the build.

Build Logs

The logs can be found in task's "Output" section (older builds will have that section empty as logs has been garbage collected), or in build's "Logs" section (persist after garbage collection).

Koji CLI

List Builds

List all image (OSBS) builds:

```
koji call listBuilds type=image
```

Apply filter for more specific search:

```
koji call listBuilds type=image createdAfter='2016-02-01 00:00:00' prefix=redis
```

Search for builds of specific users:

```
koji call listUsers prefix=<user>  # get user-ID
koji call listbuilds type=image userId=<user-ID>
```

Get Build

Retrieve build information from either the build ID or the build NVR:

```
koji buildinfo <build-ID or build-NVR>
```

Get Task

The "Extra" field in build result is useful to track the task that originated this build. Use the "container_koji_task_id", or "filesystem_koji_task_id", to get more info about task:

```
koji taskinfo <task-ID>
```

Cancel Task

You can cancel a buildContainer koji task as for other types of task, and this will cancel the OSBS build:

```
koji cancel <task-ID>
```

Build Notifications

Package owners and build submitter will be notified via email about build.

1.2.3 Building images using osbs-client

osbs-client provides `osbs` CLI command for interaction with OSBS builds and allows creation of new builds directly without koji-client.

Please note that mainly `koji` and `fedpkg` commands should be used for building container images instead of direct `osbs-client` calls.

To execute build via osbs-client CLI use:

```
osbs build -g <git_repo_url> -b <branch> -u <username> --git-commit <commit> [--
↪platforms=x86_64] [-i <instance>]
```

To see full list of options execute:

```
osbs build --help
```

To see all osbs-client subcommands execute:

```
osbs --help
```

Please note that `osbs-client` must be configured properly using config file `/etc/osbs.conf`. Please refer to *osbs-client configuration section* for configuration examples.

1.2.4 Accessing built images

Information about registry and image name is included in koji build. Use one of names listed in `extra.image.index.pull` to pull built image from a registry.

1.2.5 Writing a Dockerfile

A Dockerfile is required for building container images in OSBS. It must be placed at the root of git repository. There can only be a single Dockerfile per git repository branch.

Some labels are required to be defined:

- `com.redhat.component`: the value of this label is used when importing a build into Koji via content generator API. We recommend that all images use a component string ending in `-container` here, so that you can easily distinguish these container builds from other non-container builds in Koji. The value can't be empty.
- `name`: value is used to define the repository name in container registry to push built image. Limit this to lowercase alphanumerical values with the possibility to use dash as a word separator. A single `/` is also allowed. `.` is not allowed in the first section. For instance, **fed/rsys.log** and **rsyslog** are allowed, but **fe.d/rsyslog** and **rsys.log** aren't. The value can't be empty.
- `version`: used as the version portion of Koji build NVR, as well as, for the version tag in container repository. The value can't be empty, and may be defined via ENV variable from parent.

For example:

```
LABEL com.redhat.component=rsyslog-container \
      name=fedora/rsyslog \
      version=32
```

When OSBS builds a container image that defines the above labels, a Koji build will be created in the format rsyslog-container-32-X. Where X is the release value. The container image will be available in container registry at: `my-container-registry.example.com/fedora/rsyslog:32`.

The `release` label can also be used to specify the release value use for Koji build. The value can't be empty, and may be defined via ENV variable from parent. When omitted, the release value will be automatically determined by querying Koji's getNextRelease API method.

Other labels are set automatically when not set in the Dockerfile:

- `build-date`: Date/Time image was built as RFC 3339 date-time.
- `architecture`: Architecture for the image.
- `com.redhat.build-host`: OpenShift node where image was built.
- `vcs-ref`: A reference within the version control repository; e.g. a git commit.
- `vcs-type`: The type of version control used by the container source. Currently, only git is supported.

Although it is also possible to automatically include the `vcs-url` label, the default set of automatically included labels does not include the label.

Sites wanting to include the `vcs-url` label to the set should do so by using custom `orchestrator_inner:n.json` and `worker_inner:n.json` specifying the full set of implicit labels for the `add_labels_in_dockerfile` plugin:

```
{
  "args": {
    "auto_labels": ["build-date", "architecture", "vcs-type", "vcs-url", "vcs-ref",
↪"com.redhat.build-host"]
  },
  "name": "add_labels_in_dockerfile"
},
```

Finally, it is also possible to set additional labels through the reactor configuration, by setting the label key values in `image_labels`.

1.2.6 Image configuration

Some aspects of the container image build process are controlled by a file in the git repository named `container.yaml`. This file need not be present, but if it is it must adhere to the container.yaml schema.

An example:

```
---
platforms:
  # all these keys are optional

  only:
  - x86_64   # can be a list (as here) or a string (as below)
  - ppc64le
  - armhfp
  not: armhfp

remote_source:
  repo: https://git-forge.example.com/namespace/repo.git
  ref: AddFortyCharactersGitCommitHashRightHere
  pkg_managers:
```

(continues on next page)

(continued from previous page)

```
  - npm
  packages:
    npm:
    - path: client
    - path: proxy

compose:
  # used for requesting ODCS compose of type "tag"
  packages:
  - nss_wrapper  # package name, not an NVR.
  - httpd
  - httpd-devel
  # used for requesting ODCS compose of type "pulp"
  pulp_repos: true
  # used for requesting ODCS compose of type "module"
  modules:
  - "module_name1:stream1"
  - "module_name2:stream1"
  # Possible values, and default, are configured in OSBS environment.
  signing_intent: release
  # used for inheritance of yum repos and ODCS composes from baseimage build
  inherit: true

image_build_method: docker_api

autorebuild:
  from_latest: false
  add_timestamp_to_release: false
```

platforms

Keys in this map relate to multi-platform builds. The full set of platforms for which builds may be required will come initially from the Koji build tag associated with the build target, or from the `platforms` parameter provided to the `create_orchestrator_build` API method when Koji is not used.

only list of platform names (or a single platform name as a string); this restricts the platforms to build for using set intersection

not list of platform names (or a single platform name as a string); this restricts the platforms to build for using set difference

go

> **Warning:** Using this key is deprecated in favor of using Cachito integration

Keys in this map relate to source code in the Go language which the user intends to be built into the container image. They are responsible for building the source code into an executable themselves. Keys here are only for identifying source code which was used to create the files in the container image.

modules sequence of mappings containing information for the Go modules (packages) built and shipped in the container image. The accepted mappings are listed bellow.

module top-level go module (package) name to be built in the image. If `modules` is specified, this entry is required.

archive possibly-compressed archive containing full source code including vendored dependencies.

path path to directory containing source code (or its parent), possibly within archive.

compose

This section is used for requesting yum repositories at build time. When this section is defined, a compose will be requested by using ODCS.

packages list of package names to be included in ODCS compose. Package in this case refers to the "name" portion of the NVR (name-version-release) of an RPM, not the Koji package name. Packages will be selected based on the Koji build tag of the Koji build target used. The following command is useful in determining which packages are available in a given Koji build tag: `koji list-tagged --inherit --latest TAG`

If "packages" key is declared but is empty (`packages: []` in YAML), the compose will include all packages from the Koji build tag of the Koji build target.

ODCS will work more quickly if you only specify the minimum set of packages you need here, but if you want to avoid hard-coding a complete package list in `container.yaml`, you can use the empty list to just make everything available.

pulp_repos boolean to control whether or not an ODCS compose of type "pulp" should be requested. If set to true, `content_sets.yml` must also be provided. A compose will be requested for each architecture in `content_sets.yml`. See *Content Sets*. Additionally also `build_only_content_sets` will be used if provided.

modules list of modules for requesting ODCS compose of type "module". ODCS will cherry-pick each module into the compose.

Use this `modules` option to make module builds available that are not yet available from the other options like Pulp. This is useful if you want to test a newly-built module before it is available in Pulp, or if you want to pin to a specific module that MBS has built.

This list can be of the format `name:stream`, `name:stream:version`, or `name:stream:version:context`.

If you specify a `name:stream` without specifying a `version:context`, ODCS will query MBS to find the very latest `version:context` build. For example, if you specify `go-toolset:rhel8`, ODCS will query MBS for the latest `go-toolset` module build for the `rhel8` stream, whereas if you specify `go-toolset:rhel8:8020020200128163444:0ab52eed`, ODCS will compose that exact module instead.

Note that if you simply specify a `name:stream` for a module, ODCS will compose the very latest module that a module developer has built for that stream, and this module might not be tested by QE or GPG signed. Alternatively, if your desired module is already QE'd, signed, and available in Pulp, the `pulp_repos: true` option will ensure that your container build environment only uses tested and signed modules.

signing_intent used for verifying packages in yum repositories are signed with expected signing keys. The possible values for signing intent are defined in OSBS environment. See *odcs* section for environment configuration details, and full explanation of *Signing intent*.

inherit boolean to control whether or not to inherit yum repositories and odcs composes from baseimage build, default false. Scratch and isolated builds do not support inheritance and false is always assumed.

include_unpublished_pulp_repos If you set `include_unpublished_pulp_repos: true` under the `compose` section in `container.yaml`, the ODCS composes can pull from unpublished pulp repositories.

The default is `false`. Use this setting to make pre-release RPMs available to your container images. Use caution with this setting, because you could end up publicly shipping container images with RPMs that you have not exposed publicly otherwise.

ignore_absent_pulp_repos If you set `ignore_absent_pulp_repos: true` under the `compose` section in `container.yaml`, ODCS will ignore missing content sets. Use this setting if you want to pre-configure your container's `content_sets.yml` in dist-git before a Pulp administrator creates all the repositories you expect to use in the future. Alternatively, do not enable this setting if you want to enforce strict error-checking on all the the content set names in `content_sets.yml`.

multilib_method List of methods used to determine if a package should be considered multilib. Available methods are `iso`, `runtime`, `devel`, and `all`.

multilib_arches Platform list for which the multilib should be enabled. For each entry in the list, ODCS will also include packages from other compatible architectures in the compose. For example when "x86_64" is included, ODCS will also include "i686" packages in the compose.

modular_koji_tags List of Koji tags in which the modular Koji Content Generator builds are tagged. Such builds will be included in the compose. When `true` is specified instead of list, koji build tag of the koji build target will be used instead.

build_only_content_sets Content sets used only for building content, not for distributing. Will be used only if `pulp_repos` is set to true. These content sets won't be included in ICM *Image Content Manifests*. A compose will be requested for each architecture additionally with `content_sets.yml`. Definition is the same as for `content_sets.yml` See *Content Sets*.

If there is a "modules" key, it must have a non-empty list of modules. The "packages" key, and only the "packages" key, can have an empty list.

The "packages", "modules", "modular_koji_tags" and "pulp_repos" keys can be used mutually.

flatpak

This section holds the information needed to build a Flatpak. For more information on Flatpak builds, see flatpak-docs. This is a map with the following keys:

id The ID of the application or runtime. Required.

name `name` label in generated Dockerfile. Used for the repository when pushing to a registry. Defaults to the module name.

component `com.redhat.component` label in generated Dockerfile. Used to name the build when uploading to Koji. Defaults to the module name.

base_image The image that is used when installing packages to create the filesystem. It is also recorded as the parent image of the output image. This defaults to the `flatpak: base_image` setting in the **reactor-config-map**.

branch The branch of the application or runtime. In many cases, this will match the stream name of the module. Required.

cleanup-commands A shell script that is run after installing all packages. Only applicable to runtimes.

command The name of the executable to run to start the application. If not specified, defaults to the first executable found in /usr/bin. Only applicable to applications.

tags Tags to add to the Flatpak metadata for searching. Only applicable to applications.

finish-args Arguments to `flatpak build-finish` (see the flatpak-build-finish man page). This is a string split on white space with shell style quoting. Only applicable to applications.

tags

List of tags to be applied to the built image. When this option is specified, the tags described will be applied to the image. If present, the `{version}`, `latest`, and the tags listed in the `additional-tags` file will no longer be automatically applied. See the *image-tags* section below for further reference.

version

This key is no longer used by OSBS and is only kept in the schema for backwards compatibility.

set_release_env

Optional string. If set, osbs-client will modify each stage of the image's Dockerfile, adding an ENV statement immediately following the FROM statement. The ENV statement will assign an environment variable with the same name as the value of set_release_env and the value of the current build's release number. Users can use this environment variable to get the release value when running tools inside the container.

autorebuild

This map accepts keys, as described below. This values are only used for autorebuilds, if autorebuilds are enabled.

from_latest Boolean to control whether to rebuild from the latest commit in the build branch. Defaults to `false`. When `true`, each autorebuild will use the latest commit in build branch, when `false`, each autorebuild will use the original commit from build branch of original build.

add_timestamp_to_release Boolean to control whether to append timestamp to explicitly specified release for autorebuilds. Defaults to `false`. When `true` it will append timestamp to release with `.` separator. For example if name is `fedora/rsyslog`, version is `32`, and release is `5`, the container image will be available in container registry at: `my-container-registry.example.com/fedora/rsyslog:32-5.20191007151825`.

ignore_isolated_builds Boolean to control whether to rebuild when parent image triggering build was isolated build. Defaults to `false`. When `true` and base image specified for autorebuild is isolated build, it won't trigger the autorebuild. When `false` and base image specified for autorebuild is isolated build, it will trigger the autorebuild as usual.

Automatic Rebuilds

This section specifies whether and how a build should be rebuilt based on changes to the base parent image.

By default autorebuild is disabled. The feature can be enabled by making some changes in your dist-git repo and submitting a container build.

You can modify behavior of autorebuilds in container.yaml within *autorebuild*.

Enabling Automatic Rebuilds

Enable autorebuild in config:

```
fedpkg container-build-setup --set-autorebuild true
```

This will create/update the .osbs-repo-config file. The file will be automatically added for commit.

Finally, add all modified files, commit, and push modifications. For these **changes to take place, request a container build** as usual:

```
fedpkg container-build
```

This must be a regular non-scratch/non-isolated build. The steps above apply to a single branch in your dist-git repo. It must be repeated for each branch you wish to enable the feature.

The next time the parent image used by your container image is updated, your image will be automatically rebuilt.

Disabling Automatic Rebuilds

First, use fedpkg to disable autorebuild:

```
fedpkg container-build-setup --set-autorebuild false
```

This will create/update the .osbs-repo-config file. The file will be automatically added for commit.

Finally, commit and push modifications. For these **changes to take place, request a container build** as usual:

```
fedpkg container-build
```

This must be a regular non-scratch/non-isolated build. The steps above apply to a single branch in your dist-git repo. It must be repeated for each branch you wish to disable the feature.

image_build_method

This string indicates which build-step plugin to use in order to perform the layered image build, on a per-image basis. The **docker_api** plugin uses the docker-py module to run the build via the Docker API, while the **imagebuilder** plugin uses the imagebuilder utility to do the same. Both have similar capabilities, but the **imagebuilder** plugin brings two advantages:

1. It performs all changes made in the build in a single layer, which is a little more efficient and removes the need to squash layers afterward.
2. It can perform multistage builds without requiring Docker 17+ (which Red Hat and Fedora do not support).

In order to use the **imagebuilder** plugin, the imagebuilder binary must be available and in the PATH for the builder image, or an error will result.

Fetching source code from external source using cachito

As described in *Cachito integration*, it is possible to use cachito to download a tarball with an upstream project and its dependencies and make it available for usage during an OSBS build.

remote_source

This map contains configuration of what sources OSBS will request from cachito and how they will be requested. The keys accepted here are described below. If OSBS cachito integration is not configured in the OSBS instance, the entries here will be ignored.

repo String with an URL to the upstream project SCM repository, such as `https://git.example.com/team/repo.git`. Required.

ref String with a 40-character reference to the SCM reference of re project described in `repo` to be fetched. This should be a complete git commit hash. Required.

pkg_managers A list of package managers to be used for resolving the upstream project dependencies. If not provided, Cachito will assume `gomod` due to backward compatibility reasons, however, this default could be configured differently on different Cachito deployments (make sure to check with your Cachito instance admins). Finally, if this is set to an empty array (`[]`), Cachito will provide the sources with no package manager magic. In other words, no environment variables, dependencies, or extra configuration will be provided with the sources.

flags List of flags to pass to the cachito request. See the cachito documentation for further reference.

packages A map of package managers where each value is an array of maps describing custom behavior for the packages of that package manager. For example, if you have two npm packages in the same source repository, you can specify the subdirectories with the `path` key. For example `{"npm": [{"path": "client"}, {"path": "proxy"}]}`.

Once the *remote_source* map described above is set in `container.yaml`, you can now copy the upstream sources (with bundled dependencies) provided by cachito in your build image by adding:

```
COPY $REMOTE_SOURCE $REMOTE_SOURCE_DIR
```

to your Dockerfile. From this point on, you will find the root of the upstream project at `$REMOTE_SOURCE_DIR/app`, and the project dependencies at `$REMOTE_SOURCE_DIR/deps`.

Note that `$REMOTE_SOURCES_DIR` is a build arg, available only in build time, with the absolute path to the directory where the sources are expected to be (OSBS sets other build args such as `GOPATH` for Golang sources relying on the existence of the dependencies in this directory). Hence, for cleaning up the image after using the sources, add the following line to the Dockerfile after the build is complete:

```
RUN rm -rf $REMOTE_SOURCE_DIR
```

`$REMOTE_SOURCE` is another build arg, which points to the extracted tar archive provided by cachito in the buildroot workdir.

Note: To better use the cachito provided dependencies, a full gomod supporting Golang version is required. In other words, you should use Golang >= 1.13

Replacing project dependencies with cachito

Cachito also provides a feature to allow users to replace a project's dependencies with another version of that same dependency or with a completely different dependency (this is useful when you want to use a patched fork for a dependency).

OSBS allows users to use this feature for test purposes. In other words, you can use cachito dependency replacements for scratch builds, and **only for scratch builds**.

You can use this feature using the `--replace-dependency` option, which is available for the `fedpkg`, `koji`, and `osbs` commands.

This option expects a string with the following information, separated by the `:` character: `pkg_manager:name:version[:new_name]`, where `pkg_manager` is the package manager used by cachito to handle the dependency; `name` is the name of the dependency to be replaced; `version` is the new version of the dependency to be injected by cachito; and `new_name` is an optional entry, to inform cachito that the dependency known as `name` by the package manager should be replaced with a new dependency, known as `new_name` by the package manager.:

```
fedpkg container-build --scratch --replace-dependency gomod:pagure.org/cool-go-
↪project:v1.2 gomod:gopkg.in/foo:2:github.com/bar/foo
```

or:

```
koji container-build [...] --scratch --replace-dependency gomod:pagure.org/cool-go-
↪project:v1.2 --replace-dependency gomod:gopkg.in/foo:2:github.com/bar/foo
```

In the examples above, two dependencies would be replaced. cool-go-project would be used in version `v1.2`, no matter what version is specified by the project requesting it. Whereas `gopkg.in/foo` will be replaced by `github.com/bar/foo` version 2.

Note that while in `fedpkg` the replace dependency option receives multiple parameters, the same option should be specified multiple times in `koji` or the `osbs` CLI. This was done to keep the consistency with the similar option to specify yum repository URLs in each particular CLI.

1.2.7 Content Sets

The file `content_sets.yml` is used to define the content sets relevant to the container image. This is relevant if RPM packages in container image are in pulp repositories. See `pulp_repos` in *compose* for how this file is used during build time. If this file is present, it must adhere to the content_sets.yml schema.

An example:

```
---
x86_64:
- server-rpms
- server-extras-rpms

ppc64le:
- server-for-power-le-rpms
- server-extras-for-power-le-rpms
```

1.2.8 Using Artifacts from Koji

During a container build, it might be desirable to fetch some artifacts from an existing Koji build. For instance, when building a Java-based container, JAR archives from a Koji build are required to be added to the resulting container image.

The atomic-reactor pre-build plugin, fetch_maven_artifacts, can be used for including non-RPM content in a container image during build time. This plugin will look for the existence of two files in the git repository in the same directory as the Dockerfile: fetch-artifacts-koji.yaml and fetch-artifacts-url.yaml. (See fetch-artifacts-url.json and fetch-artifacts-nvr.json for their YAML schema.)

The first is meant to fetch artifacts from an existing Koji build. The second allows specific URLs to be used for fetching artifacts. Note that all combinations of the yaml files here described are valid, i.e., you can have either one of the two files in your repository or you could have both files in the repository. `fetch-artifacts-koji.yaml` will be processed first.

fetch-artifacts-koji.yaml

```
- nvr: foobar # All archives will be downloaded

- nvr: com.sun.xml.bind.mvn-jaxb-parent-2.2.11.redhat_4-1
  archives:
  # pull a specific archive
  - filename: jaxb-core-2.2.11.redhat-4.jar
    group_id: org.glassfish.jaxb

  # group_id omitted - multiple archives may be downloaded
  - filename: jaxb-jxc-2.2.11.redhat-4.jar

  # glob support
  - filename: txw2-2.2.11.redhat-4-*.jar

  # pull all archives for a specific group
  - group_id: org.glassfish.jaxb

  # glob support with group_id restriction
  - filename: txw2-2.2.11.redhat-4-*.jar
    group_id: org.glassfish.jaxb

  # causes build failure due to unmatched archive
  - filename: archive-filename-with-a-typo.jar
```

Each archive will be downloaded to artifacts/<mavenfile_path> at the root of git repository. It can be used from Dockerfile via ADD/COPY instruction:

```
COPY \
  artifacts/org/glassfish/jaxb/jaxb-core/2.2.11.redhat-4/jaxb-core-2.2.11.redhat-4.
↪jar /jars
```

The directory structure under `artifacts` directory is determined by `koji.PathInfo.mavenfile` method. It's essentially the end of the URL after `/maven/` when downloading archive from Koji Web UI.

Upon downloading each file, the plugin will verify the file checksum by leveraging the checksum value in the archive info stored in Koji. If checksum fails, container build fails immediately. The checksum algorithm used is dictated by Koji via the *checksum_type* value in the archive info.

If build specified in nvr attribute does not exist, the container build will fail.

If any of the archives does not produce a match, the container build will fail. In other words, every item in the archives list is expected to match at least one archive from specified Koji build. However, the build will not fail if it matches multiple archives.

Note that only archives of maven type are supported. If in the nvr supplied an archive item references a non maven artifact, the container build will fail due to no archives matching request.

fetch-artifacts-url.yaml

```
- url: http://download.example.com/JBossDV/6.3.0/jboss-dv-6.3.0-teiid-jdbc.jar
  md5: e85807e42460b3bc22276e6808839013
- url: http://download.example.com/JBossDV/6.3.0/jboss-dv-6.3.0-teiid-javadoc.jar
  # Use different hashing algorithm
  sha256: 3ba8a145a3b1381d668203cd73ed62d53ba8a145a3b1381d668203cd73ed62d5
  # Optionally, overwrite target name
  target: custom-dir/custom-name.jar
```

Each archive will be downloaded to artifacts/<target_path> at the root of git repository. It can be used from Dockerfile via ADD/COPY instruction:

```
COPY artifacts/jboss-dv-6.3.0-teiid-jdbc.jar /jars/
COPY artifacts/custom-dir/custom-name.jar /jars/
```

By default, target_path is set to the filename from provided url. It can be customized by providing a target. The target value can be either a filename, archive.jar, or also include a path, my/path/archive.jar, for easier archive management.

The md5, sha1, sha256 attributes specify the corresponding hash to be used when verifying artifact was downloaded properly. At least one of them is required. If more than one is defined, multiple hashes will be computed and verified.

Koji Build Metadata Integration

In the future, a reference of each artifact fetched by OSBS will be added to the koji build metadata once imported via content generator API. The list of components for the container image in output list will include the fetched artifacts in addition to the installed RPMs.

1.2.9 Image tags

OSBS's atomic-reactor pushes the new container image to the container registry (or Pulp, if Pulp integration is enabled) and updates various tag references in the registry. In addition, when multi-platform builds are enabled, atomic-reactor groups each set of images into a manifest list and tags that manifest list.

OSBS determines the name of the repository from the `name` label in the Dockerfile. There are three categories of tags that OSBS creates when tagging the resulting image in the registry:

- A "**unique**" tag: This tag includes the timestamp of when the image was built. For scratch builds, this is the only tag that OSBS applies. Example: `rsync-containers-candidate-93619-20191017205627`
- A "**primary**" tag: This tag is the `{version}-{release}` for the image (a combination of the `version` and `release` labels in the Dockerfile). Example: `4-2`. This tag is unique for each Koji build.
- "**floating**" tag(s): These tags transition to newer image references over time. In other words, every time you build a new container image, OSBS updates these floating tags. Examples: `latest`, or `{version}`

  Floating tags are configurable. If you set `tags` in container.yaml, OSBS applies those tags to your newly-built image as floating tags.

  If you do not set `tags` in container.yaml, OSBS applies the following floating tags automatically:

  - `{version}` (the `version` label)
  - `latest`
  - any additional tags named in the `additional-tags` file (DEPRECATED and will no longer be supported in a future version. Please consider using `tags` in container.yaml instead)

These tags are applied to the manifest list or, if multi-platform image builds are not enabled (see *group_manifests*), to the sole image manifest resulting from the build.

1.2.10 Override Parent Image

OSBS uses the `FROM` instruction in the Dockerfile to find a parent image for a layered image build. Users can override this behavior by specifying a koji parent build via the `koji_parent_build` API parameter. When a user specifies a `koji_parent_build` parameter, OSBS will look up the image reference for that koji build and override the

`FROM` instruction with that image instead. The same source registry restrictions apply. For multi-stage builds, the `koji_parent_build` parameter will only override the final `FROM` instruction.

Additionally, the koji parent build must use the same container image repository as the value of the FROM instruction in Dockerfile. For instance, if the Dockerfile states `FROM fedora:27`, the koji parent build has to be of a container image that pushed to the `fedora` repository. The koji parent build may refer to a `fedora:26` image, but using a koji parent build for an image that was pushed to `rsyslog` will cause a build failure.

This behavior requires koji integration to be enabled in the OSBS environment.

1.2.11 Koji NVR

When koji integration is enabled, every container image build requires a unique Name-Version-Release, NVR. The Name and Version are extracted from the **name** and **version** labels in Dockerfile. Users can also use the **release** label to hard code the release value, although this requires a git commit for every build to change the value. A better alternative is to leave off the **release** label which causes OSBS to query koji for what the next release value should be. This is done via koji's `getNextRelease` API method. In either case, the release value can also be overridden by using the `release` API parameter.

During the build process, OSBS will query koji for the builds of all parent images using their NVRs. If any of the parent image builds is not found in koji, or if NVR information cannot be extracted from the parent image, OSBS assumes that the parent image was not built by OSBS and halts the current build. In other words, an image cannot be built using a parent image which has not been built by OSBS. It is possible to disable this feature through reactor configuration, with `skip_koji_check_for_base_image` option in config.json, when there are no NVR labels set on the base image, if the NVR labels are set on the base image, the check is performed regardless.

Digests verification

Once OSBS has the koji build information for a parent image, it compares the digest of the parent image manifest available in koji metadata (stored when that parent build had completed) with the actual parent image manifest digest (calculated by OSBS during the build). In case manifests do not match, the build will fail and the parent image **must** be rebuilt in OSBS before it is used in another build.

If the manifest in question is a manifest list and the digests comparison fail, the V2 manifest digests in the manifest list will be compared with the koji build archive metadata digests. In this case, OSBS will only halt the build with an error, advising rebuilding the parent image, if the V2 manifest digests in the manifest list do not match the analogous koji information. This behavior can be deactivated through the `deep_manifest_list_inspection` option. See config.json for further reference.

Manifest lists can be manually pushed to the registry to make sure a specific tag (e.g., latest) is available for all platforms. In such cases, these manifest lists may include images from different koji builds. OSBS will only perform digest checks for the images requested in the current build. Moreover, build requests for platforms that were not built in the same koji build as the one found for the given image reference (manifest list) will fail.

It is also possible to have OSBS only warn about any digest mismatches (instead of halting the build with an error). This is done by setting the `fail_on_digest_mismatch` option to false in the config.json file.

1.2.12 Isolated Builds

In some cases, you may not want to update the floating tags for certain builds.

Consider the case of a container image that includes packages that have new security vulnerabilities. To address this issue, you must build a new container image. You only want to apply changes related to the security fixes, and you want to ignore any new unrelated development work. It is not correct to update the `latest` floating tag reference for this build. You can use OSBS's isolated builds feature to achieve this.

As an example, let's use the image `rsyslog` again. At some point the container image 7.4-2 is released (version 7.4, release 2). Soon after, minor bug fixes are addressed in 7.4-3, a new feature is added to 7.4-4, and so on. A security vulnerability is then discovered in the released image 7.4-2. To minimize disruption to users, you may want to build a patched version of 7.4-2, say 7.4-2.1. The packages installed in this new container image will differ from the former only when needed to address the security vulnerability. It will not include the minor bug fixes from 7.4-3, nor the new features added in 7.4-4. For this reason, updating the `latest` tag is considered incorrect.

```
7.4 version
|
|_____
|    |1 release
|
|____________________
|    |2 release     |2.1 release
|
|_____
|    |3 release
|
|_____
|    |4 release
|
```

To start an isolated build, use the `isolated` boolean parameter. Due to the nature of isolated builds, you must explicitly specify your build's `release` parameter, which must match the format `^\d+\.\d+(\..+)?$`.

Here is an example of an isolated build using fedpkg:

```
fedpkg container-build --isolated --build-release=2.1
```

Isolated builds will only create the `{version}-{release}` primary tag and the unique tag in the container registry. OSBS does not update any floating tags for an isolated build.

1.2.13 Yum repositories

In most cases, part of the process of building container images is to install RPM packages. These packages must come from yum repositories. There are various methods for making a yum repository available for your container build.

ODCS compose

The preferred method for injecting yum repositories in container builds is by enabling ODCS integration via the "compose" key in `container.yaml`. See *Image configuration* and *Signing intent* for details.

RHEL subscription

If the underlying host is Red Hat Enterprise Linux (RHEL), its subscriptions will be made available during container builds. Note that changes in the underlying host to enable/disable yum repositories is not reflected in container builds. `Dockerfile` must explicitly enable/disable yum repositories as needed. Although this is desirable in most cases, in an OSBS deployment it can cause unexpected behavior. It's recommended to disable subscription for RHEL hosts when they are being used by OSBS.

Yum repository URL

As part of a build request, you may provide the `repo-url` parameter with the URL to a yum repository file. This file is injected into the container build. Current OSBS versions support the combination of ODCS composes with repository files. This is a change to OSBS former behavior, where the ODCS compose would be disabled if a repository file URL was given.

Koji tag

When Koji integration is enabled, a Koji build target parameter is provided. The yum repository for the build tag of target is automatically injected in container build. This behavior is disabled if either "ODCS compose" or "Yum repository URL" are used.

Inherited yum repository and ODCS compose

If you want to inherit yum repositories and ODCS composes from baseimage build, you can enable it via the "inherit" key under "compose" in `container.yaml`. Does not support scratch or isolated builds. See *Image configuration*.

1.2.14 Signing intent

When the "compose" section in `container.yaml` is defined, ODCS composes will be requested at build time. ODCS is aware of RPM package signatures and can be used to ensure that only signed packages are added to the generated yum repositories. Ultimately, this can be used to ensure a container image only contains packages signed by known signing keys.

Signing intents are an abstraction for signing keys. It allows the OSBS environment administrator to define which signing keys are valid for different types of releases. See *odcs* section for details.

For instance, an environment may provide the following signing intents: `release`, `beta`, and `unsigned`. Each one of those intents is then mapped to a list of signing keys. These signing keys are then used during ODCS compose creation. The packages to be included must have been signed by any of the signing keys listed. In the example above, the intents could be mapped to the following keys:

```
# Only include packages that have been signed by "my-release-key"
release -> my-release-key
# Include packages that have been signed by either "my-beta-key" or
# "my-release-key"
beta -> my-beta-key, my-release-key
# Do not check signature of packages - may include unsigned packages
unsigned -> <empty>
```

The signing intents are also defined by their restrictive order, which will be enforced when building layered images. For instance, consider the case of two images, X and Y. Y uses X as its parent image (FROM X). If image X was built with "beta" intent, image Y's intent can only be "beta" or "unsigned". If the dist-git repo for image Y has it configured to use "release" intent, this value will be downgraded to "beta" at build time.

Automatically downgrading the signing intent, instead of failing the build, is important for allowing a hierarchy of layered images to be built automatically by `ImageChangeTriggers`. For instance, with Continuous Integration in mind, a user may want to perform daily builds without necessarily requiring signed packages, while periodically also producing builds with signed content. In this case, the `signing_intent` in `container.yaml` can be set to `release` for all the images in hierarchy. Whether or not the layered images in the hierarchy use signed packages can be controlled by simply overriding the signing intent of the top most ancestor image. The signing intent of the layered images would then be automatically adjusted as needed.

In the case where multiple composes are used, the least restrictive intent is used. Continuing with our previous signing intent example, let's say a container image build request uses two composes. Compose 1 was generated with no signing keys provided, and compose 2 was generated with "my-release-key". In this case, the intent is "unsigned".

Compose IDs can be passed in to OSBS in a build request. If one or more compose IDs are provided, OSBS will classify the intent of the existing compose. This is done by inspecting the signing keys used for generating the compose and performing a reverse mapping to determine the signing intent. If a match cannot be determined, the build will fail. Note that if given compose is expired or soon to be expired, OSBS will automatically renew it.

The `signing_intent` specified in `container.yaml` can be overridden with the build parameter of same name. This particular parameter will be ignored for autorebuilds. The value in `container.yaml` should always be used in that case. Note that the signing intent used by the compose of parent image is still taken into account which may lead to downgrading signing intent for the layered image.

The Koji build metadata will contain a new key, `build.extra.image.odcs.signing_intent_overridden`, to indicate whether or not the `signing_intent` was overridden (CLI parameter, automatically downgraded, etc). This value will only be `true` if `build.extra.image.odcs.signing_intent` does not match the `signing_intent` in `container.yaml`.

1.2.15 Base image builds

OSBS is able to create base images, and it does by creating Koji image-build task, importing its output as a new container image, then continuing to build using a Dockerfile that inherits from that imported image.

Each dist-git branch should have the following files:

- Dockerfile
- image-build.conf
- kickstart.ks (or any .ks name, but must match what image-build.conf references)

The Dockerfile should start "FROM koji/image-build", and continue with LABEL and CMD etc instructions as needed.

The image-build.conf file should start "[image-build]" and set the target (for the image-build task), distro, and ksversion, for example:

```
[image-build]
target = f30
distro = Fedora-30
ksversion = Fedora
```

The image-build task will need to know where to find the kickstart configuration; it finds this from the 'ksurl' and 'kickstart' parameters in image-build.conf. If these are absent from the file in dist-git, atomic-reactor will provide defaults:

- kickstart: 'kickstart.ks'
- ksurl: the dist-git URL and commit hash used for the OSBS build

In this way, the kickstart configuration can be placed in the dist-git repository as 'kickstart.ks' alongside the Dockerfile and image-build.conf files, and the correct git URL and commit hash will be recorded in Koji when the image is built. This is the recommended way of providing a kickstart configuration for base images.

Alternatively it can be stored elsewhere (perhaps another git repository) in which case a URL is needed. However, when doing this please make sure to use a git commit hash in the 'ksurl' parameter instead of a symbolic name (e.g. branch name); failure to do this means there will be no reliable way to discover the kickstart configuration used for the built image.

To execute base image build, run:

```
fedpkg container-build --target=<target> --repo=url=<repo-url>
```

The –repo-url parameter specifies the URL to a repofile. The first section of this is inspected and the 'baseurl' is examined to discover the compose URL. You can also use –compose-id parameter to specify ODCS composes from which additional yum repos will be used.

1.2.16 Multistage builds

Often users may wish to build an image directly from project sources (rather than intermediate build artifacts), but not include the sources or toolchain necessary for compiling the project in the final image. Multistage builds are a simple solution.

Multistage refers to container image builds with at least two stages in the Dockerfile; initial stage(s) provide a build environment and produce some kind of artifact(s) which in the final stage are copied into a clean base image. The most obvious signature of a multistage build is that the Dockerfile has more than one "FROM" statement. For example:

```
FROM toolchain:latest AS builder1
ADD .
RUN make artifact

FROM base:release
COPY --from=builder1 artifact /dest/
```

In most respects, multistage builds operate very similarly to multiple single-stage builds; the results from initial stage(s) are simply not tagged or used except by later `COPY --from` statements. Refer to Docker multistage docs for complete details.

In OSBS, multistage builds require using the **imagebuilder** plugin, which can be configured as the system default or per-image in `container.yaml`.

In a multistage build, yum repositories are made available in all stages. The build may have multiple parent builds, as each stage may specify a different image. The parent images FROM initial stages are pulled and rewritten similarly as the parent in the final stage (known as the "base image"). Note that ENV and LABEL entries from earlier stages do not affect later stages.

Note that the `COPY --from=<image>` form (with a full image specification as opposed to a stage alias) should not be used in OSBS builds. It works, but the image used is not treated as other parents are (rewritten, etc). To achieve the same effect, specify such images with another stage, for example:

```
FROM registry.example.com/image:tag AS source1
FROM base
COPY --from=source1 src/ dest/
```

1.2.17 Operator manifests

OSBS is able to extract operator manifests from an operator image. This image should contain a `/manifests` directory, whose content can be extracted to koji for later distribution.

To activate the operator manifests extraction from the image, you must set a specific label in your Dockerfile to identify your build as either an operator bundle build or an appregistry build:

```
LABEL  com.redhat.delivery.appregistry=true
LABEL  com.redhat.delivery.operator.bundle=true
```

Only one of these labels (the appropriate one for your build) may be present, otherwise build will fail.

When present (and set to `true`), this label triggers the atomic-reactor `export_operator_manifests` plugin. This plugin extracts the content from the `/manifests` directory in the built image and uploads it to koji. If the `/manifests` directory is either empty or not present in the image, the build will fail.

Since the operator manifests are not tied to any specific architecture, OSBS will decide from which worker build the manifests will be extract (and make sure only a single platform will upload the archive to koji). If, for some reason, you need to select which platform will extract and upload the manifests archive, you can set the `operator_manifests_extract_platform` build param to the desired platform.

[Backward compatibility] If the build succeeds, the `build.extra.operator_manifests_archive` koji metadata will be set to the name of the archive containing the operator manifests (currently, `operator_manifests.zip`).

The operator manifests archive is uploaded to koji as a separate type: `operator-manifests` (currently with filename `operator_manifests.zip`).

Operator manifest bundle builds

This type of build is for the newer style of operator manifests targeting Openshift 4.4 or higher. It is identified by the `com.redhat.delivery.operator.bundle` label.

To make OSBS cooperate on building your operator manifest bundle, you will need to set up the following:

Dockerfile

```
# Base needs to be scratch
FROM scratch

# Make this an operator bundle build
LABEL com.redhat.delivery.operator.bundle=true

# Does not matter where you keep your manifests in the repo, but in the
# final image, they need to be in /manifests
COPY my-manifests-dir/ /manifests
```

container.yaml (see `operator_manifests` in container.yaml schema)

```
operator_manifests:
  # Relative path to your manifests dir from root of repo
  manifests_dir: my-manifests-dir
```

Pinning pullspecs for related images

In addition to extracting the `/manifests` dir to koji after build as described above, the `pin_operator_digest` plugin is able to pin related image pullspecs in your **ClusterServiceVersion** files.

Put simply, `pin_operator_digest` replaces floating tags in image pullspecs with manifest list digests, creates a `.spec.relatedImages` section in the file and puts all the pullspecs in it.

Note: If your **ClusterServiceVersion** file has a non-empty `.spec.relatedImages` section, OSBS will assume that it already contains correctly pinned pullspecs and will not touch the file.

Additionally, the plugin provides repo and registry replacement to allow workflows where you use private or testing pullspecs in your manifest and OSBS replaces them with final (perhaps customer-facing) pullspecs.

Replacing pullspecs

Each step of pullspec replacement can be enabled/disabled using the corresponding option in `container.yaml`. By default, all steps are enabled.

```
operator_manifests:
  enable_digest_pinning: true
  enable_repo_replacements: true
  enable_registry_replacements: true
```

digest pinning Query registry for manifest list digest, replace tag with it.

E.g. `private.com/test/foobar:v1` -> `private.com/test/foobar@sha256:123...`

repo replacements Query registry for package name (determined by component label in image), replace namespace/repo based on said package name. Only applies to images from registries that have a package mapping configured, either in OSBS site configuration or in `container.yaml`:

```
operator_manifests:
  repo_replacements:
    - registry: private.com
      package_mappings:
        foobar-package: foo/bar
```

E.g. `private.com/test/foobar@sha256:123...` -> `private.com/foo/bar@sha256:132...`

It may happen that the registry of one of your images has a package mapping, but is missing the replacement / has multiple possible replacements for a package. In this case, the build will fail and you will need to define the replacement in `container.yaml` as shown above.

registry replacements Based purely on OSBS site configuration, after digest is pinned and repo is replaced, registry may also be replaced if atomic-reactor is configured to do so.

E.g. `private.com/foo/bar@sha256:123...` -> `public.io/foo/bar@sha256:123...`

Pullspec locations

Before OSBS can pin your pullspecs, it first needs to find them. Because it is practically impossible to tell if a string is a pullspec, atomic-reactor has a predefined set of locations where it will look for pullspecs.

1. metadata.annotations.containerImage anywhere in the file

   jq: `.. | .metadata?.annotations.containerImage | select(. != null)`

2. All containers in each deployment

   jq: `.spec.install.spec.deployments[].spec.template.spec.containers[]`

3. All initContainers in each deployment

   jq: `.spec.install.spec.deployments[].spec.template.spec.initContainers[]`

4. All RELATED_IMAGE_* variables for all containers and initContainers

   jq: `.env[] | select(.name | test("RELATED_IMAGE_"))` for each of [2], [3]

5. All pullspecs from all annotations. This is done heuristically (OSBS needs to guess what might be a pullspec). See *heuristic annotations* below.

Heuristic annotations

OSBS will attempt to extract all pullspecs from all attributes of each `metadata.annotations` object in a CSV. If an attribute contains more than one pullspec (as text, e.g. comma-separated), all of them should be found. Here is how OSBS implements this. One important thing to note is that only pullspecs conforming to a relatively strict format will be found this way:

`registry/namespace*/repo:tag` or `registry/namespace*/repo@sha256:digest`

Any number of namespaces, including 0, is valid. Registry must contain at least one dot: `registry.io` is valid but `localhost` is not. Digest, if present, must be exactly 64 base16 characters. Tag or digest *must* be specified, implicit `latest` tag is not supported. **example.clusterserviceversion.yaml**

```
kind: ClusterServiceVersion
metadata:
  annotations:
    containerImage: registry.io/namespace/foo  # [1]
    foobar: registry.io/foobar:latest, registry.io/ham/jam@sha256:... # [5]
spec:
  install:
    spec:
      deployments:
      - spec:
          template:
            metadata:
              annotations:
                containerImage: registry.io/namespace/bar  # [1]
            spec:
              containers:
              - name: baz
                image: registry.io/namespace/baz  # [2]
                env:
                - name: RELATED_IMAGE_SPAM
                  value: registry.io/namespace/spam  # [4]
              initContainers:
              - name: eggs
                image: registry.io/namespace/eggs  # [3]
```

Creating the relatedImages section

Each entry in the `.spec.relatedImages` section needs a `name` and an `image` - image being simply the pullspec itself. The name is constructed differently depending on where the pullspec was found.

annotations *[1], [5]* Take repo name and tag/digest from pullspec, add "-annotation"

E.g.

- `registry.io/foo:v1.1` -> `foo-v1.1-annotation`
- `registry.io/foo@sha256:123abc...` -> `foo-123abc...-annotation`

containers *[2]* Reuse original name, e.g. `{name: baz, image: ...}` -> `baz`

initContainers *[3]* Same as containers

RELATED_IMAGE env vars *[4]* Take name of variable, remove `RELATED_IMAGE_` prefix, convert to lowercase

E.g. `RELATED_IMAGE_SPAM` -> `spam`

The final `relatedImages` section of the example file would look like this:

```
spec:
  relatedImages:
  - name: foo-<digest>-annotation
    image: registry.io/namespace/foo@sha256:...
  - name: bar-<digest>-annotation
    image: registry.io/namespace/bar@sha256:...
  - name: baz
    image: registry.io/namespace/baz@sha256:...
  - name: eggs
    image: registry.io/namespace/eggs@sha256:...
  - name: spam
    image: registry.io/namespace/spam@sha256:...
  - name: jam-<digest>-annotation
    image: registry.io/ham/jam@sha256:...
  - name: foobar-<digest>-annotation
    image: registry.io/foobar@sha256:...
```

OSBS makes no guarantees about the order in which relatedImages will be added.

If 2 or more entries with the same name are found, then their images must also match, otherwise this is a conflict and build will fail. This is especially important for annotations, where the name is a combination of repo and tag. Using 2 images with the same repo and tag but different registry/namespace is not allowed.

Skip all processing for operator bundles

When option `skip_all` is enabled in `container.yaml`, it will enforce skip processing of operator bundles.

```
operator_manifests:
  # Relative path to your manifests dir from root of repo
  manifests_dir: my-manifests-dir
  skip_all: true
```

OSBS won't do any changes to operator's CSV file, but will still perform some basic checks like:

1. check that only 1 CSV file exists
2. check that `relatedImages` section exists, if there are any pullspecs defined in CSV file

`skip_all` option is allowed to use only for koji packages which are defined in atomic-reactor config in `skip_all_allow_list` in `operator_manifests` section. See configuration details in config.json.

Operator manifest appregistry builds

This type of build is for the older style of operator manifests targeting Openshift 4.3 or lower. It is identified by the `com.redhat.delivery.appregistry` label.

After a successful build, if *OMPS integration* is enabled, operator manifests are uploaded into configured application registry and namespace.

Details on how operator manifest can be accessed from the application registry are stored in koji build, in section `build.extra.operator_manifests.appregistry`.

Manifests will not be pushed to the application registry for scratch builds, isolated builds, or re-builds to prevent unwanted changes

1.3 Inspecting built image components

It is possible to inspect OSBS built image contents from within the image container.

In addition to being able to do so with the package manager available in the image, if any, e.g., RPM through `rpm -qa` to list all the packages installed in the image, OSBS also makes sure the following artifacts are shipped within the image

1.3.1 Dockerfiles

The Dockerfiles used to build the current container image and its parents, which is located in the `/root/buildinfo` directory.

Note that this is not necessarily the same Dockerfile provided by the user in the dist-git repository. OSBS makes changes to the Dockerfile and some of these changes may appear in these files whenever relevant. For instance, the FROM instruction may show the parent image digest instead of the repository and tag information.

1.3.2 Image Content Manifests

Image Content Manifests are JSON files shipped in OSBS built images with additional information on the contents shipped in the image.

The Image Content Manifest file is layer specific, and is located under the `/root/buildinfo/content_manifests` directory. It is named after the image NVR, and it is validated against the JSON Schema that defines the Image Content Manifest.

Among the data available in the Image Content Manifest file (Check the JSON Schema for further information), most important are the `image_layer_index`, which point to the layer that introduced the components listed in that file, i.e., the most recent layer for that image, and:

Content Sets

The `content_sets` field lists the content sets listed in the git repository for the platform supported by the image. This attribute may differ for each different platform the image was built for.

See *Content Sets* for further reference.

Extra contents

The `image_contents` field lists the non-RPM contents fetched from Cachito (see *Fetching source code from external source using cachito*) that were used during the image build and that were made available in the image.

For additional information on how to navigate through these contents, refer to the Image Content Manifest JSON Schema.

1.4 Building Source Container Images

OSBS is able to build source container image from a particular koji build previously created by OSBS. To create a source container build you have to specify either koji N-V-R or build ID for the image build you want to create a source container image for.

When koji build is using lookaside cache, that may include all sort of things about which we can't get any information, in that case source container build will fail.

Under the hood the BSI project is used to generate source images from sources identified and collected by OSBS. Please note that BSI script must be available in the OSBS buildroot as *bsi* executable in *$PATH*.

Current limitations:

- only Source RPMs and sources fetched through *Cachito integration* are added into source container image
- only koji internal RPMs are supported

Support for other types of sources and external builds will be added in future.

1.4.1 Signing intent resolution

Resolution of signing intent is done in following order:

- signing intent specified from CLI params (koji, osbs-client),
- otherwise signing intent is taken from the original image build,
- if undefined then default signing intent from *odcs* configuration section in *reactor-config-map* is used.

If ODCS integration is disabled, unsigned packages are allowed by default.

1.4.2 Koji integration

Koji integration must be enabled for building source container images. Source container build requires metadata stored in koji builds and koji database of RPM builds which source container build uses to lookup for sources.

Source container builds uses different task type: *buildSourceContainer*.

Koji Build Metadata Integration

Source container build uses metadata from specified image build in the following manner:

- **name**: suffix *-source* is appended to original name (*ubi8-container* will be transformed to *ubi8-container-source*)
- **version**: value is the same as original image build
- **release**: a suffix *.X* is appended to original release value, where *X* is a sequential integer starting from 1 increased by OSBS for each source image rebuild.

For example, from N-V-R *ubi8-container-8.1-20* OSBS creates source container build *ubi8-container-source-8.1-20.1*.

The original image N-V-R is stored in *extra.image.sources_for_nvr* attribute in koji source container build metadata.

1.4.3 Building source container images using koji

Using a koji client CLI directly you have to specify git repo URL and branch:

```
koji source-container-build <target>  --koji-build-nvr=NVR --koji-build-id=ID
```

For a full list of options:

```
koji source-container-build --help
```

1.4.4 Building source container images using osbs-client

Please note that mainly `koji` and `fedpkg` commands should be used for building container images instead of direct `osbs-client` calls.

To execute build via osbs-client CLI use:

```
osbs build-source-container -c <component> -u <username> --sources-for-koji-build-
↪nvr=N-V-R --sources-for-koji-build-id=ID
```

To see full list of options execute:

```
osbs build-source-container --help
```

To see all osbs-client subcommands execute:

```
osbs --help
```

Please note that `osbs-client` must be configured properly using config file `/etc/osbs.conf`. Please refer to *osbs-client configuration section* for configuration examples.

1.5 Understanding the Build Process

OSBS creates an OpenShift BuildConfig to store the configuration details about how to run atomic-reactor and which git commit to build from. If there is already a BuildConfig for the image (and branch), it is updated. Afterwards, a Build is instantiated from the BuildConfig.

The default OpenShift BuildConfig runPolicy of "Serial" is used, meaning only one Build will run at a time for a given BuildConfig, and other Builds will remain queued and unprocessed until the running build finishes. This is preferable to "SerialLatestOnly", which cancels all but the most recent pending queued build, because build logs for any user-submitted build can be watched through to completion.

OSBS uses two types of OpenShift Build:

worker build for creating the container images

orchestrator build for creating and managing worker builds

The cluster which runs orchestrator builds is referred to here as the *orchestrator cluster*, and the cluster which runs worker builds is referred to here as the *worker cluster*.

Note that the orchestrator cluster itself may also be configured to accept worker builds, so the cluster may be both orchestrator and worker. Alternatively some sites may want separate clusters for these functions.

The orchestrator build makes use of *Server-side Configuration for atomic-reactor* to discover which worker clusters to direct builds to and whether/which node selector is required for each.

The separation of tasks between the orchestrator build and the worker build is called the "arrangement".

Note: Arrangement versions lower than 6 are no longer supported (but each new version typically builds on the previous one, so no functionality is lost).

1.5.1 Arrangement Version 6 (reactor_config_map)

The orchestrator build chooses worker clusters based on platform (multi-platform build) and creates worker builds in those clusters. Worker builds perform their tasks and produce an image for their platform. After all worker builds finish, the orchestrator build continues with its own tasks.

This allows automatically triggered rebuilds (where the orchestrator cluster has complete knowledge of the latest build configurations and their triggers).

In this arrangement version, environment parameters are provided by the **reactor_config**. The order of plugins is the same, but hard coded, or placeholder, environment parameters in **orchestrator_inner** and **worker_inner** json files change.

An osbs-client configuration option **reactor_config_map** is required to define the name of the `ConfigMap` object holding the **reactor_config**. This configuration option is mandatory for arrangement versions greater than or equal to 6. The existing osbs-client configuration **reactor_config_secret** is deprecated (for all arrangements).

Orchestrator

Steps performed by the orchestrator build are:

- Get the parent images without pulling in order to inspect environment variables (this is to allow substitution to be performed in the "release" label)
- Verify that parent images comes from a build that exists in Koji
- Resolve composes, integration with odcs. See *ODCS compose* for details.
- For base images, the add_filesystem plugin runs in the orchestrator build as well as the worker build. This is to create a single Koji "image-build" task to create filesystems for all required architectures.
- Supply a value for the "release" label if it is missing (this value is provided to the worker build)
- Reserve a build in Koji
- Apply labels supplied via build request parameter to the Dockerfile
- Parse server-side configuration for atomic-reactor in order to know which worker clusters may be considered
- Create worker builds on the configured clusters, and collect logs and status from them
- Fetch workers metadata (see *Metadata Fragment Storage*)
- The group_manifests plugin creates a manifest list object in the registry, grouping together the image manifests from the worker builds.
- Import the Koji build
- Push floating tags to the container registry for a *manifest list*
- Tag the Koji build
- Update this OpenShift Build with annotations about output, performance, errors, worker builds used, etc
- Send email notifications if required
- Perform any clean-up required

Worker

The orchestration step will create an OpenShift build for performing the worker build. Inside this, atomic-reactor will execute these steps as plugins:

- Get (or create) parent layers against which the Dockerfile will run
  - For base images, the add_filesystem plugin runs but does not create a Koji task. Instead the orchestrator build tells it which Koji task ID to stream the filesystem tar archive from. Each worker build only streams the filesystem tar archive for the architecture it is running on, and imports it as the initial image layer
  - For layered images, the FROM images are pulled from the source registry

- Supply a value for the "release" label if it is missing (provided by the orchestator build)
- Make various alterations to the Dockerfile
- Fetch any supplemental files required by the Dockerfile
- Build the container
- Squash the image so that only a single layer is added since the parent image
- Query the image to discover installed RPM packages (by running `rpm` inside it)
- Tag and push the image to the container registry
- Compress the docker image tar archive
- Upload image tar archive to Koji
- Update this OpenShift Build with annotations about output, performance, errors, etc
- Perform any clean-up required:
  - Remove the parent images which were fetched or created at the start

For more details on how the build system is configured as of Arrangement 6, consult the *Build Parameters* document.

As of October 2019, Pulp is no longer supported in Arrangement 6 for either worker or orchestrator builds.

1.5.2 Logging

Logs from worker builds is made available via the orchestrator build, and clients (including koji-containerbuild) are able to separate individual worker build logs out from that log stream using an osbs-client API method.

Multiplexing

In order to allow the client to de-multiplex logs containing a mixture of logs from an orchestrator build and from its worker builds, a special logging field, platform, is used. Within atomic-reactor all logging goes through a LoggerAdapter which adds this `platform` keyword to the `extra` dict passed into logging calls, resulting in log output like this:

```
2017-06-23 17:18:41,791 platform:- - atomic_reactor.foo - DEBUG - this is from the␣
↪orchestrator build
2017-06-23 17:18:41,791 platform:x86_64 - atomic_reactor.foo - INFO - 2017-06-23␣
↪17:18:41,400 platform:- atomic_reactor.foo - DEBUG - this is from a worker build
2017-06-23 17:18:41,791 platform:x86_64 - atomic_reactor.foo - INFO - continuation␣
↪line
```

Demultiplexing is possible using a the osbs-client API method, `get_orchestrator_build_logs`, a generator function that returns objects with these attributes:

platform str, platform name if worker build, else None

line str, log line (Unicode)

The "Example" section below demonstrates how to use the `get_orchestrator_build_logs()` method in Python to parse the above log lines.

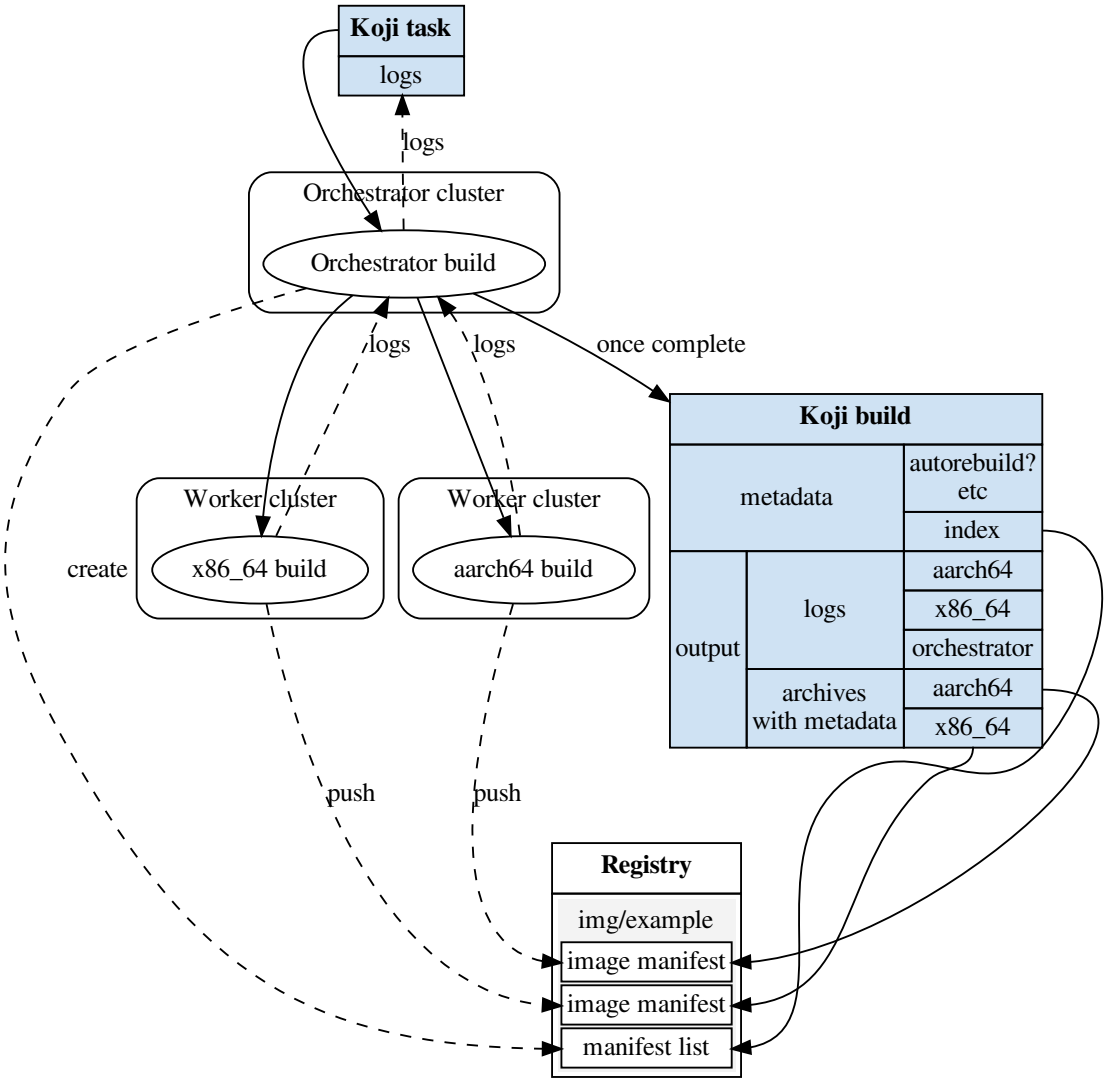

Fig. 1: Orchestrator and worker builds (arrangement v6)

Encoding issues

When retrieving logs from containers, the text encoding used is only known to the container. It may be based on environment variables within that container; it may be hard-coded; it may be influenced by some other factor. For this reason, container logs are treated as byte streams.

This applies to:

- containers used to construct the built image
- the builder image running atomic-reactor for a worker build
- the builder image running atomic-reactor for an orchestrator build

When retrieving logs from a build, OpenShift cannot say which encoding was used. However, atomic-reactor can define its own output encoding to be UTF-8. By doing this, all its log output will be in a known encoding, allowing osbs-client to decode it. To do this it should call `locale.setlocale(locale.LC_ALL, "")` and the Dockerfile used to create the builder image must set an appropriate environment variable:

```
ENV LC_ALL=en_US.UTF-8
```

Orchestrator builds want to retrieve logs from worker builds, then relay them via logging. By knowing that the builder image for the worker is the same as the builder image for the orchestrator, we also know the encoding for those logs to be UTF-8.

Example

Here is an example Python session demonstrating this interface:

```
>>> server = OSBS(...)
>>> logs = server.get_orchestrator_build_logs(...)
>>> [(item.platform, item.line) for item in logs]
[(None, '2017-06-23 17:18:41,791 platform:- - atomic_reactor.foo - DEBUG - this is␣
↪from the orchestrator build'),
 ('x86_64', '2017-06-23 17:18:41,400 atomic_reactor.foo - DEBUG - this is from a␣
↪worker build'),
 ('x86_64', 'continuation line')]
```

Note:

- the lines are (Unicode) string objects, not bytes objects
- the orchestrator build's logging fields have been removed from the worker build log line
- the "outer" orchestrator log fields have been removed from the worker build log line, and the `platform:-` field has also been removed from the worker build's log line
- where the worker build log line had no timestamp (perhaps the log line had an embedded newline, or was logged outside the adapter using a different format), the line was left alone

1.5.3 Metadata Fragment Storage

When creating a Koji Build using arrangement 3 and newer, the koji_import plugin needs to assemble Koji Build Metadata, including:

- components installed in each builder image (worker builds and orchestrator build)
- components installed in each built image

- information about each build host

To assist the orchestrator build in assembling this (JSON) data, the worker builds gather information about their build hosts, builder images, and built images. They then need to pass this data to the orchestrator build. After creating the Koji Build, the orchestrator build must then free any resources used in passing the data.

The method used for passing the data from the worker builds to the orchestrator build is to store it temporarily in a ConfigMap object in the worker cluster. Its name is stored in the OpenShift Build annotations for the worker build. To do this the worker cluster's "builder" service account needs permission to create ConfigMap objects.

The orchestrator build collects the metadata fragments and assembles them, together with the platform-neutral metadata, in the koji_import plugin.

The orchestrator build is then responsible for removing the OpenShift ConfigMap from the worker cluster. To do this, the worker cluster's "orchestrator" service account needs permission to get and delete ConfigMap objects.

1.5.4 Autorebuilds

OSBS's autorebuild feature automatically starts new builds of layered images whenever the base parent image changes. This is particularly useful for image owners that maintain a large hierarchy of images, which would otherwise require manually starting each image build in the correct order. Instead, image owners can start a build for the topmost ancestor which upon completion triggers the next level of layered images, and so on.

Builds may opt in to autorebuilds with an *autorebuild entry in the dist-git configuration.* Additional options for autorebuilds can be configured in *container.yaml.*

1.6 Build Parameters

1.6.1 Environment vs User Parameters

atomic-reactor requires various parameters to build container images. These range from the Koji Hub URL to git commit. We can categorize them into environment and user parameters. The main difference between them is how they are reused. Environment parameters are shared by different build requests, while user parameters are unique to each build request.

Environment parameters are used for configuring usage of external services such as Koji, ODCS, SMTP, etc. They are also used for controlling some aspects of the container images built, for example, distribution scope, vendor, authoritative-registry, etc.

User parameters contain the unique information for a user's build request: git repository, git branch, Koji target, etc. These should be reused for autorebuilds and are not affected by environment changes.

1.6.2 Reactor Configuration

As of Arrangement 6, environment configuration is provided to build containers in a way that is less coupled to the `Build`/`BuildConfig` objects. The pre-build plugin `reactor_config` determines and provides all environment configuration to other plugins.

Previously, the value of `reactor_config` was mounted into container as a secret. With Arrangement 6 it is supplied as a `ConfigMap` like:

```
apiVersion: v1
kind: ConfigMap
```

(continues on next page)

(continued from previous page)

```
data:
    "config.yaml": <encoded yaml>
```

For an orchestrator build, the `ConfigMap` is mapped via the downward API into the **REACTOR_CONFIG** environment variable in a build container. When the `BuildConfig` is instantiated, OpenShift selects the `ConfigMap` with name given in **reactor-config-map**, retrieves the contents of the key **config.yaml**, and sets it as the value of the **REACTOR_CONFIG** environment variable in the build container.

For worker builds, the **REACTOR_CONFIG** environment variable is defined as an inline **value** (via the **reactor_config_override** build parameter in osbs-client). To populate this parameter, the `orchestrate_build` plugin uses the `reactor_config` plugin to read the reactor configuration for the orchestrator build, using it as the basis of the reactor configuration for worker builds with the following modifications:

- **openshift** section is replaced with worker specific values. These values can be read from the osbs-client `Configuration` object created for each worker cluster.
- **worker_token_secrets** is completely removed. This section is intended for orchestrator builds only.

The schema definition config.json in atomic-reactor contains a description for each property.

Example of **REACTOR_CONFIG**:

```
version: 1

clusters:
    x86_64:
    - name: x86_64-worker-1
      max_concurrent_builds: 15
      enabled: True
    - name: x86_64-worker-2
      max_concurrent_builds: 6
      enabled: True

clusters_client_config_dir: /var/run/secrets/atomic-reactor/client-config-secret

koji:
    hub_url: https://koji.example.com/hub
    root_url: https://koji.example.com/root
    auth:
        ssl_certs_dir: /var/run/secrets/atomic-reactor/kojisecret
    use_fast_upload: false

pulp:
    name: my-pulp
    auth:
        ssl_certs_dir: /var/run/secrets/atomic-reactor/pulpsecret

odcs:
    api_url: https://odcs.example.com/api/1
    auth:
        ssl_certs_dir: /var/run/secrets/atomic-reactor/odcssecret
    signing_intents:
    - keys: ['R123', 'R234']
      name: release
    - keys: ['B123', 'B234', 'R123', 'R234']
      name: beta
    - keys: []
      name: unsigned
```

(continues on next page)

(continued from previous page)

```
    default_signing_intent: release

smtp:
    host: smtp.example.com
    from_address: osbs@example.com
    error_addresses:
    - support@example.com
    domain: example.com
    send_to_submitter: True
    send_to_pkg_owner: True

arrangement_version: 6

artifacts_allowed_domains:
- download.example.com/released
- download.example.com/candidates

image_labels:
    vendor: "Spam, Inc."
    authoritative-source-url: registry.public.example.com
    distribution-scope: public

image_equal_labels:
- [description, io.k8s.description]

openshift:
    url: https://openshift.example.com
    auth:
        enable: True
    build_json_dir: /usr/share/osbs/

group_manifests: False

platform_descriptors:
- platform: x86_64
  architecture: amd64

content_versions:
- v2

# Output registries (built images are pushed here)
registries:
- url: https://container-registry.example.com/v2
  auth:
    cfg_path: /var/run/secrets/atomic-reactor/v2-registry-dockercfg

# Default source registry (base images are pulled from here)
source_registry:
    url: https://registry.private.example.com

# Additional source registries
pull_registries:
- url: https://registry.public.example.com
  auth:
    cfg_path: /var/run/secrets/atomic-reactor/registries-secret

sources_command: "fedpkg sources"
```

(continues on next page)

(continued from previous page)

```
required_secrets:
- kojisecret
- pulpsecret
- odcssecret
- v2-registry-dockercfg
- client-config-secret

worker_token_secrets:
- x86-64-worker-1
- x86-64-worker-2

default_image_build_method: imagebuilder

skip_koji_check_for_base_image: False

build_env_vars:
- name: HTTP_PROXY
  value: "http://proxy.example.com"
- name: HTTPS_PROXY
  value: "https://proxy.example.com"
- name: NO_PROXY
  value: localhost,127.0.0.1
```

1.6.3 Atomic Reactor Plugins and Arrangement Version 6

Prior to Arrangement 6, atomic-reactor plugins received environment parameters as their own plugin parameters. Arrangement 6 was introduced to indicate that plugins should retrieve environment parameters from **reactor_config** instead. Plugin parameters that are really environment parameters have been made optional.

The osbs-client configuration **reactor_config_map** defines the name of the `ConfigMap` object holding **reactor_config**. This configuration option is mandatory for arrangement versions greater than or equal to 6. Previous osbs-client configuration **reactor_config_secret** is deprecated.

An osbs-client build parameter **reactor_config_override** allows reactor configuration to be passed in as a python dict. It is also validated against config.json schema. When both **reactor_config_map** and **reactor_config_override** are defined, **reactor_config_override** takes precedence. NOTE: **reactor_config_override** is a python dict, not a string of serialized data.

1.6.4 Creating Builds

osbs-client no longer renders the atomic-reactor plugin configuration at `Build` creation. Instead, the **USER_PARAMS** environment variable is set on the `Build` containing only user parameters as JSON. For example:

```
{
    "build_type": "orchestrator",
    "git_branch": "my-git-branch",
    "git_ref": "abc12",
    "git_uri": "git://git.example.com/spam.git",
    "is_auto": False,
    "isolated": False,
    "koji_task_id": "123456",
```

(continues on next page)

(continued from previous page)

```
    "platforms": ["x86_64"],
    "scratch": False,
    "target": "my-koji-target",
    "user": "lcarva",
    "yum_repourls": ["http://yum.example.com/spam.repo", "http://yum.example.com/
↪bacon.repo"],
}
```

1.6.5 Rendering Plugins

Once the build is started, control is handed over to atomic-reactor. Its input plugin `osv3` looks for the environment variable **USER_PARAMS** and uses the osbs-client method `render_plugins_configuration` to generate the plugin configuration on the fly. The generated plugin configuration contains the order in which plugins will run as well as user parameters.

1.6.6 Secrets

Because the plugin configuration renders at build time (after `Build` object is created), we cannot select which secrets to mount in container build based on which plugins have been enabled. Instead, all the secrets that may be needed must be mounted. The **reactor_config** `ConfigMap` defines the full set of secrets it needs via its **required_secrets** list.

When orchestrator build starts worker builds, it uses the same set of secrets. This requires worker clusters to have the same set of secrets available. For example, if **reactor_config** defines:

```
required_secrets:
- kojisecret
- pulpsecret
```

A secret named **kojisecret** must be available in orchestrator and worker clusters. The worker and orchestrator versions don't need to have the same value. For instance, worker and orchestrator builds may use different authentication certificates.

Secrets needed for communication from orchestrator build to worker clusters are defined separately in **worker_token_secrets**. These are not passed along to worker builds.

1.6.7 Site Customization

The site customization configuration file is no longer read from the system creating the OpenShift `Build` (usually koji builder). Instead, this customization file must be stored and read from inside the builder image.

1.7 Contributing to OSBS

1.7.1 Setting up a (local) development environment

OSBS-Box aims to provide a convenient way to set up an entire OSBS environment on your machine or a VM.

You will first need to set up an OpenShift Origin cluster. OSBS-Box *might* work with other OpenShift clusters as long as you are able to log in as `system:admin`.

Afterwards, refer to the OSBS-Box readme for further instructions on deployment, usage etc.

OSBS-Box environment

osbs-koji

Containerized deployment of Koji. Koji-builder uses osbs-client, koji-client, koji-hub and koji-builder use the koji-containerbuild cli, hub and builder plugins respectively.

Use the koji-client container to run container builds.

osbs-orchestrator

Main OSBS namespace, contains reactor-config-map and buildroot.

osbs-buildroot: Image inside the orchestrator namespace, does most of the container-building work. Uses atomic-reactor, osbs-client and dockerfile-parse.

osbs-worker

Worker namespace for the x86_64 architecture (OSBS-Box does not support other arches).

osbs-registry

Container registry service. Push base images for your builds to the registry, pull the built images from it.

1.7.2 Contribution guidelines

Submitting changes

Changes are accepted through pull requests. If you want to implement major new functionality, we'd like to ask you to open a bug with proposal to discuss it with us. We're nice folks and we don't bite – we just want to see what you're up to and a chance to give some suggestions to save your time as well as ours.

Please create your feature branch from the master branch. Make sure to add unit tests under the tests/ subdirectory (we use py.test and flexmock for this). When you push new commits, tests will be triggered to be run in Travis CI and results will be shown in your pull request. You can also run them locally from the top directory (py.test tests). You can also use the test.sh script to run these tests in a container.

Follow the PEP8 coding style. This project allows 99 characters per line.

Please make sure each commit is for a complete logical change, and has a useful commit message. When making changes in response to suggestions, it is fine to make new commits for them but please make sure to squash them into the relevant "logical change" commit before the pull request is merged.

Before a pull request is approved it must meet these criteria:

- Commit messages are descriptive enough
- "Signed-off-by:" line is present in each commit
- Unit tests pass
- Code coverage from testing does not decrease and new code is covered
- JSON/YAML configuration changes are updated in the relevant schema
- Changes to metadata also update the documentation for the metadata

- Pull request includes link to an osbs-docs PR for user documentation updates
- New feature can be disabled from a configuration file

Once it is approved by two developer team members someone from the team will merge it. To avoid creating merge commits the pull request will be rebased during the merge.

Licensing

This project is licensed using the BSD-3-Clause license. When submitting pull requests please make sure your commit messages include a signed-off-by line to certify the below text:

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

You can do this by using git commit –signoff when committing changes, making sure that your real name is included.

Code policies

Retry on error

Requests to a remote service should be retried if the failure is not definitely permanent.

In most cases there should be a delay before retrying. The only reason to retry immediately is when the request is likely to succeed when immediately retried (i.e. the cause is known). An example would be retrying a PUT operation which was rejected due to conflicts.

There should be a maximum number of attempts before giving up and reporting an exception.

The delay and the maximum number of attempts should both be configurable if feasible. If not explicitly configured and an HTTP request to be retried included a Retry-After header in the response, the Retry-After header should be used to set the delay time.

When alternate remote services are available, retries should only be attempted if no alternates succeed. For example, when selecting a worker cluster a failure should immediately result in the next worker cluster being tried. If there

is no such cluster, or all configured worker clusters fail, retries should be attempted, including a delay if appropriate (depending on the failure).

There should only be a single "level" of retry logic, unless there is a good reason. For example, if dockpulp implements retry logic for one of its operations, atomic-reactor should not retry that operation (unless it makes sense to). In other words: be aware of which calls implicitly retry.

1.8 Documentation of the OSBS tools

1.8.1 Koji Containerbuild

Extends Koji buildsystem with plugins to allow building containers via OpenShift.

Provides plugin to submit builds with koji CLI.

- Installation and usage instructions
- General concepts behind container build architecture

1.8.2 Atomic Reactor

Python library with command line interface for building docker images.

- Installation and Usage Instructions
- Building Base Images
- Plugins
- build.json

1.8.3 OSBS Client

Python module and command line client for OpenShift Build Service.

- Basic configuration and usage
- Description of the Build Process
- Configuration file

1.8.4 Ansible Playbook

- Ansible playbook for deploying OpenShift Build Service

1.8.5 Deploying and Running OSBS

- Deploying OpenShift Build System
- OSBS Backups
- Local Development
- Resource limiting

CHAPTER 2

Indices and tables

- genindex
- search