Red Hat Quay 3.4

Deploy Red Hat Quay on OpenShift with the Quay Operator

Deploy Red Hat Quay on OpenShift with Quay Operator

Last Updated: 2021-04-20

Red Hat Quay 3.4 Deploy Red Hat Quay on OpenShift with the Quay Operator

Deploy Red Hat Quay on OpenShift with Quay Operator

Legal Notice

Abstract

Deploy Red Hat Quay on an OpenShift Cluster with the Red Hat Quay Operator

Table of Contents

PREFACE

Red Hat Quay is an enterprise-quality container registry. Use Red Hat Quay to build and store container images, then make them available to deploy across your enterprise.

The Red Hat Quay Operator provides a simple method to deploy and manage a Red Hat Quay cluster. This is the preferred procedure for deploying Red Hat Quay on OpenShift and is covered in this guide.

Note that this version of the Red Hat Quay Operator has been completely rewritten and differs substantially from earlier versions. Please review this documentation carefully.

CHAPTER 1. PREREQUISITES FOR RED HAT QUAY ON OPENSHIFT

Here are a few things you need to know before you begin the Red Hat Quay Operator on OpenShift deployment:

- **OpenShift cluster**: You need a privileged account to an OpenShift 4.5 or later cluster on which to deploy the Red Hat Quay Operator. That account must have the ability to create namespaces at the cluster scope.
- **Resource Requirements**: Each Red Hat Quay application pod has the following resource requirements:
  - 8Gi of memory
  - 2000 millicores of CPU.

The Red Hat Quay Operator will create at least one application pod per Red Hat Quay deployment it manages. Ensure your OpenShift cluster has sufficient compute resources for these requirements.

- **Object Storage**: By default, the Red Hat Quay Operator uses the **ObjectBucketClaim** Kubernetes API to provision object storage. Consuming this API decouples the Operator from any vendor-specific implementation. OpenShift Container Storage provides this API via its NooBaa component, which will be used in this example. Otherwise, Red Hat Quay can be manually configured to use any of the following supported cloud storage options:
  - Amazon S3 (see S3 IAM Bucket Policy for details on configuring an S3 bucket policy for Red Hat Quay)
  - Azure Blob Storage
  - Google Cloud Storage
  - Ceph Object Gateway (RADOS)
  - OpenStack Swift
  - CloudFront + S3

CHAPTER 2. INSTALLING THE QUAY OPERATOR

2.1. DIFFERENCES FROM EARLIER VERSIONS

As of Red Hat Quay 3.4.0, the Operator has been completely re-written to provide an improved out of the box experience as well as support for more Day 2 operations. As a result the new Operator is simpler to use and is more opinionated. The key differences from earlier versions of the Operator are:

- The **QuayEcosystem** custom resource has been replaced with the **QuayRegistry** custom resource
- The default installation options produces a fully supported Quay environment with all managed dependencies (database, object storage, etc) ready for production use
- A new robust validation library for Quay's configuration which is shared by the Quay application and config tool for consistency
- Registry object storage can now be managed by the Operator using the **ObjectBucketClaim** Kubernetes API (the NooBaa component of Red Hat OpenShift Container Storage (RHOCS) is one implementation of this API)
- Customization of the container images used by deployed pods for testing and development scenarios

2.2. BEFORE INSTALLING THE QUAY OPERATOR

2.2.1. Deciding On a Storage Solution

If you want the Operator to manage its own object storage, you will first need to ensure the RHOCS is available on your OpenShift cluster to provide the **ObjectBucketClaim** API. If you already have object storage ready to be used by the Operator, skip to Installing the Operator.

2.2.2. Enabling OpenShift Container Storage

To install the RHOCS Operator and configure a lightweight NooBaa (S3-compatible) object storage:

1. Open the OpenShift console and select Operators → OperatorHub, then select the OpenShift Container Storage Operator.
2. Select Install. Accept all default options and select Install again.
3. After a minute or so, the Operator will install and create a namespace **openshift-storage**. You can confirm it is completed when the **Status** column is marked **Succeeded**.
4. Create NooBaa object storage. Save the following YAML to a file called **noobaa.yml**.

```
apiVersion: noobaa.io/v1alpha1
kind: NooBaa
metadata:
  name: noobaa
  namespace: openshift-storage
spec:
 dbResources:
   requests:
```

```
      cpu: '0.1'
      memory: 1Gi
  coreResources:
    requests:
      cpu: '0.1'
      memory: 1Gi
```

Then run the following:

```
$ oc create -n openshift-storage -f noobaa.yml
noobaa.noobaa.io/noobaa created
```

5. After a minute or so, you should see the object storage ready for use (**PHASE** column is marked **Ready**)

```
$ oc get -n openshift-storage noobaas noobaa -w
NAME     MGMT-ENDPOINTS               S3-ENDPOINTS               IMAGE
PHASE   AGE
noobaa   [https://10.0.32.3:30318]   [https://10.0.32.3:31958]   registry.redhat.io/ocs4/mcg-
core-
rhel8@sha256:56624aa7dd4ca178c1887343c7445a9425a841600b1309f6deace37ce6b8678d
Ready   3d18h
```

2.3. INSTALLING THE OPERATOR FROM OPERATORHUB

1. Using the OpenShift console, Select Operators → OperatorHub, then select the Quay Operator. If there is more than one, be sure to use the Red Hat certified Operator and not the community version.

2. Select Install. The Operator Subscription page appears.

3. Choose the following then select Subscribe:

   - Installation Mode: Choose either 'All namespaces' or 'A specific namespace' depending on whether you want the Operator to be available cluster-wide or only within a single namespace (all-namespaces recommended)

   - Update Channel: Choose the update channel (only one may be available)

   - Approval Strategy: Choose to approve automatic or manual updates

4. Select Install.

5. After a minute you will see the Operator installed successfully in the Installed Operators page.

CHAPTER 3. HIGH LEVEL CONCEPTS

3.1. QUAYREGISTRY API

The Quay Operator provides the **QuayRegistry** custom resource API to declaratively manage Quay container registries on the cluster. Use either the OpenShift UI or a command-line tool to interact with this API.

- Creating a **QuayRegistry** will result in the Operator deploying and configuring all necessary resources needed to run Quay on the cluster.
- Editing a **QuayRegistry** will result in the Operator reconciling the changes and creating/updating/deleting objects to match the desired configuration.
- Deleting a **QuayRegistry** will result in garbage collection of all previously created resources and the Quay container registry will no longer be available.

The **QuayRegistry** API is fairly simple, and the fields are outlined in the following sections.

3.1.1. Components

Quay is a powerful container registry platform and as a result, requires a decent number of dependencies. These include a database, object storage, Redis, and others. The Quay Operator manages an opinionated deployment of Quay and its dependencies on Kubernetes. These dependencies are treated as *components* and are configured through the **QuayRegistry** API.

In the **QuayRegistry** custom resource, the **spec.components** field configures components. Each component contains two fields: **kind** - the name of the component, and **managed** - boolean whether the component lifecycle is handled by the Operator. By default (omitting this field), all components are managed and will be autofilled upon reconciliation for visibility:

```
spec:
  components:
    - kind: postgres
      managed: true
    ...
```

Unless your **QuayRegistry** custom resource specifies otherwise, the Operator will use defaults for the following managed components:

- **postgres** Stores the registry metadata. Uses a version of Postgres 10 from the Software Collections.
- **redis** Handles Quay builder coordination and some internal logging.
- **objectstorage** Stores image layer blobs. Utilizes the **ObjectBucketClaim** Kubernetes API which is provided by Noobaa/RHOCS.
- **clair** Provides image vulnerability scanning.
- **horizontalpodautoscaler** Adjusts the number of Quay pods depending on memory/cpu consumption.
- **mirror** Configures a repository mirror worker (to support optional repository mirroring).

- **route** Provides an external entrypoint to the Quay registry from outside of OpenShift.

3.1.1.1. Considerations For Managed Components

While the Operator will handle any required configuration and installation work needed for Red Hat Quay to use the managed components, there are several considerations to keep in mind.

- Database backups should be performed regularly using either the supplied tools on the Postgres image or your own backup infrastructure. The Operator does not currently ensure the Postgres database is backed up.
- Restoring the Postgres database from a backup must be done using Postgres tools and procedures. Be aware that your Quay **Pods** should not be running while the database restore is in progress.
- Database disk space is allocated automatically by the Operator with 50 GiB. This number represents a usable amount of storage for most small to medium Red Hat Quay installations but may not be sufficient for your use cases. Resizing the database volume is currently not handled by the Operator.
- Object storage disk space is allocated automatically by the Operator with 50 GiB. This number represents a usable amount of storage for most small to medium Red Hat Quay installations but may not be sufficient for your use cases. Resizing the RHOCS volume is currently not handled by the Operator. See the section below on resizing managed storage for more details.
- The Operator will deploy an OpenShift **Route** as the default entrypoint to the registry. If you prefer a different entrypoint (e.g. **Ingress** or direct **Service** access that configuration will need to be done manually).

If any of these considerations are unacceptable for your environment, it would be suggested to provide the Operator with unmanaged resources or overrides as described in the following sections.

3.1.1.2. Using Existing (Un-Managed) Components With the Quay Operator

If you have existing components such as Postgres, Redis or object storage that you would like to use with Quay, you first configure them within the Quay configuration bundle (**config.yaml**) and then reference the bundle in your **QuayRegistry** (as a Kubernetes **Secret**) while indicating which components are unmanaged.

For example, to use an existing Postgres database:

1. Create a **Secret** with the necessary database fields in a **config.yaml** file:

   config.yaml:

   ```
   DB_URI: postgresql://test-quay-database:postgres@test-quay-database:5432/test-quay-database
   ```

   ```
   $ kubectl create secret generic --from-file config.yaml=./config.yaml test-config-bundle
   ```

2. Create a QuayRegistry which marks postgres component as unmanaged and references the created Secret:

   quayregistry.yaml

```
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: test
spec:
  configBundleSecret: test-config-bundle
  components:
    - kind: postgres
      managed: false
```

The deployed Quay application will now use the external database.

NOTE

The Quay config editor can also be used to create or modify an existing config bundle and simplify the process of updating the Kubernetes **Secret**, especially for multiple changes. When Quay's configuration is changed via the config editor and sent to the Operator, the Quay deployment will be updated to reflect the new configuration.

3.1.2. Config Bundle Secret

The **spec.configBundleSecret** field is a reference to the **metadata.name** of a **Secret** in the same namespace as the **QuayRegistry**. This **Secret** must contain a **config.yaml** key/value pair. This **config.yaml** file is a Quay config YAML file. This field is optional, and will be auto-filled by the Operator if not provided. If provided, it serves as the base set of config fields which are later merged with other fields from any managed components to form a final output **Secret**, which is then mounted into the Quay application pods.

3.1.3. AWS S3 CloudFront

If you use AWS S3 CloudFront for backend registry storage, specify the private key as shown in the following example:

```
$ oc create secret generic --from-file config.yaml=./config_awss3cloudfront.yaml --from-file default-cloudfront-signing-key.pem=./default-cloudfront-signing-key.pem test-config-bundle
```

3.2. QUAYREGISTRY STATUS

Lifecycle observability for a given Quay deployment is reported in the **status** section of the corresponding **QuayRegistry** object. The Operator constantly updates this section, and this should be the first place to look for any problems or state changes in Quay or its managed dependencies.

3.2.1. Registry Endpoint

Once Quay is ready to be used, the **status.registryEndpoint** field will be populated with the publicly available hostname of the registry.

3.2.2. Config Editor Endpoint

Access Quay's UI-based config editor using **status.configEditorEndpoint**.

3.2.3. Config Editor Credentials Secret

The username/password for the config editor UI will be stored in a **Secret** in the same namespace as the **QuayRegistry** referenced by **status.configEditorCredentialsSecret**.

3.2.4. Current Version

The current version of Quay that is running will be reported in **status.currentVersion**.

3.2.5. Conditions

Certain conditions will be reported in **status.conditions**.

CHAPTER 4. DEPLOYING QUAY USING THE QUAY OPERATOR

4.1. CREATING A QUAY REGISTRY

The default configuration tells the Operator to manage all of Quay's dependencies (database, Redis, object storage, etc).

4.1.1. OpenShift Console

1. Select Operators → Installed Operators, then select the Quay Operator to navigate to the Operator detail view.
2. Click 'Create Instance' on the 'Quay Registry' tile under 'Provided APIs'.
3. Optionally change the 'Name' of the **QuayRegistry**. This will affect the hostname of the registry. All other fields have been populated with defaults.
4. Click 'Create' to submit the **QuayRegistry** to be deployed by the Quay Operator.
5. You should be redirected to the **QuayRegistry** list view. Click on the **QuayRegistry** you just created to see the detail view.
6. Once the 'Registry Endpoint' has a value, click it to access your new Quay registry via the UI. You can now select 'Create Account' to create a user and sign in.

4.1.2. Command Line

The same result can be achieved using the CLI.

1. Create the following **QuayRegistry** custom resource in a file called **quay.yaml**.

   quay.yaml:

   ```
   apiVersion: quay.redhat.com/v1
   kind: QuayRegistry
   metadata:
     name: my-registry
   ```

2. Create the **QuayRegistry** in your namespace:

   ```
   $ oc create -n <your-namespace> -f quay.yaml
   ```

3. Wait until the **status.registryEndpoint** is populated.

   ```
   $ oc get -n <your-namespace> quayregistry my-registry -o jsonpath="
   {.status.registryEndpoint}" -w
   ```

4. Once the **status.registryEndpoint** has a value, navigate to it using your web browser to access your new Quay registry via the UI. You can now select 'Create Account' to create a user and sign in.

4.2. DEPLOYING QUAY ON INFRASTRUCTURE NODES

By default, Quay-related pods are placed on arbitrary worker nodes when using the Operator to deploy the registry. The OpenShift Container Platform documentation shows how to use machine sets to configure nodes to only host infrastructure components (see https://docs.openshift.com/container-platform/4.7/machine_management/creating-infrastructure-machinesets.html).

If you are not using OCP MachineSet resources to deploy infra nodes, this section shows you how to manually label and taint nodes for infrastructure purposes.

Once you have your configured your infrastructure nodes, either manually or using machine sets, you can then control the placement of Quay pods on these nodes using node selectors and tolerations.

4.2.1. Label and taint nodes for infrastructure use

In the cluster used in this example, there are three master nodes and six worker nodes:

```
$ oc get nodes
NAME                                          STATUS   ROLES    AGE     VERSION
user1-jcnp6-master-0.c.quay-devel.internal        Ready    master   3h30m   v1.20.0+ba45583
user1-jcnp6-master-1.c.quay-devel.internal        Ready    master   3h30m   v1.20.0+ba45583
user1-jcnp6-master-2.c.quay-devel.internal        Ready    master   3h30m   v1.20.0+ba45583
user1-jcnp6-worker-b-65plj.c.quay-devel.internal   Ready    worker   3h21m   v1.20.0+ba45583
user1-jcnp6-worker-b-jr7hc.c.quay-devel.internal   Ready    worker   3h21m   v1.20.0+ba45583
user1-jcnp6-worker-c-jrq4v.c.quay-devel.internal   Ready    worker   3h21m   v1.20.0+ba45583
user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal   Ready    worker   3h21m   v1.20.0+ba45583
user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal   Ready    worker   3h22m   v1.20.0+ba45583
user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal   Ready    worker   3h21m   v1.20.0+ba45583
```

Label the final three worker nodes for infrastructure use:

```
$ oc label node --overwrite user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal node-
role.kubernetes.io/infra=
$ oc label node --overwrite user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal node-
role.kubernetes.io/infra=
$ oc label node --overwrite user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal node-
role.kubernetes.io/infra=
```

Now, when you list the nodes in the cluster, the last 3 worker nodes will have an added role of **infra**:

```
$ oc get nodes
NAME                                          STATUS   ROLES          AGE     VERSION
user1-jcnp6-master-0.c.quay-devel.internal        Ready    master         4h14m   v1.20.0+ba45583
user1-jcnp6-master-1.c.quay-devel.internal        Ready    master         4h15m   v1.20.0+ba45583
user1-jcnp6-master-2.c.quay-devel.internal        Ready    master         4h14m   v1.20.0+ba45583
user1-jcnp6-worker-b-65plj.c.quay-devel.internal   Ready    worker         4h6m    v1.20.0+ba45583
user1-jcnp6-worker-b-jr7hc.c.quay-devel.internal   Ready    worker         4h5m    v1.20.0+ba45583
user1-jcnp6-worker-c-jrq4v.c.quay-devel.internal   Ready    worker         4h5m    v1.20.0+ba45583
user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal   Ready    infra,worker   4h6m    v1.20.0+ba45583
user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal   Ready    infra,worker   4h6m    v1.20.0+ba45583
user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal   Ready    infra,worker   4h6m    v1.20.0+ba45583
```

With an infra node being assigned as a worker, there is a chance that user workloads could get inadvertently assigned to an infra node. To avoid this, you can apply a taint to the infra node and then add tolerations for the pods you want to control.

```
$ oc adm taint nodes user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal node-role.kubernetes.io/infra:NoSchedule
$ oc adm taint nodes user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal node-role.kubernetes.io/infra:NoSchedule
$ oc adm taint nodes user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal node-role.kubernetes.io/infra:NoSchedule
```

4.2.2. Create a Project with node selector and toleration

If you have already deployed Quay using the Quay Operator, remove the installed operator and any specific namespace(s) you created for the deployment.

Create a Project resource, specifying a node selector and toleration as shown in the following example:

quay-registry.yaml

```
kind: Project
apiVersion: project.openshift.io/v1
metadata:
  name: quay-registry
  annotations:
    openshift.io/node-selector: 'node-role.kubernetes.io/infra='
    scheduler.alpha.kubernetes.io/defaultTolerations: >-
      [{"operator": "Exists", "effect": "NoSchedule", "key":
      "node-role.kubernetes.io/infra"}
      ]
```

Use the **oc apply** command to create the project:

```
$ oc apply -f quay-registry.yaml
project.project.openshift.io/quay-registry created
```

Any subsequent resources created in the **quay-registry** namespace should now be scheduled on the dedicated infrastructure nodes.

4.2.3. Install the Quay Operator in the namespace

When installing the Quay Operator, specify the appropriate project namespace explicitly, in this case **quay-registry**. This will result in the operator pod itself landing on one of the three infrastructure nodes:

```
$ oc get pods -n quay-registry -o wide
NAME                                    READY   STATUS    RESTARTS   AGE   IP            NODE

quay-operator.v3.4.1-6f6597d8d8-bd4dp   1/1     Running   0          30s   10.131.0.16   user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal
```

4.2.4. Create the registry

Create the registry as explained earlier, and then wait for the deployment to be ready. When you list the Quay pods, you should now see that they have only been scheduled on the three nodes that you have labelled for infrastructure purposes:

```
$ oc get pods -n quay-registry -o wide
NAME                                              READY   STATUS      RESTARTS   AGE     IP            NODE

example-registry-clair-app-789d6d984d-gpbwd               1/1     Running     1          5m57s   10.130.2.80
user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal
example-registry-clair-postgres-7c8697f5-zkzht            1/1     Running     0          4m53s   10.129.2.19
user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal
example-registry-quay-app-56dd755b6d-glbf7                1/1     Running     1          5m57s   10.129.2.17
user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal
example-registry-quay-config-editor-7bf9bccc7b-dpc6d      1/1     Running     0          5m57s
10.131.0.23   user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal
example-registry-quay-database-8dc7cfd69-dr2cc            1/1     Running     0          5m43s   10.129.2.18
  user1-jcnp6-worker-c-pwxfp.c.quay-devel.internal
example-registry-quay-mirror-78df886bcc-v75p9             1/1     Running     0          5m16s   10.131.0.24
user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal
example-registry-quay-postgres-init-8s8g9                 0/1     Completed   0          5m54s   10.130.2.79
user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal
example-registry-quay-redis-5688ddcdb6-ndp4t              1/1     Running     0          5m56s   10.130.2.78
user1-jcnp6-worker-d-m9gg4.c.quay-devel.internal
quay-operator.v3.4.1-6f6597d8d8-bd4dp                     1/1     Running     0          22m     10.131.0.16
user1-jcnp6-worker-d-h5tv2.c.quay-devel.internal
```

CHAPTER 5. UPGRADING QUAY USING THE QUAY OPERATOR

The Quay Operator follows a *synchronized versioning* scheme, which means that each version of the Operator is tied to the version of Quay and its components which it manages. There is no field on the **QuayRegistry** custom resource which sets the version of Quay to deploy; the Operator only knows how to deploy a single version of all components. This scheme was chosen to ensure that all components work well together and to reduce the complexity of the Operator needing to know how to manage the lifecycles of many different versions of Quay on Kubernetes.

5.1. OPERATOR LIFECYCLE MANAGER

The Quay Operator should be installed and upgraded using the Operator Lifecycle Manager (OLM). When creating a **Subscription** with the default **approvalStrategy: Automatic**, OLM will automatically upgrade the Quay Operator whenever a new version becomes available.

WARNING

When the Quay Operator is installed via Operator Lifecycle Manager it may be configured to support automatic or manual upgrades. This option is shown on the Operator Hub page for the Quay Operator during installation. It can also be found in the Quay Operator **Subscription** object via the **approvalStrategy** field. Choosing **Automatic** means that your Quay Operator will automatically be upgraded whenever a new Operator version is released. If this is not desireable, then the **Manual** approval strategy should be selected.

5.2. UPGRADING QUAY BY UPGRADING THE QUAY OPERATOR

The general approach for upgrading installed Operators on OpenShift is documented at Upgrading installed Operators.

5.2.1. Upgrading Quay

From a Red Hat Quay point of view, to update from one minor version to the next, for example, 3.4 → 3.5, you need to actively change the update channel for the Quay Operator.

For **z** stream upgrades, for example, 3.4.2 → 3.4.3, updates are released in the major-minor channel that the user initially selected during install. The procedure to perform a **z** stream upgrade depends on the **approvalStrategy** as outlined above. If the approval strategy is set to **Automatic**, the Operator will upgrade automatically to the newest **z** stream, resulting in automatic, rolling Quay updates to newer **z** streams with little to no downtime. Otherwise, the update must be manually approved before installation can begin.

5.2.2. Changing the update channel for an Operator

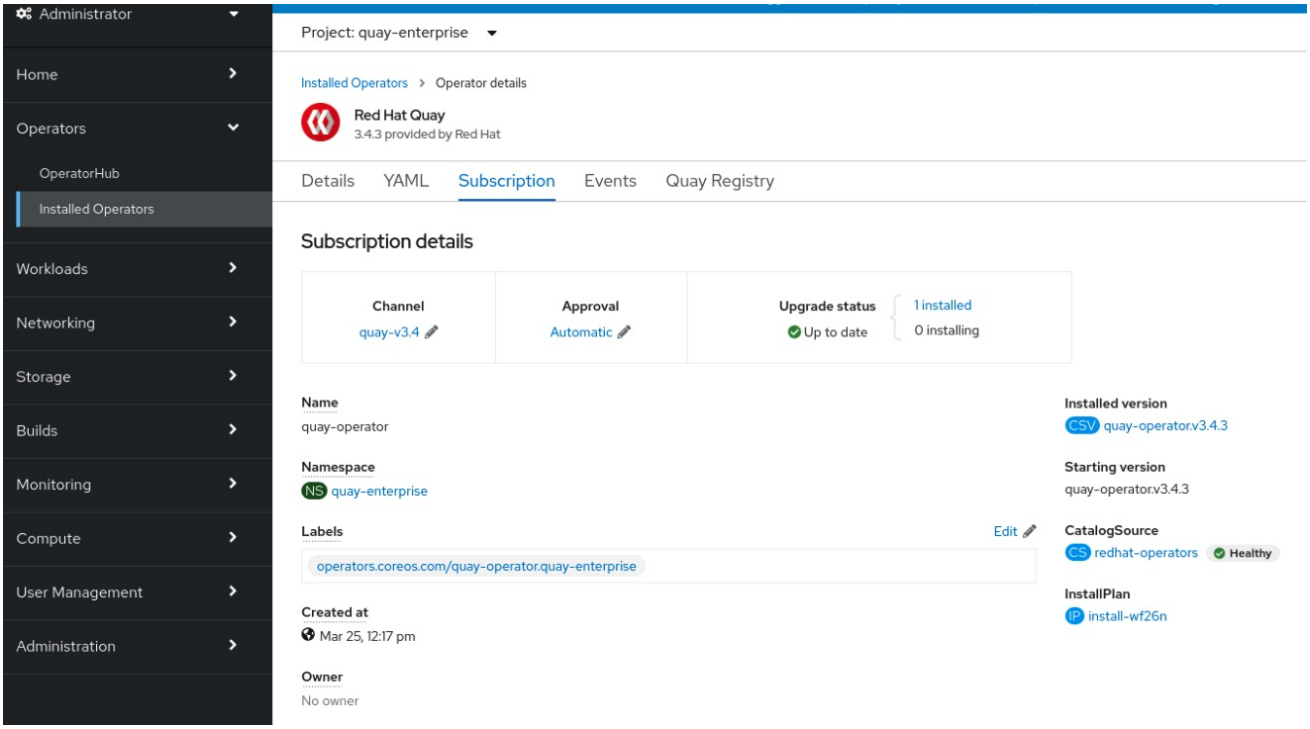
The subscription of an installed Operator specifies an update channel, which is used to track and receive updates for the Operator. To upgrade the Quay Operator to start tracking and receiving updates from a newer channel, change the update channel in the **Subscription** tab for the installed Quay Operator. For

subscriptions with an **Automatic** approval strategy, the upgrade begins automatically and can be monitored on the page that lists the Installed Operators.

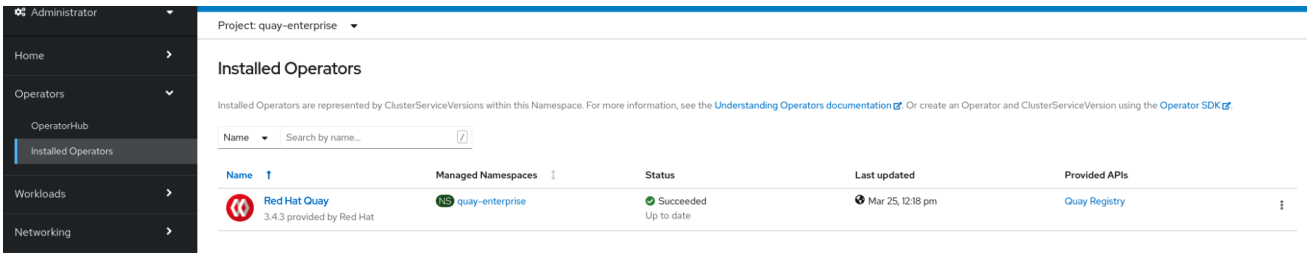
5.2.3. Manually approving a pending Operator upgrade

If an installed Operator has the approval strategy in its subscription set to **Manual**, when new updates are released in its current update channel, the update must be manually approved before installation can begin. If the Quay Operator has a pending upgrade, this status will be displayed in the list of Installed Operators. In the **Subscription** tab for the Quay Operator, you can preview the install plan and review the resources that are listed as available for upgrade. If satisfied, click **Approve** and return to the page that lists Installed Operators to monitor the progress of the upgrade.

The following image shows the **Subscription** tab in the UI, including the update **Channel**, the **Approval** strategy, the **Upgrade status** and the **InstallPlan**:



The list of Installed Operators provides a high-level summary of the current Quay installation:



5.3. UPGRADING A QUAYREGISTRY

When the Quay Operator starts up, it immediately looks for any **QuayRegistries** it can find in the namespace(s) it is configured to watch. When it finds one, the following logic is used:

- If **status.currentVersion** is unset, reconcile as normal.
- If **status.currentVersion** equals the Operator version, reconcile as normal.

- If **status.currentVersion** does not equal the Operator version, check if it can be upgraded. If it can, perform upgrade tasks and set the **status.currentVersion** to the Operator's version once complete. If it cannot be upgraded, return an error and leave the **QuayRegistry** and its deployed Kubernetes objects alone.

5.4. UPGRADING A QUAYECOSYSTEM

Upgrades are supported from previous versions of the Operator which used the **QuayEcosystem** API for a limited set of configurations. To ensure that migrations do not happen unexpectedly, a special label needs to be applied to the **QuayEcosystem** for it to be migrated. A new **QuayRegistry** will be created for the Operator to manage, but the old **QuayEcosystem** will remain until manually deleted to ensure that you can roll back and still access Quay in case anything goes wrong. To migrate an existing **QuayEcosystem** to a new **QuayRegistry**, follow these steps:

1. Add **"quay-operator/migrate": "true"** to the **metadata.labels** of the **QuayEcosystem**.

```
$ oc edit quayecosystem <quayecosystemname>
```

```
metadata:
  labels:
    quay-operator/migrate: "true"
```

2. Wait for a **QuayRegistry** to be created with the same **metadata.name** as your **QuayEcosystem**. The **QuayEcosystem** will be marked with the label **"quay-operator/migration-complete": "true"**.

3. Once the **status.registryEndpoint** of the new **QuayRegistry** is set, access Quay and confirm all data and settings were migrated successfully.

4. When you are confident everything worked correctly, you may delete the **QuayEcosystem** and Kubernetes garbage collection will clean up all old resources.

5.4.1. Reverting QuayEcosystem Upgrade

If something goes wrong during the automatic upgrade from **QuayEcosystem** to **QuayRegistry**, follow these steps to revert back to using the **QuayEcosystem**:

- Delete the **QuayRegistry** using either the UI or **kubectl**:

```
$ kubectl delete -n <namespace> quayregistry <quayecosystem-name>
```

- If external access was provided using a **Route**, change the **Route** to point back to the original **Service** using the UI or **kubectl**.

NOTE

If your **QuayEcosystem** was managing the Postgres database, the upgrade process will migrate your data to a new Postgres database managed by the upgraded Operator. Your old database will not be changed or removed but Quay will no longer use it once the migration is complete. If there are issues during the data migration, the upgrade process will exit and it is recommended that you continue with your database as an unmanaged component.

5.4.2. Supported QuayEcosystem Configurations for Upgrades

The Quay Operator will report errors in its logs and in **status.conditions** if migrating a **QuayEcosystem** component fails or is unsupported. All unmanaged components should migrate successfully because no Kubernetes resources need to be adopted and all the necessary values are already provided in Quay's **config.yaml**.

Database

Ephemeral database not supported (**volumeSize** field must be set).

Redis

Nothing special needed.

External Access

Only passthrough **Route** access supported for automatic migration. Manual migration required for other methods.

- **LoadBalancer** without custom hostname: After the **QuayEcosystem** is marked with label **"quay-operator/migration-complete": "true"**, delete the **metadata.ownerReferences** field from existing **Service** *before* deleting the **QuayEcosystem** to prevent Kubernetes from garbage collecting the **Service** and removing the load balancer. A new **Service** will be created with **metadata.name** format **<QuayEcosystem-name>-quay-app**. Edit the **spec.selector** of the existing **Service** to match the **spec.selector** of the new **Service** so traffic to the old load balancer endpoint will now be directed to the new pods. You are now responsible for the old **Service**; the Quay Operator will not manage it.

- **LoadBalancer**/**NodePort**/**Ingress** with custom hostname: A new **Service** of type **LoadBalancer** will be created with **metadata.name** format **<QuayEcosystem-name>-quay-app**. Change your DNS settings to point to the **status.loadBalancer** endpoint provided by the new **Service**.

Clair

Nothing special needed.

Object Storage

QuayEcosystem did not have a managed object storage component, so object storage will always be marked as unmanaged. Local storage is not supported.

Repository Mirroring

Nothing special needed.

CHAPTER 6. ADVANCED CONCEPTS

6.1. CUSTOMIZING THE QUAY DEPLOYMENT

The Quay Operator takes an opinionated strategy towards deploying Quay and its dependencies, however there are places where the Quay deployment can be customized.

6.1.1. Quay Application Configuration

Once deployed, the Quay application itself can be configured as normal using the config editor UI or by modifying the **Secret** containing the Quay configuration bundle. The Operator uses the **Secret** named in the **spec.configBundleSecret** field but does not watch this resource for changes. It is recommended that configuration changes be made to a new **Secret** resource and the **spec.configBundleSecret** field be updated to reflect the change. In the event there are issues with the new configuration, it is simple to revert the value of **spec.configBundleSecret** to the older **Secret**.

6.1.2. Customizing External Access to the Registry

When running on OpenShift, the **Routes** API is available and will automatically be used as a managed component. After creating the QuayRegistry, the external access point can be found in the status block of the **QuayRegistry**:

```
status:
  registryEndpoint: some-quay.my-namespace.apps.mycluster.com
```

The Operator creates a Service of **type: Loadbalancer** for your registry. You can configure your DNS provider to point the **SERVER_HOSTNAME** to the external IP address of the service.

```
$ oc get services -n <namespace>
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP        PORT(S)             AGE
some-quay            ClusterIP   172.30.143.199   34.123.133.39      443/TCP,9091/TCP    23h
```

6.1.2.1. Using a Custom Hostname and TLS

By default, a **Route** will be created with the default generated hostname and a certificate/key pair will be generated for TLS. If you want to access Red Hat Quay using a custom hostname and bring your own TLS certificate/key pair, follow these steps.

Because Quay will use TLS for in-cluster communication with other services within Kubernetes (like Clair), you must ensure that the certificate/key pair you use has Subject Alternative Names (SANs) for each of the following hostname patterns:

- **<quayregistry-name>-quay-app**
- **<quayregistry-name>-quay-app.<quayregistry-namespace>.svc**
- **<quayregistry-name>-quay-app.<quayregistry-namespace>.svc.cluster.local**

If **FEATURE_BUILD_SUPPORT: true**, then make sure the certificate/key pair also includes **BUILDMAN_HOSTNAME**.

If all of the above hostnames are not included as SANs, then the Quay Operator will reject your provided certificate/key pair and generate one to be used by Red Hat Quay.

Next, create a **Secret** with the following content:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-config-bundle
data:
  config.yaml: <must include SERVER_HOSTNAME field with your custom hostname>
  ssl.cert: <your TLS certificate>
  ssl.key: <your TLS key>
```

Then, create a QuayRegistry which references the created **Secret**:

```
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: some-quay
spec:
  configBundleSecret: my-config-bundle
```

6.1.3. Disabling Route Component

To prevent the Operator from creating a **Route**, mark the component as unmanaged in the **QuayRegistry**:

```
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: some-quay
spec:
  components:
    - kind: route
      managed: false
```

NOTE

Disabling the default **Route** means you are now responsible for creating a **Route**, **Service**, or **Ingress** in order to access the Quay instance and that whatever DNS you use must match the **SERVER_HOSTNAME** in the Quay config.

6.1.4. Resizing Managed Storage

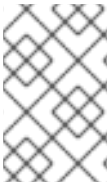
The Quay Operator creates default object storage using the defaults provided by RHOCS when creating a **NooBaa** object (50 Gib). There are two ways to extend this storage; you can resize an existing PVC or add more PVCs to a new storage pool.

6.1.4.1. Resize Noobaa PVC

1. Log into the OpenShift console and select **Storage → Persistent Volume Claims**.
2. Select the **PersistentVolumeClaim** named like **noobaa-default-backing-store-noobaa-pvc-***.
3. From the Action menu, select **Expand PVC**.

4. Enter the new size of the Persistent Volume Claim and select **Expand**.

After a few minutes (depending on the size of the PVC), the expanded size should reflect in the PVC's **Capacity** field.



NOTE

Expanding CSI volumes is a Technology Preview feature only. For more information, see https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html/storage/expanding-persistent-volumes.

6.1.4.2. Add Another Storage Pool

1. Log into the OpenShift console and select **Networking → Routes**. Make sure the **openshift-storage** project is selected.
2. Click on the **Location** field for the **noobaa-mgmt** Route.
3. Log into the Noobaa Management Console.
4. On the main dashboard, under **Storage Resources**, select **Add Storage Resources**.
5. Select **Deploy Kubernetes Pool**
6. Enter a new pool name. Click **Next**.
7. Choose the number of Pods to manage the pool and set the size per node. Click **Next**.
8. Click **Deploy**.

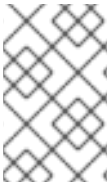
After a few minutes, the additional storage pool will be added to the Noobaa resources and available for use by Red Hat Quay.

6.1.5. Disabling the Horizontal Pod Autoscaler

If you wish to disable autoscaling or create your own **HorizontalPodAutoscaler**, simply specify the component as unmanaged in the **QuayRegistry** instance:

```
apiVersion: quay.redhat.com/v1
kind: QuayRegistry
metadata:
  name: some-quay
spec:
  components:
    - kind: horizontalpodautoscaler
      managed: false
```

6.1.6. Customizing Default Operator Images



NOTE

Using this mechanism is not supported for production Quay environments and is strongly encouraged only for development/testing purposes. There is no guarantee your deployment will work correctly when using non-default images with the Quay Operator.

In certain circumstances, it may be useful to override the default images used by the Operator. This can be done by setting one or more environment variables in the Quay Operator **ClusterServiceVersion**.

6.1.6.1. Environment Variables

The following environment variables are used in the Operator to override component images:

Environment Variable	Component
RELATED_IMAGE_COMPONENT_QUAY	**base**
RELATED_IMAGE_COMPONENT_CLAIR	**clair**
RELATED_IMAGE_COMPONENT_POSTGRES	**postgres** and **clair** databases
RELATED_IMAGE_COMPONENT_REDIS	**redis**

NOTE

Override images **must** be referenced by manifest (@sha256:), not by tag (:latest).

6.1.6.2. Applying Overrides to a Running Operator

When the Quay Operator is installed in a cluster via the Operator Lifecycle Manager (OLM), the managed component container images can be easily overridden by modifying the **ClusterServiceVersion** object, which is OLM's representation of a running Operator in the cluster. Find the Quay Operator's **ClusterServiceVersion** either by using a Kubernetes UI or **kubectl/oc**:

```
$ oc get clusterserviceversions -n <your-namespace>
```

Using the UI, **oc edit**, or any other method, modify the Quay **ClusterServiceVersion** to include the environment variables outlined above to point to the override images:

JSONPath: **spec.install.spec.deployments[0].spec.template.spec.containers[0].env**

```
- name: RELATED_IMAGE_COMPONENT_QUAY
  value:
quay.io/projectquay/quay@sha256:c35f5af964431673f4ff5c9e90bdf45f19e38b8742b5903d41c10cc7f63
39a6d
- name: RELATED_IMAGE_COMPONENT_CLAIR
  value:
quay.io/projectquay/clair@sha256:70c99feceb4c0973540d22e740659cd8d616775d3ad1c1698ddf71d
0221f3ce6
- name: RELATED_IMAGE_COMPONENT_POSTGRES
  value: centos/postgresql-10-
centos7@sha256:de1560cb35e5ec643e7b3a772ebaac8e3a7a2a8e8271d9e91ff023539b4dfb33
- name: RELATED_IMAGE_COMPONENT_REDIS
  value: centos/redis-32-
centos7@sha256:06dbb609484330ec6be6090109f1fa16e936afcf975d1cbc5fff3e6c7cae7542
```

Note that this is done at the Operator level, so every QuayRegistry will be deployed using these same overrides.

ADDITIONAL RESOURCES

- For more details on the Red Hat Quay Operator, see the upstream quay-operator project.