End-to-End Tests in OpenShift

Bruno Barcarol Guimarães

2022-09-09

End-to-End Tests in OpenShift

End-to-End Tests in OpenShift

Bruno Barcarol Guimarães

Introduction

Test types

Releases

Input images

Image pipeline

Red Hat

2022-09-09

End-to-End Tests in OpenShift

Overview

Overview

Introduction

Test types

Releases

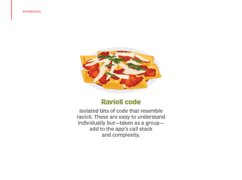
Input images

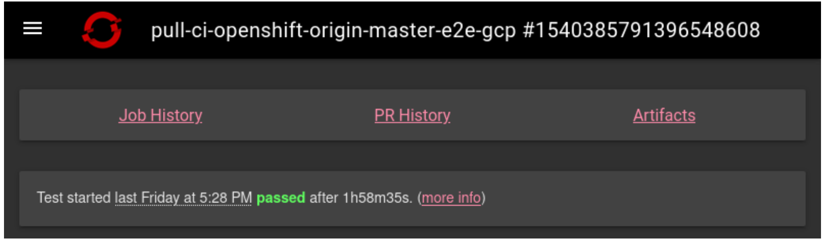
Image pipeline

Ravioli code

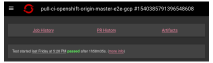
Isolated bits of code that resemble ravioli. These are easy to understand individually but—taken as a group—add to the app's call stack and complexity.

Red Hat





https://prow.ci.openshift.org/view/gs/origin-ci-test/pr-logs/pull/27275/pull-ci-openshift-origin-master-e2e-gcp/1540385791396548608



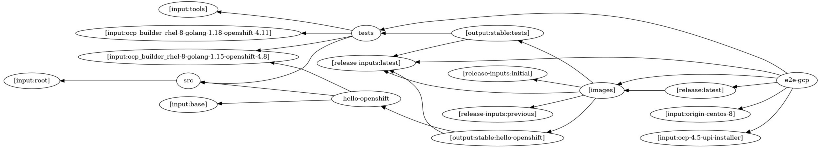
We will revisit today the example job we looked at in the last third of the previous presentation, now in much more detail.

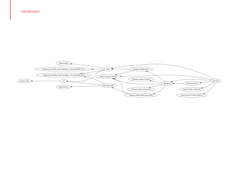
https://gcsweb-ci.apps.ci.l2s4.p1.openshiftapps.com/gcs/origin-ci-test/pr-logs/pull/27275/
pull-ci-openshift-origin-master-e2e-gcp/1540385791396548608/prowjob.json*

```
command:
- ci-operator
args:
- --gcs-upload-secret=/secrets/gcs/service-account.json
- --image-import-pull-secret=/etc/pull-secret/.dockerconfigjson
- --lease-server-credentials-file=/etc/boskos/credentials
- --report-credentials-file=/etc/report/credentials
- --secret-dir=/secrets/ci-pull-credentials
- --secret-dir=/usr/local/e2e-gcp-cluster-profile
- --target=e2e-gcp
```

* *how-tos: document artifacts directory #266* – `openshift/ci-docs`

As a reminder, this will (via `prowgen`) result in a `ProwJob` which will execute `ci-operator` targeting the single test name `e2e-gcp`, declared in its configuration file (obtained from the `configresolver`).





And this will be done by constructing and execting this step graph. See the previous presentation for a reminder of how all of this generally works.

https://github.com/openshift/release/blob/master/ci-operator/config/openshift/origin/openshift-origin-master.yaml

```yaml
tests:
- as: e2e-gcp
  steps:
    cluster_profile: gcp-openshift-gce-devel-ci-2
    workflow: openshift-e2e-gcp-loki
```


Red Hat

Introduction
https://github.com/openshift/release/blob/master/ci-operator/config/openshift/origin/openshift-origin-master.yaml
tests:
- as: e2e-gcp
  steps:
    cluster_profile: gcp-openshift-gce-devel-ci-2
    workflow: openshift-e2e-gcp-loki

It all starts with this innocent test definition in the configuration file...

Test types

Red Hat

Test types

(we have fancy section title slides now) ibid

```
- as: verify-deps
  commands: make verify-deps …
  container:
    from: src
```

```
…/openshift-origin-release-3.11.yaml
```

```
- as: e2e-gcp
  commands: … run-tests
  openshift_ansible:
    cluster_profile: gcp
```


Red Hat

Test types
ibid
- as: verify-deps
commands: make verify-deps …
container:
from: src
…/openshift-origin-release-3.11.yaml
- as: e2e-gcp
commands: … run-tests
openshift_ansible:
cluster_profile: gcp

The `test` entries in the configuration file have many different forms, although they have many similarities. `steps` (or, sometimes, `literal_steps`, as in the previous example), denotes a *multi-stage* test. Other basic test types are:

- simple *container* tests, declared with `container`
- a large variety of *template* tests, declared with fields in the form `openshift_*`

https://github.com/openshift/ci-tools/blob/master/pkg/api/types.go

```
type TestStepConfiguration struct {
    As string `json:"as"`
    Commands string `json:"commands,omitempty"`
    // …
    // Only one of the following can be not-null.
    ContainerTestConfiguration                         …
    MultiStageTestConfiguration                        …
    MultiStageTestConfigurationLiteral                 …
    OpenshiftAnsibleClusterTestConfiguration           …
    OpenshiftAnsibleSrcClusterTestConfiguration        …
    OpenshiftAnsibleCustomClusterTestConfiguration     …
    OpenshiftInstallerClusterTestConfiguration         …
    OpenshiftInstallerUPIClusterTestConfiguration      …
    OpenshiftInstallerUPISrcClusterTestConfiguration   …
    OpenshiftInstallerCustomTestImageClusterTestConfiguration …
}
```


Red Hat

This is manifested in code in the `TestStepConfiguration` structure (not to be confused with the `TestStep` structure, used in multi-stage tests), which uses the common pattern of many (optional) pointers to other structures, only one of which is ever non-null (a *sum type*).

```
// Only one of the following can be not-null.
ContainerTestConfiguration \
    *ContainerTestConfiguration \
    `json:"container,omitempty"`
// …
```


Test types / container

// Only one of the following can be not-null.
ContainerTestConfiguration \
    *ContainerTestConfiguration \
    `json:"container,omitempty"`
// …

(these identifiers are enormous, so here is what a full line looks like)

```
type ContainerTestConfiguration struct {
    From PipelineImageStreamTagReference
    MemoryBackedVolume *MemoryBackedVolume
    Clone *bool
}
```


Red Hat

2022-09-09

End-to-End Tests in OpenShift
└─Test types
  └─Container
    └─Test types / container

Test types / container

type ContainerTestConfiguration struct {
From PipelineImageStreamTagReference
MemoryBackedVolume *MemoryBackedVolume
Clone *bool
}

Starting with container tests, their structure is deceptively simple. It declares its container image plus a couple of other, more esoteric fields.

```
type TestStepConfiguration struct {
    As string
    Commands string
    Cluster Cluster
    Secret *Secret
    Secrets []*Secret
    Cron *string
    Interval *string
    ReleaseController bool
    Postsubmit bool
    ClusterClaim *ClusterClaim
    RunIfChanged string
    Optional bool
    SkipIfOnlyChanged string
    Timeout *prowv1.Duration
    // …
```


Red Hat

Test types / container

```
type TestStepConfiguration struct {
    As string
    Commands string
    Cluster Cluster
    Secret *Secret
    Secrets []*Secret
    Cron *string
    Interval *string
    ReleaseController bool
    Postsubmit bool
    ClusterClaim *ClusterClaim
    RunIfChanged string
    Optional bool
    SkipIfOnlyChanged string
    Timeout *prowv1.Duration
    // …
```


This is because most of the fields live in the original structure, previously abbreviated. The list of fields here is somewhat unruly. In the past, we had a very relaxed policy for external contributions, so the code base — and this area in particular — grew very "organically" (to put it favorably).
Some of these, such as the build cluster, the periodic/post-submit fields, etc. are still useful. Some are obsolete and kept for compatibility.
As an aside, the capabilities of container tests are roughly a subset of those of multi-stage, there is a long-term plan to unify their underlying implementation.

```
- as: e2e-gcp
  commands: … run-tests
  openshift_ansible:
    cluster_profile: gcp
```

```
args:
- --image-import-pull-secret=/etc/pull-secret/.dockerconfigjson
- --report-credentials-file=/etc/report/credentials
- --secret-dir=/usr/local/e2e-gcp-periodic-cluster-profile
- --target=e2e-gcp-periodic
- --template=/usr/local/e2e-gcp-periodic
- --gcs-upload-secret=/secrets/gcs/service-account.json
command:
- ci-operator
```


Red Hat

Test types / template
- as: e2e-gcp
commands: … run-tests
openshift_ansible:
cluster_profile: gcp
args:
- --image-import-pull-secret=/etc/pull-secret/.dockerconfigjson
- --report-credentials-file=/etc/report/credentials
- --secret-dir=/usr/local/e2e-gcp-periodic-cluster-profile
- --target=e2e-gcp-periodic
- --template=/usr/local/e2e-gcp-periodic
- --gcs-upload-secret=/secrets/gcs/service-account.json
command:
- ci-operator

The second type of test (also in chronological order) is everyone's favorite: template tests. This was the first mechanism added to `ci-operator` to support end-to-end tests, or in general anything more complex than a container test.

They are mostly a historical curiosity at this point, used only in very old, 3.11 jobs, but they provide some context to some of the more dubious aspects of `ci-operator`.

There is no (with one exception due to a failed plan) corresponding test definition in the configuration file for these tests: the entry in `tests` is used exclusively by `prowgen`. Instead, the definition is supplied at runtime via the `--template` argument.

```yaml
  volumeMounts:
  - mountPath: /usr/local/e2e-gcp-periodic
    name: job-definition
    subPath: cluster-launch-e2e.yaml
volumes:
- configMap:
    name: prow-job-cluster-launch-e2e
  name: job-definition
```


Red Hat

Test types / template

```yaml
  volumeMounts:
  - mountPath: /usr/local/e2e-gcp-periodic
    name: job-definition
    subPath: cluster-launch-e2e.yaml
volumes:
- configMap:
    name: prow-job-cluster-launch-e2e
  name: job-definition
```


In our Prow jobs, this is done by mounting the definition via a `ConfigMap`...

```
https://github.com/openshift/release/tree/master/
ci-operator/templates
```

```
ci-operator/templates/
    master-sidecar-3.yaml
    master-sidecar-4.4.yaml
    openshift/
        installer/
            cluster-launch-installer-custom-test-image.yaml
            cluster-launch-installer-e2e.yaml
            cluster-launch-installer-libvirt-e2e.yaml
            cluster-launch-installer-metal-e2e.yaml
            cluster-launch-installer-openstack-e2e.yaml
            cluster-launch-installer-openstack-upi-e2e.yaml
            cluster-launch-installer-src.yaml
            cluster-launch-installer-upi-e2e.yaml
        openshift-ansible/
            cluster-launch-e2e-openshift-ansible.yaml
            cluster-launch-e2e.yaml
            cluster-scaleup-e2e-40.yaml
```


Red Hat

...which in turn are populated via `updateconfig` from the files in the dreaded `ci-operator/templates` directory in `openshift/release`.

.../openshift/installer/cluster-launch-installer-e2e.yaml

```yaml
kind: Template
apiVersion: template.openshift.io/v1

parameters:
- name: JOB_NAME
  required: true
- name: JOB_NAME_SAFE
  required: true
# ...
```


Red Hat

Test types / template

.../openshift/installer/cluster-launch-installer-e2e.yaml

kind: Template
apiVersion: template.openshift.io/v1

parameters:
- name: JOB_NAME
  required: true
- name: JOB_NAME_SAFE
  required: true
...

Each of these files is an OpenShift `Template` object, which consists of a list of parameters (strings, essentially)...

```yaml
objects:
# We want the cluster to be able to access
# these images
- kind: RoleBinding
  apiVersion: authorization.openshift.io/v1
  metadata:
    name: ${JOB_NAME_SAFE}-image-puller
    namespace: ${NAMESPACE}
  # …
```


Test types / template
objects:
We want the cluster to be able to access
these images
- kind: RoleBinding
  apiVersion: authorization.openshift.io/v1
  metadata:
    name: ${JOB_NAME_SAFE}-image-puller
    namespace: ${NAMESPACE}
  # …

…and a list of objects. ${…} strings are replaced by parameter values when the template is instantiated (and good luck telling what is bash interpolation and what is template substitution in a complex Pod definition).

```
# The e2e pod spins up a cluster, runs e2e tests,
# and then cleans up the cluster.
- kind: Pod
  apiVersion: v1
  metadata:
    name: ${JOB_NAME_SAFE}
    namespace: ${NAMESPACE}
  # …
```


Red Hat

Test types / template

The e2e pod spins up a cluster, runs e2e tests,
and then cleans up the cluster.
- kind: Pod
apiVersion: v1
metadata:
name: ${JOB_NAME_SAFE}
namespace: ${NAMESPACE}
…

And that is a summary of the entirety of the capabilities provided by template tests. From there, users would create a Pod definition (n.b.: a single one) to execute their test using colossal, inline shell scripts.

```
containers:
# Once the cluster is up, executes shared tests
- name: test
# …
# Runs an install
- name: setup
# …
# Performs cleanup of all created resources
- name: teardown
# …
```


Red Hat

Test types / template
containers:
Once the cluster is up, executes shared tests
- name: test
…
Runs an install
- name: setup
…
Performs cleanup of all created resources
- name: teardown
…

In practice, a few templates were developed and used by most tests, all following roughly this structure, later mirrored in multi-stage tests: a `setup` container performed the cluster installation, a `test` container executed OpenShift or repository tests, and a `teardown` container destroyed the temporary cluster.

```
parameters:
# …
- name: IMAGE_FORMAT
- name: IMAGE_INSTALLER
  required: true
- name: IMAGE_TESTS
  required: true
# …
- name: RELEASE_IMAGE_LATEST
  required: true
# …
```


Red Hat

Test types / template
parameters:
…
- name: IMAGE_FORMAT
- name: IMAGE_INSTALLER
  required: true
- name: IMAGE_TESTS
  required: true
…
- name: RELEASE_IMAGE_LATEST
  required: true
…

Configuration and parameterization was done via these template parameters, some of which are treated especially by `ci-operator`:

- `IMAGE_FORMAT` is populated with the public registry *pull spec* for built images.
- `IMAGE_*` entries are populated with entries from the input release stream.
- `RELEASE_IMAGE_*` entries are populated with the release payload *pull spec*.

The presence of each of these variables also causes the template step to depend on the step which provides it (the `Provides` method in each step type). Environment variables can also be used to initialize or override these values, which is still used in some of our E2E tests, even in multi-stage (e.g. the release controller uses RELEASE_IMAGE_LATEST to override the input release payload).

Test types / multi-stage

- test definition
- test phases
  - pre
  - test
  - post
- step registry
  - references
  - chains
  - workflows
  - observers

- container image
- parameters
- dependencies
- credentials
- leases
- overriding
- …

Test types / multi-stage
test definition
test phases
pre
test
post
step registry
references
chains
workflows
observers
container image
parameters
dependencies
credentials
leases
overriding
…

Of course, multi-stage tests are a universe of their own and worth (at least) a presentation in themselves. Here are some of the capabilities, most of which we will not have time to analyze today.

```
type MultiStageTestConfiguration struct {
    ClusterProfile ClusterProfile
    Pre []TestStep
    Test []TestStep
    Post []TestStep
    Workflow *string
    Environment TestEnvironment
    Dependencies TestDependencies
    DNSConfig *StepDNSConfig
    Leases []StepLease
    AllowSkipOnSuccess *bool
    AllowBestEffortPostSteps *bool
    Observers *Observers
    DependencyOverrides DependencyOverrides
}
```


Red Hat

Test types / multi-stage

type MultiStageTestConfiguration struct {
ClusterProfile ClusterProfile
Pre []TestStep
Test []TestStep
Post []TestStep
Workflow *string
Environment TestEnvironment
Dependencies TestDependencies
DNSConfig *StepDNSConfig
Leases []StepLease
AllowSkipOnSuccess *bool
AllowBestEffortPostSteps *bool
Observers *Observers
DependencyOverrides DependencyOverrides
}

Two structures, which share most of their fields, are involved in the configuration of multi-stage tests. `MultiStageTestConfiguration` is loaded directly from the `steps` field. It represents a user test definition which potentially needs to go through *resolution*, where references to steps, chains, and workflows from the step registry have to be replaced with their defintions.

The `--unresolved-config` argument and the `UNRESOLVED_CONFIG` variables correspond to this structure.

```go
type MultiStageTestConfigurationLiteral struct {
    ClusterProfile ClusterProfile
    Pre []LiteralTestStep
    Test []LiteralTestStep
    Post []LiteralTestStep
    Environment TestEnvironment
    Dependencies TestDependencies
    DNSConfig *StepDNSConfig
    Leases []StepLease
    AllowSkipOnSuccess *bool
    AllowBestEffortPostSteps *bool
    Observers []Observer
    DependencyOverrides DependencyOverrides
    Timeout *prowv1.Duration
}
```


Red Hat

Test types / multi-stage

type MultiStageTestConfigurationLiteral struct {
    ClusterProfile ClusterProfile
    Pre []LiteralTestStep
    Test []LiteralTestStep
    Post []LiteralTestStep
    Environment TestEnvironment
    Dependencies TestDependencies
    DNSConfig *StepDNSConfig
    Leases []StepLease
    AllowSkipOnSuccess *bool
    AllowBestEffortPostSteps *bool
    Observers []Observer
    DependencyOverrides DependencyOverrides
    Timeout *prowv1.Duration
}

Its counterpart is `MultiStageTestConfigurationLiteral`, which represents a *resolved* configuration, and corresponds to the `--config` argument and the `CONFIG_SPEC` variable.

```
type LiteralTestStep struct {
    As string
    From string
    FromImage *ImageStreamTagReference
    Commands string
    Resources ResourceRequirements
    Timeout *prowv1.Duration
    GracePeriod *prowv1.Duration
    Credentials []CredentialReference
    Environment []StepParameter
    Dependencies []StepDependency
```


Red Hat

Test types / multi-stage
type LiteralTestStep struct {
As string
From string
FromImage *ImageStreamTagReference
Commands string
Resources ResourceRequirements
Timeout *prowv1.Duration
GracePeriod *prowv1.Duration
Credentials []CredentialReference
Environment []StepParameter
Dependencies []StepDependency

This distinction is also reflected in the `LiteralTestStep` structure, lists of which compose the input configuration…

```
    DNSConfig *StepDNSConfig
    Leases []StepLease
    OptionalOnSuccess *bool
    BestEffort *bool
    Cli string
    Observers []string
    RunAsScript *bool
}
```


Test types / multi-stage
DNSConfig *StepDNSConfig
Leases []StepLease
OptionalOnSuccess *bool
BestEffort *bool
Cli string
Observers []string
RunAsScript *bool
}

```
type TestStep struct {
    *LiteralTestStep
    Reference *string
    Chain *string
}
```


Red Hat

type TestStep struct {
    *LiteralTestStep
    Reference *string
    Chain *string
}

...and the `TestStep` structure, which has the same contents but has also the option of referring to an external definition from the registry.

```
tests:
- as: multi-stage
  steps: # …
- as: multi-stage-literal
  literal_steps: # …

$ JOB_SPEC=… ci-operator
$ ci-operator --config …
$ ci-operator --unresolved-config …
$ CONFIG_SPEC=… ci-operator …
$ UNRESOLVED_CONFIG=… ci-operator …
```


Red Hat

These two types exist to distinguish the two states in code and between services, e.g.:

- regular jobs receive a literal configuration from the resolver
- `pj-rehearse` loads the unresolved configuration and expands it itself based on the PR contents, setting `$UNRESOLVED_CONFIG`
- release jobs provide their own inline configuration via `$CONFIG_SPEC` or `$UNRSOLVED_CONFIG` depending on the case
- etc.

Releases

Releases

https://github.com/openshift/ci-tools/blob/master/pkg/api/types.go

```
type ReleaseBuildConfiguration struct {
    Metadata Metadata
    InputConfiguration
    // …
}

type InputConfiguration struct {
    // …
    Releases map[string]UnresolvedRelease
}
```


Red Hat

Releases

https://github.com/openshift/ci-tools/blob/master/pkg/api/types.go

type ReleaseBuildConfiguration struct {
Metadata Metadata
InputConfiguration
// …
}

type InputConfiguration struct {
// …
Releases map[string]UnresolvedRelease
}

The first major input to E2E tests, seen at the beginning of the output log, are the release streams / payloads. They are configured in the `releases` entry in the configuration file.

Originally, they were specified in `tag_specification`, which provides a fixed subset of the same functionality. That field is now deprecated and will be removed, but can still be found in many configuration files.

```
type UnresolvedRelease struct {
    // Integration describes an integration stream
    // which we can create a payload out of
    Integration *Integration
    // Candidate describes a candidate release
    // payload
    Candidate *Candidate
    // Prerelease describes a yet-to-be released
    // payload
    Prerelease *Prerelease
    // Release describes a released payload
    Release *Release
}
```


Red Hat


```
type UnresolvedRelease struct {
    // Integration describes an integration stream
    // which we can create a payload out of
    Integration *Integration
    // Candidate describes a candidate release
    // payload
    Candidate *Candidate
    // Prerelease describes a yet-to-be released
    // payload
    Prerelease *Prerelease
    // Release describes a released payload
    Release *Release
}
```


The top level keys of the `releases` field are simply identifiers. Each value is a structure in the familiar format where only one of the pointers is ever non-null.

```
type Candidate struct {
    Product ReleaseProduct
    Architecture ReleaseArchitecture
    Stream ReleaseStream
    Version string
    Relative int
}

type Prerelease struct {
    Product ReleaseProduct
    Architecture ReleaseArchitecture
    VersionBounds VersionBounds
}
```


Red Hat


```
type Candidate struct {
    Product ReleaseProduct
    Architecture ReleaseArchitecture
    Stream ReleaseStream
    Version string
    Relative int
}

type Prerelease struct {
    Product ReleaseProduct
    Architecture ReleaseArchitecture
    VersionBounds VersionBounds
}
```


The `release`, `prerelease`, and `candidate` types all refer to existing payloads: they vary only in their source. `integration` streams (when not overridden, as described later) use `ImageStreams`.

```
pkg/release/
    candidate/
        client.go
        types.go
    client.go
    config/
        client.go
        config.go
    official/
        client.go
        types.go
    prerelease/
        client.go
```


Red Hat

Releases

pkg/release/
candidate/
client.go
types.go
client.go
config/
client.go
config.go
official/
client.go
types.go
prerelease/
client.go

The different sources used for these types can be seen in `pkg/release` in `openshift/ci-tools`.

- `candidate` / `prerelease`
  - `https://amd64.ocp.releases.ci.openshift.org`
  - release controller
- `release`
  - `https://api.openshift.com/api/upgrades_info/v1/graph`
  - cincinnati

Red Hat

Releases
candidate / prerelease
https://amd64.ocp.releases.ci.openshift.org
release controller
release
https://api.openshift.com/api/upgrades_info/v1/graph
cincinnati

Both `candidate` and `prerelease` types obtain their release payloads from the `release-controller`, according to the input values.
The `release` type obtains official releases from `cincinnati`.

```yaml
releases:
  initial:
    integration:
      name: "4.10"
      namespace: ocp
  latest:
    integration:
      include_built_images: \
        true
      name: "4.10"
      namespace: ocp
```

- `ReleaseImagesTagStep`
  - source → destination `ImageStream`
  - `$namespace/$name` → `ci-op-*/stable*`
- `AssembleReleaseStep`
  - `ImageStream` → release payload
  - `stable*` → `release:*`
  - will wait for built images if `include_built_images`

The two categories (payload vs. stream) determine the steps `ci-operator` will take to import and process the release in order to make it available to the test. `integration` streams, as mentioned previously, come from an `ImageStream`. This means two steps are required:

- `ReleaseImagesTagStep` will import (i.e. copy) the tags from the source.
- `AssembleReleaseStep` will create a release payload from the resulting `ImageStream`. If an entry declares `include_built_images`, this will cause this step to wait for all images to be built and tagged into the stream, so that they can be included in the payload. This is usually the case for `latest` payloads, so that they can be used to test the resulting release containing images built using the code under test.

```
tag_specification:
  namespace: ocp
  name: "4.10"
```

- always `initial` and `latest`
- `include_built_images` implicitly for `latest`
- `ReleaseImagesTagStep` (≈ `ReleaseSnapshotStep`)
- `RELEASE_IMAGE_*`

Red Hat

`tag_specification` is the precursor to `integration` (and `releases` in general). It is legacy now but can be found in some old jobs (and sometimes causes problems). It works roughly like a group of fixed values for integration streams.

Both `integration` releases and `tag_specification` can have their values overridden by `RELEASE_IMAGE_*` environment variables. When this happens (e.g. in jobs created by the release controller), the images are treated as input release payloads and processed as described below.

```
$ oc adm release extract \
    --file image-references \
    quay.io/openshift/okd:4.10.0-0.okd-2022-07-09-073606 \
    | yaml
kind: ImageStream
apiVersion: image.openshift.io/v1
metadata:
  name: 4.10.0-0.okd-2022-07-09-073606
  creationTimestamp: 2022-07-10T09:12:53Z
  annotations:
    release.openshift.io/from-image-stream: >
      origin/4.10-2022-07-09-073606
    release.openshift.io/from-release: >
      registry.ci.openshift.org/origin/release:4.10.0-0.…
…
```


Here is an example of the relevant contents of a release payload image. It contains the name, date of creation, source, …

```yaml
spec:
  lookupPolicy:
    local: false
  tags:
  - name: alibaba-cloud-controller-manager
    annotations:
      io.openshift.build.commit.id: 0
      io.openshift.build.commit.ref: release-4.10
      io.openshift.build.source-location: >
        https://github.com/openshift/…
    from:
      kind: DockerImage
      name: quay.io/openshift/okd-content@sha256:…
    generation: null
    importPolicy:
    referencePolicy:
      type: 0
…
```


Red Hat

…and the list of image references which will be used in the cluster installation and configuration.

```yaml
releases:
  latest:
    release:
      architecture: amd64
      channel: stable
      version: "4.10"
```

- `candidate`/`prerelease`
- `ImportReleaseStep`
  - release payload → `ImageStream`
  - `$src` → `release:*`
  - tags → `ImageStream`
  - `release:*` → `oc … extract` → `stable*`

The other types of releases use a completely different input mechanism. Since these are already published as release payloads, `ImportReleaseStep` is used instead. It:

- imports the payload directly to the `release` `ImageStream` (via OpenShift)
- extracts the images to `stable*`, so that individual images can be used in the same way as integration streams

Input images

Red Hat

Input images

"CI cycle"

```
0. import images / releases
1. build images
2. execute tests
3. promote images
4. goto 0
```


Red Hat

Input images / image mirroring
"CI cycle"
0. import images / releases
1. build images
2. execute tests
3. promote images
4. goto 0

In the next few sections, we are going to look at what is described as the "CI cycle" or "CI loop": the process by which a release stream goes from version x to version $x+1$.

We start with two preexisting sets of images (more on that later) which are imported into the test namespace:

- images from the release the particular component is part of
- auxiliary images, used in image builds and in the execution of tests

Images are then built and tests are executed to validate them, both by themselves and incorporated into the release stream (this is why you must have image builds / tests if there is a promotion rule).

Finally, if all checks are satisfied, the images are "promoted", i.e. written to the same release stream which was imported at the beginning, thus generating the $x+1$ release. Future executions of this and other pipelines will use the new set of images.

"CI cycle"

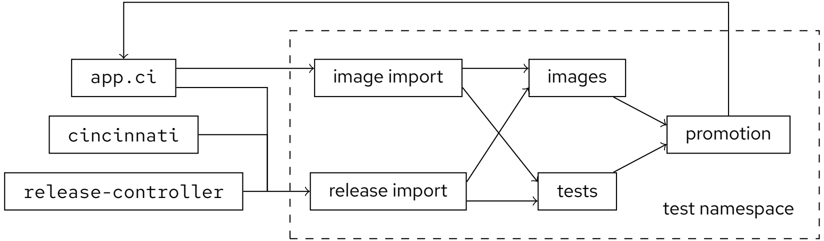
```
-1. ?
 0. import images / releases
 1. build images
 2. execute tests
 3. promote images
 4. goto 0
```


Red Hat

Input images / image mirroring

"CI cycle"
-1. ?
0. import images / releases
1. build images
2. execute tests
3. promote images
4. goto 0

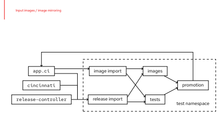
There remains, however, the question of how this process originates: if each pipeline execution is an inductive step, what is the basis?



Red Hat

2022-09-09

End-to-End Tests in OpenShift
└Input images
 └Image mirroring
  └Input images / image mirroring



This is the pictorial representation of this process. Images come from the left: base images come from the central registry in `app.ci` (more on that also later), release images come from any of the three places, depending on which type of `releases` field is used.
The sub-graph which originates in `app.ci` and returns to it finally after the promotion step is the CI cycle.

- supplemental images
  - `https://github.com/openshift/release/tree/master/clusters/app.ci/supplemental-ci-images`
  - `BuildConfigs`
- image mirroring
  - `https://github.com/openshift/release/tree/master/core-services/image-mirroring`
  - Quay/etc. ↔ `app.ci`
- ART / OCP builder images
  - `https://docs.ci.openshift.org/docs/architecture/images/`
  - `https://github.com/openshift/ocp-build-data.git`

Input images / image mirroring
supplemental images
https://github.com/openshift/release/tree/master/clusters/app.ci/supplemental-ci-images
BuildConfigs
image mirroring
https://github.com/openshift/release/tree/master/core-services/image-mirroring
Quay/etc. ↔ app.ci
ART / OCP builder images
https://docs.ci.openshift.org/docs/architecture/images/
https://github.com/openshift/ocp-build-data.git

Base images can come from several places:

- Images can be built directly using OpenShift `BuildConfigs`.
- A mirroring process exists between `app.ci` and Quay. It is actually bidirectional, but here we are only interested in images which are imported from Quay.
- *Productized* images, used to build official OpenShift release images, come from ART.

- supplemental images
  - `https://github.com/openshift/release/tree/master/clusters/app.ci/supplemental-ci-images`
  - `BuildConfigs`
  - `registry.ci.openshift.org/ci/ci-tools-build-root`
- image mirroring
  - `https://github.com/openshift/release/tree/master/core-services/image-mirroring`
  - Quay/etc. ↔ `app.ci`
  - `registry.ci.openshift.org/coreos/stream9:9`
- ART / OCP builder images
  - `https://docs.ci.openshift.org/docs/architecture/images/`
  - `https://github.com/openshift/ocp-build-data.git`
  - `registry.ci.openshift.org/ocp/builder:…`

Input images / image mirroring
- supplemental images
  - https://github.com/openshift/release/tree/master/clusters/app.ci/supplemental-ci-images
  - BuildConfigs
  - registry.ci.openshift.org/ci/ci-tools-build-root
- image mirroring
  - https://github.com/openshift/release/tree/master/core-services/image-mirroring
  - Quay/etc. ↔ app.ci
  - registry.ci.openshift.org/coreos/stream9:9
- ART / OCP builder images
  - https://docs.ci.openshift.org/docs/architecture/images/
  - https://github.com/openshift/ocp-build-data.git
  - registry.ci.openshift.org/ocp/builder:…

Note, however, that these images are all located in the `app.ci` central registry. Initially, they were simply referenced directly, but that very soon turned out to not scale to the number of jobs we had at the time (which was a small fraction of the current number).

`dptp-controller-manager`

- `cmd/dptp-controller-manager/`
- `pkg/controller/test-images-distributor/`

Red Hat

Input images / dptp-controller-manager

dptp-controller-manager
- cmd/dptp-controller-manager/
- pkg/controller/test-images-distributor/

For this reason, there is now a process which imports those images to each build cluster whenever required. It is one of the processes executed as part of the `dptp-controller-manager` (famed cluster node assassin) and is named `test-images-distributor`.

```
args:
…
- --enable-controller=test_images_distributor
- --enable-controller=promotionreconciler
- --enable-controller=serviceaccount_secret_refresher
- --enable-controller=testimagestreamimportcleaner
…
```


Red Hat

The command line shows the enabled controllers, which perform various functions in the CI clusters.

```
…
- --release-repo-git-sync-path=/var/repo/release
- --kubeconfig-dir=/var/kubeconfigs
- --registry-cluster-name=app.ci
- --testImagesDistributorOptions \
    .additional-image-stream-tag=ocp/builder:golang-1.10
…
- --testImagesDistributorOptions \
    .additional-image-stream-tag= \
    ocp/builder:rhel-7-golang-1.11
…
- --testImagesDistributorOptions \
    .additional-image-stream-namespace=ci
- --testImagesDistributorOptions \
    .additional-image-stream=rhcos/machine-os-content
…
```


It has a few options to explicitly include image streams and tags...

pkg/api/helper/imageextraction.go

- `TestInputImageStreamsFromResolvedConfig`
- `TestInputImageStreamTagsFromResolvedConfig`

Red Hat

Input images / dptp-controller-manager
pkg/api/helper/imageextraction.go
TestInputImageStreamsFromResolvedConfig
TestInputImageStreamTagsFromResolvedConfig

… but its main function is to inspect every `ci-operator` configuration file and extract input images to be synchronized, which it does automatically whenever the source streams are changed.

```
https://prow.ci.openshift.org/view/gs/origin-ci-test/
logs/branch-ci-openshift-ci-tools-master-images/
1561993456950185984

…
Building src
Build src succeeded after 4m48s
Building bin
Build bin succeeded after 25m54s
Building determinize-peribolos
Building ci-secret-generator
Building ci-operator-config-mirror
…
```


Red Hat

Input images / image promotion

https://prow.ci.openshift.org/view/gs/origin-ci-test/
logs/branch-ci-openshift-ci-tools-master-images/
1561993456950185984
…
Building src
Build src succeeded after 4m48s
Building bin
Build bin succeeded after 25m54s
Building determinize-peribolos
Building ci-secret-generator
Building ci-operator-config-mirror
…

Promotion is a relatively simple matter now that we have looked at the rest of the pipeline. We start by building all images not explicitly excluded, …

```
…
Build prcreator succeeded after 14m26s
Tagging prcreator into stable
Build private-prow-configs-mirror \
    succeeded after 15m51s
Tagging private-prow-configs-mirror into stable
Promoting tags to ci/${component}:latest: \
    applyconfig, auto-aggregator-job-names, \
    auto-config-brancher, auto-peribolos-sync, \
    auto-sippy-config-generator, …
Ran for 1h7m10s
```


Red Hat

2022-09-09

End-to-End Tests in OpenShift
└─Input images
  └─Image promotion
    └─Input images / image promotion

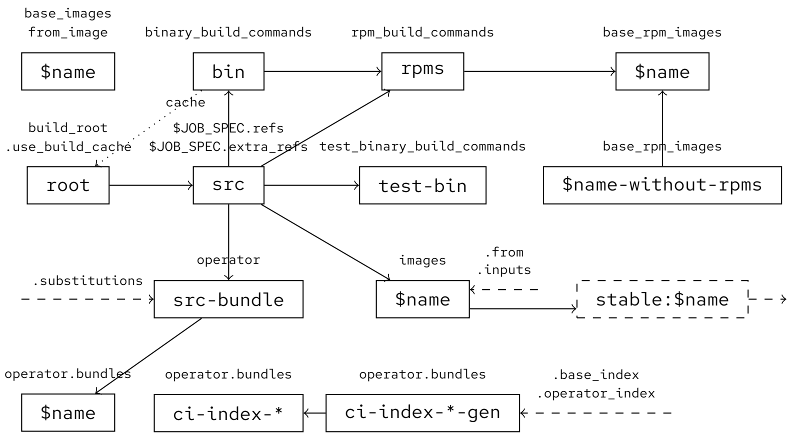
Input images / image promotion

```
…
Build prcreator succeeded after 14m26s
Tagging prcreator into stable
Build private-prow-configs-mirror \
    succeeded after 15m51s
Tagging private-prow-configs-mirror into stable
Promoting tags to ci/${component}:latest: \
    applyconfig, auto-aggregator-job-names, \
    auto-config-brancher, auto-peribolos-sync, \
    auto-sippy-config-generator, …
Ran for 1h7m10s
```


… then tag them in the `stable` `ImageStream` as usual, and finally transfer them to the central registry in `app.ci`, where they will be available to future jobs.

Image pipeline

Image pipeline

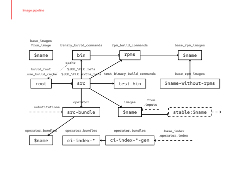


Red Hat

2022-09-09

End-to-End Tests in OpenShift
└Image pipeline

└Image pipeline



Legend:

- Solid boxes are pipeline images (tags, technically), solid lines are dependencies.
- The dashed `stable` box represents the "internal" promotion to the stable stream prior to the execution of tests.
- Dashed lines represent edges not fully depicted since they are optional and can be added to any image in the pipeline:
  - Each entry in `operator.substitutions` makes `src-bundle` depend on that image.
  - The `operator.substitutions` entry, if specified, makes the `src-bundle` depend on those images.
  - The `operator.base_index` entry, if specified, makes all index generator images depend on that image.

Thank you

Red Hat

Thank you